SPM Final Project: Skyline

Carmine Caserio

1 Introduction

For this project, I've developed a parallel and sequential computation of the skyline algorithm. The parallel implementation has been performed by using both the standard C++ mechanisms and the FastFlow library.

The testing has been performed on the server KNL Xeon Phi with 64 cores, 4 contexts each.

2 Parallel architecture design

My implementation of the skyline algorithm relies on the `sky_line_calc()` function, that is used for computing the skyline.

The architecture chosen for handling the implementation of this algorithm is composed by using the *farm* pattern; the emitter in this case has been called *stream generator*; it creates the stream of tuples and sends it sliding window by sliding window, swiping by an interval of `sliding_factor`. After the skyline computation, all the workers send their skyline to the collector, that prints all the results received. In the implementation by using the FastFlow library, one change has been done to the architecture, that is, the workers, in addition to compute the skyline, print the skyline computed without sending it to the collector, and the architecture built in FastFlow is without collector.

In the standard C++ implementation, each worker has its own input queue and output queue, for receiving the tuples from the stream generator and for sending the tuples in their local skyline to the collector.

The program's arguments that will be used below in describing the architecture are descripted in the README section.

The work carried out by the components of the farm is described in the following:

1. the stream generator generates the tuples one interval at a time (where the interval is the entire sliding window, in the first cycle, otherwise it's just the sliding factor, in all the other cycles) which is then inserted into a local stream. From the local stream the stream generator sends to the workers sliding windows in their own input queues.

   The left and right extremes of the various sliding windows occupy every time different positions in the stream: each one differs by the specified `sliding_factor` value with the next one, so the sliding windows sent are subsequences of the stream that is generated.

   After the stream generator pushes the current sliding window to the worker identified by an integer, it changes the identifier, so that at the next iteration, it'll trigger another worker thread (obviously, if the workers are at least 2).

   Afterwards, it sends to all the workers a special kind of tuple needed for terminating the workers. Finally, the stream generator terminates.

2. the workers pop from their own input queue a set of tuples and compute the calculation of the skyline until they get to pop a number of elements equal to the sliding window size. When this case came, the corresponding output queue of the worker is set to *busy*, one special tuple, obtained by calling the `create_no_tpls_in_skyline_tuple()` function, is sent for communicating to the collector the size of the skyline, the tuples of the skyline are inserted into the output queue and then, the queue is set to free again.

   If the tuple popped from the worker is a tuple of termination type, the output queue of the worker is set to *busy*, the tuple is forwarded to the collector by this output queue and then, the output queue is reset to free; afterwards, the worker terminate its execution.

3. the collector tries to pop from the workers' queues until it finds a not empty queue, by means of the `try_pop()` function, this means that it remains checking for all the worker queues in a non-blocking way, by using the `optional` data type, so that if a sliding window is present, it'll pop both the special kind of tuple and the sliding window, otherwise, it tries checking for the next worker output queue modulo number of workers.

From now on, in the formulas exposed below, sw represents the sliding window dimension, sf represents the sliding factor dimension, sl represents the length of the stream and td represents the tuple dimension.

The sequential time used for the comparison with the parallel version can be shown in the following: it is composed by the time needed for computing the function `sky_line_calc()` for all the tuples in the sliding window, by the time needed for printing the skyline in the file pointer specified by command line, and by the time needed for removing the elements that are no more into the sliding window. This whole computation has to be repeated $sw - (sf - 1)$ times, that correspond to the number of sliding windows that can be created.

In formulae, we can define the T_{seq} as:

$$T_{seq} = \big(sf \times (T_{create\_tuple} + T_{sky\_line\_calc}) + T_{print\_skyline}\big) \times \big(sw - (sf - 1)\big)$$

The timings for the parallel version is shown below:

$$T_{par_{v1}}(k) = sf \times \big(T_{create\_tuple}\big) \times \big(sw - (sf - 1)\big) +$$

$$\frac{\big(sf \times (T_{sky\_line\_calc}) + T_{print\_skyline}\big) \times \big(sw - (sf - 1)\big)}{k}$$

In this version, we don't take into account the time spent into the busy wait by the collector because it's negligible if compared with the costs of the functions that the various threads have to execute. The cost depends on the communication to the collector of the computed skyline and on the creation, in the collector, of the local skyline needed for printing the results communicated from the workers.

3 Project presentation

The project is composed by 4 files, that is:

- `blocking_queue.hpp`: it contains the class which defines the behaviour of the blocking queue as seen in the lectures. In it, there is also the `try_pop()` method that has been implemented by using a busy wait to guarantee a non-blocking behaviour on the queue. It is used by the collector because it can't be blocked on a queue, but it must continue on the other workers' queues;
- `my_tuple.cpp`: it contains the class which defines an element of the stream, it's composed by a vector of integers which represent the values of the tuple and by an integer that represent the position into the stream;
- `skyline.cpp`: it contains the class that defines:
  - the `sky_line_calc()` function exposed at the beginning of the report;
  - the `remove_old_elements()` function discussed in the sequential and parallel formulas;
  - the various functions for creating a normal or a special kind of tuples needed for the send of the results;
  - the `print_skyline()` function used for printing on the file pointer specified all the skyline computed;
  - the sequential version by means of the `start_seq()` function;
  - the parallel version by using C++ standard mechanisms behaviour by means of the `start_par()` function;
  - the parallel version by using FastFlow library by means of the `start_ff()` function;
- `skyline_par_trial.cpp`: it contains the main where the parameters are passed by command line and, depending on the latters, one version among the sequential, the parallel by using C++ standard mechanisms or the parallel by using FastFlow library is chosen and executed.

4 README

For compiling and executing the skyline algorithm, the following steps needs to be executed from bash:

```
unzip skyline.zip -d skyline
cd skyline
cmake .
make
./spm_final 123 1000 2000 1000 P 4
```

In more detail, the usage of the `./spm_final` executable can be exposed in the following:

```
Usage:  ./skytrial seed max_generated_num stream_length sliding_
    window_size sliding_factor path 'S'/'P'/'F' (for seq/par/ff
    computation) [number_of_workers_if_'P'_or_'F']
```

The parameters passed by the command line are the following ones:

- `seed`: for initializing the srand call, needed for the generation of the random values that will compose the tuples;
- `max_generated_number`: for having a maximum value that will be used in the generation of the random values inserted in each tuple;
- `stream_length`: for deciding *a priori* the length of the stream that will be generated;
- `sliding_window_size`: to fix the dimension of the sliding window;
- `sliding_factor`: to fix the dimension of the sliding between a skyline computation and the next one; it's equal to the dimension of the tuple, as per the specification;
- `path`: to fix the file pointer on which printing the computed skylines;
- a character between `'S'`, `'P'` and `'F'`, with which we decide whether computing the sequential version (`'S'`), the C++ parallel version (`'P'`) or the parallel version using the FastFlow library (`'F'`);
- `nw`: this last element is significant only whether the chosen character is `'P'` or `'F'`; it's used to fix the number of workers.

5 Experiments

The experiments have been executed on the 256 cores server KNL Xeon Phi; the fixed parameters chosen for carrying out the experiments are the following ones:

- `seed` = 123
- `max_generated_number` = 100
- `stream_length` = 4.096
- `sliding_window_size` = 1.024
- `sliding_factor` = 128

The values assigned for the sequential and parallel parts are:

char	NW
'S'	-
'P'/'F'	1
'P'/'F'	2
'P'/'F'	4
'P'/'F'	8
'P'/'F'	16
'P'/'F'	32
'P'/'F'	64
'P'/'F'	128
'P'/'F'	256

The values chosen for the experiments are those that allow a good analysis of performances: by putting a too high value for the stream length, the completion time raises quite a lot; by putting an higher value in the sliding factor means to have also an higher number of components in a single tuple, since, following the specification, the value assigned to the sliding factor is the same as the value of the tuple dimension; the sliding window size has been chosen to be at most four times higher than the sliding factor.

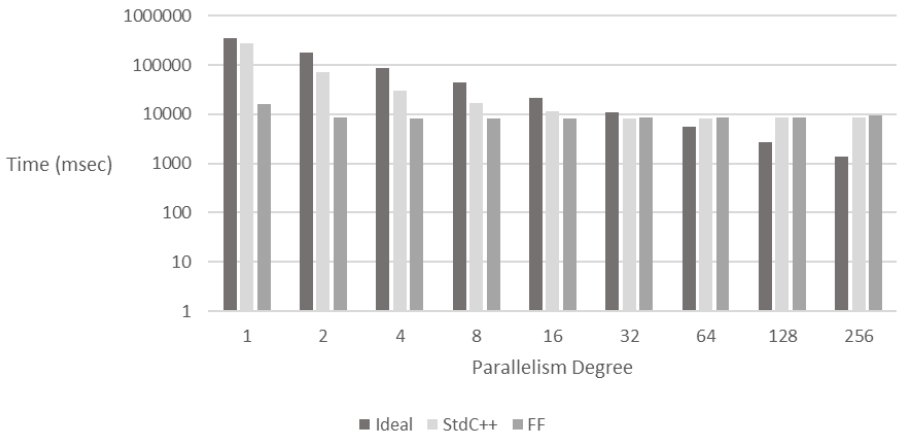
The experiment has been carried out with the aim of verifying the behaviour of the implementation in the two different versions at different parallelism degrees.

In the evaluation of the times, the ideal time has been approximated as the sequential time divided by the parallelism degree.

The time unit used for the timings shown below is the milliseconds.

The timings of the experiments are shown in the following:

NW	Ideal	Std C++	FF
1	348114	270639	15780
2	174057	71850	8635
4	87029	29787	8334
8	43514	16997	8333
16	21757	11239	8343
32	10879	8297	8376
64	5439	8301	8425
128	2720	8442	8442
256	1360	8573	9333



As we can see from the timings obtained by the experiments, the FastFlow version takes less than the standard C++ version up to 32 workers; afterwards, they take the same time. A bottleneck of all the versions is the I/O access when there is the printing of the various skylines.

From the speedup chart, we can see that the initial speedup is greater than the ideal one for both the versions and that from about 32 became worse. The same can be said for the scalability and efficiency charts.

The main difference between the behaviours of the two versions is that the FastFlow version is faster than the one implemented by using only the standard C++ mechanisms up to arriving to the parallelism degree equal to 32, where they start behaving very similarly.

The latency graphic shows us instead that the C++ version is always better than the FastFlow version and better than the ideal version, that has been computed by measuring the time spent for doing a work unit (that is composed by

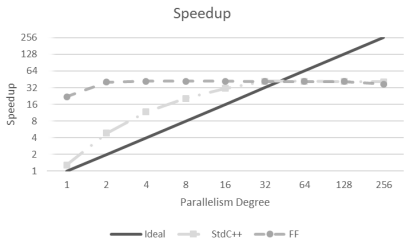


Figure 1: Speedup

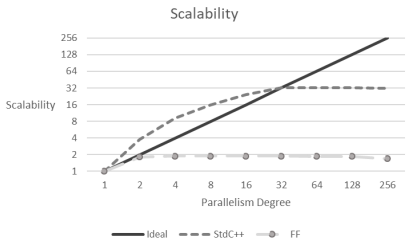


Figure 2: Scalability

Figure 3: Experiments' performances

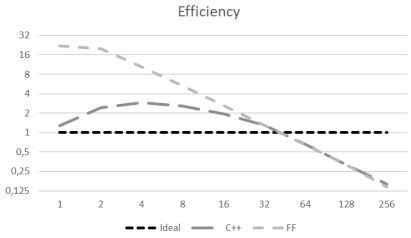


Figure 4: Efficiency

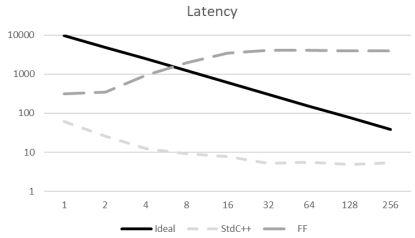


Figure 5: Latency

Figure 6: Experiments' performances

the computation of a sliding window on the sequential version). That measured time gets divided by the parallelism degree for obtaining the ideal latency.

6 Conclusion

This project has been developed for the computation of a parallel version of the skyline algorithm. The parallel implementation scale very well, but it does not respect the theoretical limits. A reason may be that the sequential version may be much faster, since the sequential version is the starting point for the analysis of the two parallel versions.

Future works may be related to the implementation and analysis of other parallel structures, such as the *map-reduce*, or to the implementation of another version which provides a parallel I/O access, for removing the I/O bottleneck.