Skyline parallelization report

Aldo D'Aquino

Parallel and Distributed Systems: Paradigms and Models
Master Degree in Computer Science - Pisa University
a.y. 2018-19

February 10, 2020

Contents

1 Introduction

The aim of the *Parallel and Distributed Systems: Paradigms and Models* course is to learn a mix of foundations and advanced knowledge in the field of parallel computing specifically targeting data intensive applications.

During the course have been presented the principles of parallel computing, including measures characterizing parallel computations, mechanisms and policies supporting parallel computing and typical data intensive patterns.

In this final project we analyse an application and design a parallel implementation of the algorithm, by comparing different design patterns and alternative parallelization strategies and techniques.

We have chosen the Skyline application, for which we provide a sequential version, a parallel version implemented with C++ threads and a parallel version based on the *FastFlow* framework developed by the Computer Science department of Pisa University.

In this report we provide the benchmark of each implementation, and we discuss the scalability of the system and the possible bottleneck and hardware limits.

1.1 Skyline algorithm

The target is to develop an application computing the skyline of a stream of tuples. We consider tuples made of t items. Items at position i have type $type_i$. For each $type_i$, an order is defined by a $>_i$ relation. A tuple $t_1 =< x_1, ..., x_t >$ dominates another tuple $t_2 =< y_1, ..., y_t >$ if and only if at least one component x_i of t_1 is better than the corresponding component y_i of t_2 (i.e. $x_i >_i y_i$) and all the other components of t_1 (i.e. all the x_j such that $j \neq i$) are better or equal (i.e. either $x_j >_j y_j$ or $x_j =_j y_j$). The skyline of a set of tuples is the subset of the tuples that are not dominated by any other tuple.

The skyline stream application considers a possibly infinite stream of tuples. For each window of size w (w consecutive tuples), the application computes the skyline and outputs the skyline onto the output stream. Consecutive windows differ by exactly k item (sliding factor).

Our implementation simplifies the skyline application as follows.

The stream, that is not part of the application, has a fixed length of l. It is made of l tuples of t integers each. The $>_i$ relation is the same for all the t items of the tuple, and is exactly the $>$ relation between integers. A tuple $t_1 =< x_1, ..., x_t >$ dominates another tuple $t_2 =< y_1, ..., y_t >$ if and only if $\forall i \in [1, t]\ x_i \geq y_i$ and $\exists i \in [1, t]\ x_i > y_i$.

Each window has a fixed length w, hence from the stream are extracted $\lfloor \frac{l-w}{k} \rfloor + 1$ windows of w tuple, where l is the size of stream and k the sliding factor.

Here there is an example stream, the windows that will be generated and the resulting skylines.

Parameters: $l = 10$, $w = 3$, $t = 2$, $k = 1$.

Stream: $< 66, 40 >, < 81, 41 >, < 12, 58 >, < 21, 40 >, < 35, 43 >, < 74, 43 >, < 17, 4 >, <$

$96, 62 >, < 92, 48 >, < 98, 59 >$

Window 0: $< 66, 40 >, < 81, 41 >, < 12, 58 >$
Skyline 0: $< 81, 41 >, < 12, 58 >$

Window 1: $< 81, 41 >, < 12, 58 >, < 21, 40 >$
Skyline 1: $< 81, 41 >, < 12, 58 >$

Window 2: $< 12, 58 >, < 21, 40 >, < 35, 43 >$
Skyline 2: $< 12, 58 >, < 35, 43 >$
...

Window 7: $< 96, 62 >, < 92, 48 >, < 98, 59 >$
Skyline 7: $< 96, 62 >, < 98, 59 >$

2 Parallel architecture design

The parallel version consists in a pipeline of three steps:

1. pick one by one the windows from the input stream and pushes them in an input queue;
2. pops the tasks from the input queue, compute the skyline of the windows and pushes the skylines in an output queue;
3. pops and consumes skylines from the output queue.

Since the second step is the one that can scale, it can also be seen as an emitter that picks the windows, a set of workers that compute the skylines and a collector that consumes the skylines, i.e. a pipeline with a single node, a self-organized farm, and another single node.

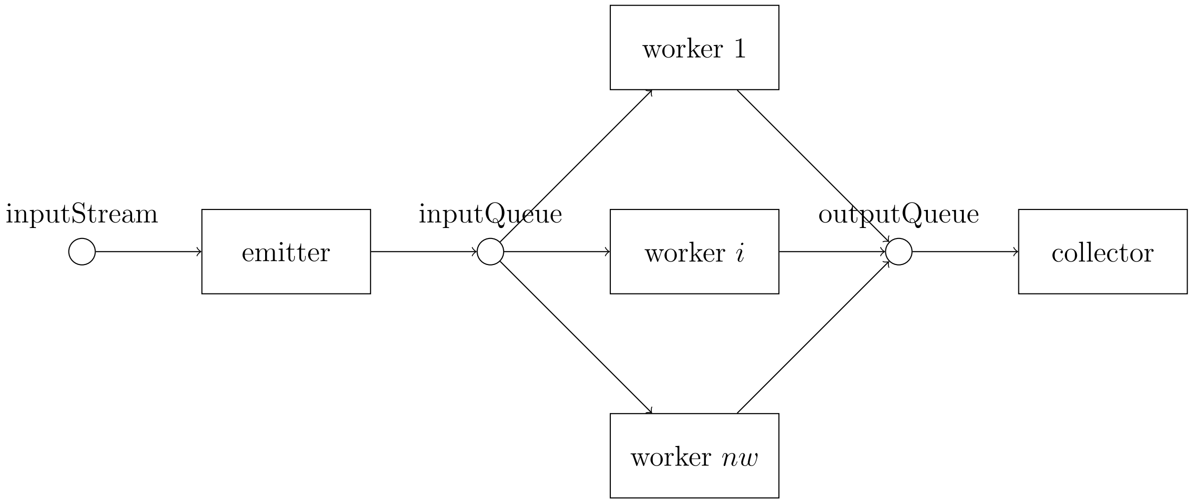

Figure 1: Parallel architecture diagram

We considered also other patterns, but they were discarded for poor performances.

For example, we can simplify the architecture explained above by reducing it to a farm where workers pick windows and consume skylines by theirself, but extracting a windows is a heavier task than popping it from a list, and is not a good idea to do that while holding a lock.

Another option was to switch from a self-organized farm to an emitter-collector farm, where the emitter pushes the windows to a worker-dedicated queue instead of the shared queue. The same for the collector. Anyway, it turns out that the real bottleneck was the memory and this implementation increase the overhead. Since the performance doesn't increase even pushing more windows at time to the workers, also this version was discarded.

3 Performance modeling

In order to model the theoretical performances of the skyline algorithm, we have to consider those input values:

- w the window size, i.e. the number of tuple inside a window;
- t the tuple size, i.e. the number of element inside a tuple;
- k the sliding factor, that means that $window_{i+1}$ starts k tuples after $window_i$;
- l the stream length, i.e. the number of tuple inside the input stream;
- nw the number of workers.
- w_no the number of windows, can be computed from the window size, the stream length and the sliding factor: $\lfloor \frac{l-w}{k} \rfloor + 1$.

From this values we can derive the following costs:

- $T_{extract\ item}$ the time to extract one item from the the input stream;
- $T_{extract\ window}$ the time to extract one window from the the input stream, that is $T_{extract\ item} \cdot w$;
- $T_{extract}$ the time to extract all the windows from the the input stream, that is $T_{extract\ window} \cdot w\_no$;
- $T_{compare\ items}$ the time to compare two items of two different tuple;
- $T_{compare\ tuples}$ the time to compare two tuple, that is $T_{compare\ items} \cdot \mathcal{O}(t)$ (is an upper bound since the optimized algorithm doesn't need to compare all the items if none of the tuple can dominate the other just after two item);
- $T_{compute\ window}$ the time to compute the skyline of a window, that is $T_{compare\ tuples} \cdot O(w^2)$ (is an upper bound since the optimized algorithm doesn't need to compare two tuple if one of them is already dominated by another one);

- $T_{compute}$ the time to compute the skyline of all the windows, that is $T_{compute\ window} \cdot w\_no$;
- $T_{consume\ skyline}$ the time to print and delete the skyline;
- $T_{consume}$ the time to print and delete all the skylines, that is $T_{consume\ skyline} \cdot w\_no$.

3.1 Sequential algorithm

The sequential algorithm picks a window from the input stream, computes and prints the skyline.

```
sequential(inputStream) {
    for (int i = 0 to w_no) {
        window = inputStream.getWindow(i);
        skyline = compute_skyline(window);
        print(skyline);
    }
}
```

Therefore, the cost of the sequential algorithm is $T_{extract} + T_{compute} + T_{consume}$.

3.2 Parallel algorithm

For the sake of simplicity, in the analysis we assume that the number of window w_no is a multiple of nw. Since the time to compute the skyline of a window can differ, we also assume that on average the time for computing the skylines of the assigned windows is more or less the same for each worker.

The cost of the parallel algorithm is $T_{extract} + \frac{T_{compute}}{nw} + T_{consume}$, plus the overhead cost of the additional data structures and locks.

The ideal situation is the one in which $T_{extract} = \frac{T_{compute}}{nw} = T_{consume}$. We reach the maximum scalability when $\frac{T_{compute}}{nw} = max(T_{extract}, T_{consume})$.

If we want to take into account the overhead, we need to introduce the following costs:

- $T_{push\ window}$ the time to push one window into a queue;
- $T_{pop\ window}$ the time to pop an window from a queue;
- T_{push} the time to push all the window into a queue, that is $T_{push\ window} \cdot w\_no$;
- T_{pop} the time to pop all the window from a queue, that is $T_{pop\ window} \cdot w\_no$.

Hence, the cost of the parallel algorithm including the overhead is the following.

$$T_{extract} + T_{push} + \frac{T_{pop} + T_{compute} + T_{push}}{nw} + T_{consume} + T_{pop}$$

From this formula we can easily see how much the application can scale. We stop gain when the threads have to wait to acquire the lock. This happens roughly when the thread have finished his bunch of work before all the other threads have done the pop to take their task, and in the same way on the output queue.
This occurs in one of the following case:

$$T_{compute\ window} + T_{push\ window} < T_{pop\ window} \cdot nw$$
$$T_{pop\ window} + T_{compute\ window} < T_{push\ window} \cdot nw$$

that is

$$T_{compute\ window} < T_{pop\ window} \cdot nw - T_{push\ window}$$
$$T_{compute\ window} < T_{push\ window} \cdot nw - T_{pop\ window}$$

and since

$$T_{pop\ window} \cdot nw - T_{push\ window} \approx max(T_{pop\ window}, T_{push\ window}) \cdot nw - 1 \approx T_{push\ window} \cdot nw - T_{pop\ window}$$

we obtain the following formula.

$$T_{compute\ window} < max(T_{pop\ window}, T_{push\ window}) \cdot nw - 1$$

Therefore, we can understand that the scalability depends on the number w of tuple per window and on the number t of item per tuple. Anyway, since both $T_{compute\ window}$ and $T_{compare\ tuples}$ definitions contain a O notation, we expect this relation to not be linear, to be instead similar to a logarithmic function.

4 Implementation structure and details

4.1 Sequential version

For the sequential version we don't use any particular data structure, except for the `inputStream`, that is a class that extends a vector with a `pickWindow(i)` function and a constructor that automatically generates the tuples in the stream. The `inputStream` is allocated and initialized before running the sequential version and deallocated after, so its cost is not included in the timestamps.

4.2 Parallel version

4.2.1 Threads

We have a thread called *emitter* that calls the `pickWindow(i)` method of the stream and pushes the windows to the workers. The *nw workers* pop a window, compute the skyline and push the result to the collector. The *collector* pop and consume the skylines coming from the workers.

4.2.2 Communication

The communication in implemented with two queue: the `inputQueue`, shared between the *emitter* and the *workers*, and the `outputQueue`, shared between the *workers* and the *collector*.

4.2.3 Synchronization

The `inputStream` is accessed only by the *emitter*, hence doesn't need a mutual exclusion.

The `Queue` class internally uses a `deque` and integrates the lock mechanisms. The lock is taken before popping from or pushing values to the `deque`. Also, a condition variables await for item in `deque` before performing a pop, and releases the lock during the wait. Eventually, a `notify_one` on the condition variable will be performed after a push.

The `EOQ` element notify the next thread (or threads) in the pipeline of the end of the tasks.

4.3 FastFlow version

This version is implemented with a `ff_Pipe` of a `ff_node_t` emitter, a `ff_Farm` of *nw* `ff_node_t` workers and another `ff_node_t` collector.

All the locks, data structures and synchronization method are handled by *FastFlow*.

5 Experimental validation

Tests are executed on the Intel Xeon Phi KNC, with 64 physical cores with a four-way multi-thread support each, for a total of 256 threads.

For the text we have prepared an *auto* mode that automatically runs the benchmarks for the provided input values. It runs the sequential, parallel and FastFlow version, and for the latter two runs test with an increasing number of workers. At the end of the execution, the automatic mode provides a summary of the timestamps, in addition to those printed during the execution.

The time of each test is expressed in millisecond. Each timestamp provided in this document represent the averages of the timestamps of 5 tests performed with the same parameters.

5.1 Sequential algorithm

The sequential version with a window size $w = 100$, a tuple size $t = 10$, a sliding factor $k = 1$ and a stream length $l = 100000$ takes ~ 24000 *msec* on the Intel Xeon Phi.

We also computed the partial times, measuring the time spent in the single operations with this parameter. We implemented the print as a delete of the skyline, since the actual print on standard out or file takes a very long time and is not scalable, hence we found it less interesting to be considered.

Pick the windows from the stream: ~ 88 *msec*, i.e. ~ 0.88 *µsec* each, i.e. $\sim 0,36\%$.
Process the windows: ~ 24000 *msec*, i.e. ~ 240 *µsec* each, i.e. $\sim 99,39\%$.
Delete the skylines: ~ 60 *msec*, i.e. ~ 0.6 *µsec* each, i.e. $\sim 0,25\%$.

Process a window takes ~ 272 times the time to pick it and ~ 400 times the time needed to delete the corresponding skyline. Since the Intel Xeon Phi supports up to 256 cores, we should be able to sped up the application by scaling on the second step, i.e. the computation of the skyline.

5.2 Completion times

The completion times are collected with a constant seed equal to 42, so that at each run we obtain the same input stream. The tuple size t is fixed at 10 and the sliding factor k at 1. The variable parameters are the nw (from 2^0 to 2^8) and the window size w (from 2^7 to 2^{11}). Since we want always the same number of window $w\_no = 12800$ (that is 50 window per worker with 256 workers), the stream length changes with the window size w. Hence, because $w\_no = \lfloor \frac{l-w}{k} \rfloor + 1$, with a fixed w_no the stream size will be $l = (w\_no - 1) \cdot k + w$. For example, with $w = 128$ we have $l = 12927$.

	nw	w = 128	w = 256	w = 512	w = 1024	w = 2048
Sequential	1	4970262	18585713	68950202	258524148	897301533
Parallel	1	5343199	19621995	70448451	257031126	898974927
Parallel	2	2680228	9903538	36605589	134886437	475516158
Parallel	4	1334705	4939809	18432914	67388771	237680980
Parallel	8	668252	2480417	9186436	33835748	120349829
Parallel	16	344069	1244297	4591405	16792859	59501326
Parallel	32	199679	632766	2307841	8501624	30764311
Parallel	64	119624	405146	1362731	4413848	15859510
Parallel	128	193414	290523	950836	3220670	11374157
Parallel	256	200697	246579	797835	2700211	9327988
FastFlow	1	5279013	19650859	72988778	267977425	952241207
FastFlow	2	2654387	9842083	36545246	134163459	475243530
FastFlow	4	1331611	4940133	18263974	67077633	237621958
FastFlow	8	671599	2481758	9156595	33563912	118830059
FastFlow	16	342098	1242880	4613028	16901159	59797504
FastFlow	32	200525	637487	2340830	8449103	30362855
FastFlow	64	168367	557349	2012049	7253576	25046960
FastFlow	128	149153	443139	1488754	5066831	17368437
FastFlow	256	478662	670081	1209638	3929258	13397304

Table 1: Completion times of the sequential version to vary the window size w and of the parallel C++ threads and *FastFlow* versions to vary the parallel degree nw and the window size w.

We can observe that with a bigger window size w our parallel implementations scale better, since the workload of each task is bigger and the overhead is spread over longer times and better mitigated.

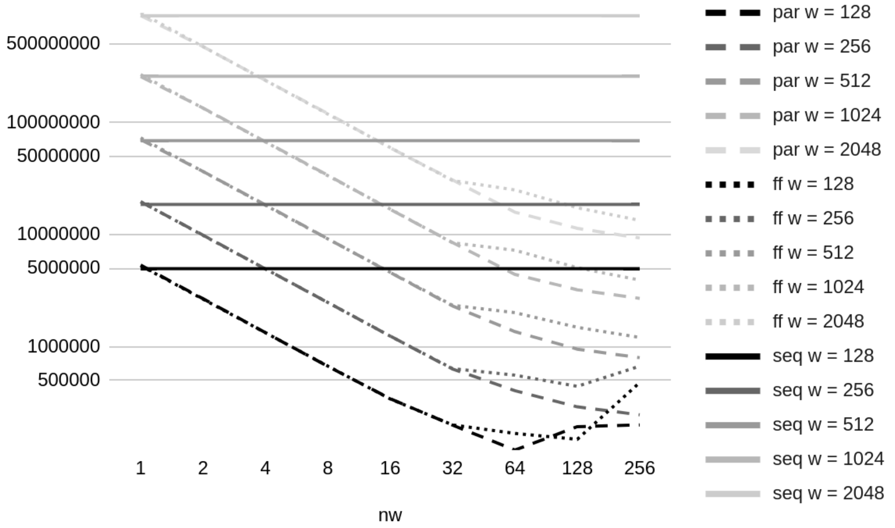

Figure 2: Completion time chart in a logarithmic scale on the y axis, nw on the x axis.

In the Figure 2 we see clearly how the lines tagged as $w = 128$ and $w = 256$ descend with a low nw but than ascend when the nw increases.

This behaviour affects the speedup and the scalability, as we can notice in the following sections.

5.3 Speedup

The *speedup* is computed as $s(p) = \frac{T_{seq}}{T_{par}(p)}$. Ideally, the *speedup* should be the same of the parallel degree p.

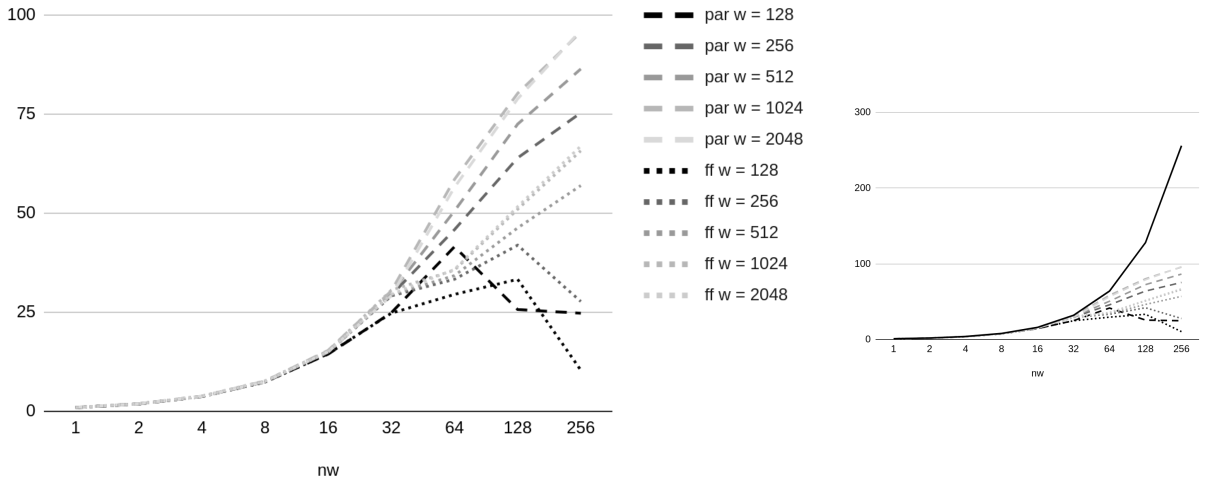

Figure 3: Speedup chart, on the right with the ideal speedup as reference.

In the figure 3 we can see how the C++ threads version is better than the *FastFlow* one. Also, with the C++ threads, only the execution with $w = 128$ stop gain (and start lose) before reaching the 256 workers, that is the machine limit. The *FastFlow* version instead stop gain also with $w = 256$. Moreover, the *FastFlow* implementation need more thread to reach the same speedup of the C++ threads one, and hence we can notice that with $w = 128$ the *FastFlow* speedup start decreasing between 128 and 256 workers, C++ Threads instead between 64 and 128 but after reaching a higher speedup value.

5.4 Scalability

The *scalability* is computed as $scalab(p) = \frac{T_{par}(1)}{T_{par}(p)}$. Ideally the *scalability* should be the same of the parallel degree p.

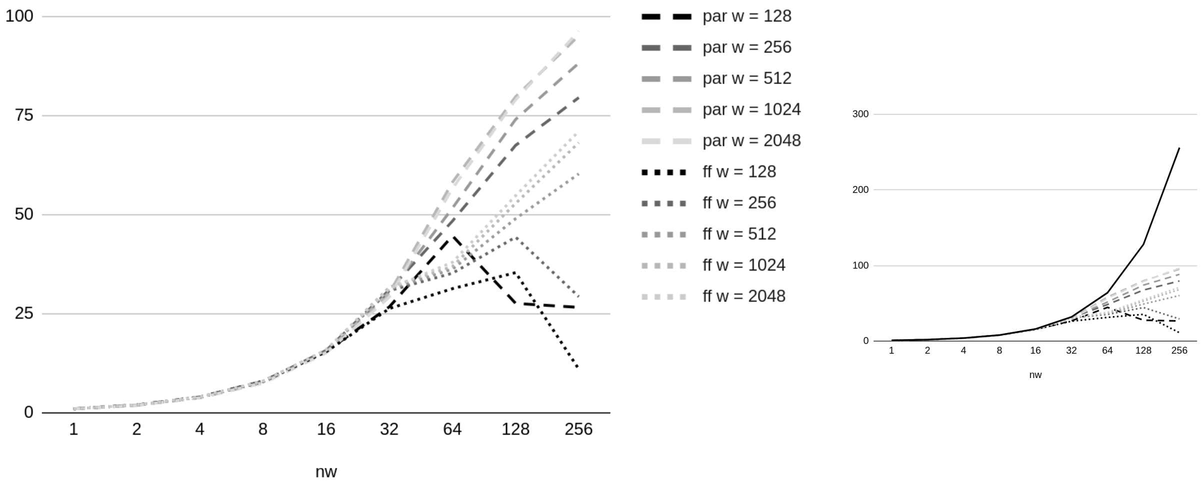

Figure 4: Scalability chart, on the right with the ideal scalability as reference.

The scalability is pretty similar to the speedup of our application. Again, the C++ Threads version is better than the *FastFlow* one. Also, there is a strange behavior of *FastFlow*: while the C++ Threads graph has an "S"-shaped curve, in this one we can notice worsening between 32 and 64 threads, but then there is a new improvement and the function continues almost linearly. This is not as good as the ideal one, that is exponential, but may mean that at some point, with a higher number of cores, using *FastFlow* can be more convenient than our C++ Thread implementation.

5.5 Efficiency

The *efficiency* is computed as $\epsilon(p) = \frac{T_{seq}}{p \cdot T_{par}(p)}$. In an efficient implementation we have $\epsilon = 1$ regardless of the parallel degree p.

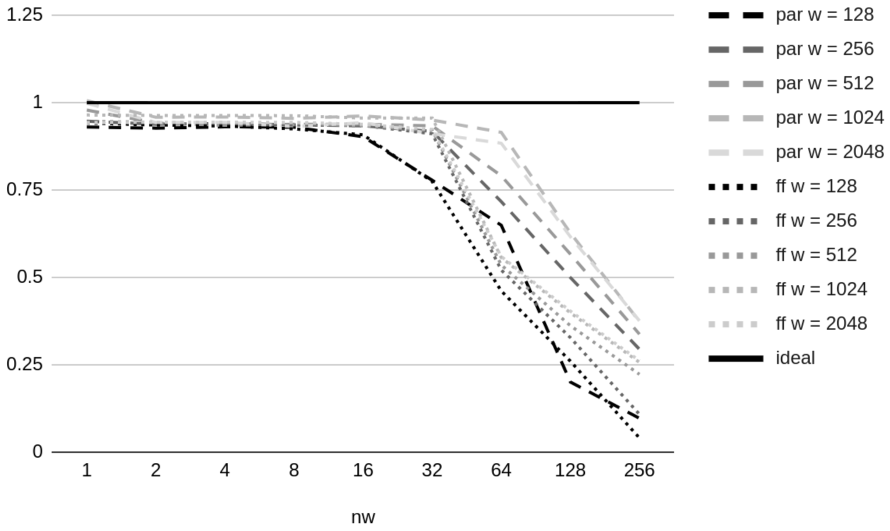

Figure 5: Efficiency chart.

We can clearly see in the figure 5 that we have a high efficiency up to 16 - 32 threads, and then starts decreasing drastically. Increasing the window size w, the efficiency start decreasing at a higher parallel degree nw.

6 Conclusion

By analysing the collected data and the provided plots, we can notice that the program does not scale perfectly. It doesn't have slow start, since for a low parallelism degree the speedup curve in the figure 3 is very few lower than the ideal one, but then the behaviour worsen significantly when approaching the maximum number of cores in the machine. It also depend on the task weight: on heavier tasks this problem is mitigated, because the overhead is spread across a longer computational time.

This is clear by watching the efficiency graph in figure 5, which shows an efficiency drop starting at $nw = 16$ and becoming worse after $nw = 32$ or 64.

That means this algorithm is efficient for machines up to 32 threads, with the best result on 8 - 16 threads, for example on octa-core PC with 2 or 4 contexts per core. On machines with a higher parallel degree we should increment the number of worker nw only for bigger windows. Another approach could be to reduce the CPU frequency: we obtain the same performances we will obtain by running with 32 threads at a higher frequency, but there could be a power save.

If we want to compare the two different approaches, from a performance point of view the native C++ Threads implementation is slightly better than using the *FastFlow*, probably since the STL code is optimized for this task while *FastFlow* is a generic library. On the other hand, from a programmer point of view, the *FastFlow* module was easier and more carefree to be written, even if the amount of code is almost the same, thus in some cases may worth losing some performance.