Francesca Sbolgi 554920
francesca.sbolgi@gmail.com

Parallel and Distributed Systems: Paradigms and Models Final Project

SKYLINE STREAM APPLICATION

Academic Year 2018/2019

1 Problem Description

The aim of the project is to compute the skyline of a stream of tuples. We started by implementing a sequential version and, once we saw which computations took the longest, we tried to optimize it by performing them in parallel.

So, first of all, we decided a set of important properties to be exploited in the computation:

- Skyline changes can occur only when a new tuple arrives or some skyline point expires;
- All points dominated by an incoming tuple r can be safely discarded since they are guaranteed not to appear in the skyline in the future;
- An arriving tuple r cannot be directly discarded even if it is dominated by some existing tuple r_0 since after r_0 expires it may become part of the skyline later;
- A tuple r can appear in the skyline for at most a single continuous time interval, which can be estimated upon the arrival of r. And it's given by the expiration of dominating tuples and its expiring time. If a new dominating tuple arrives it may only shrink this interval.

With these properties in mind we tried to implement a solution that minimize the memory consumption and reduce the number of tuple comparison.

The first point was respected by storing only the tuples that are currently in the skyline and those that may became part of the skyline at some point. The second was accomplished by doing some preprocessing and storing some additional information for each tuple.

1.1 Variables

- type of elements in the tuple
- tuple size
- window size
- sliding factor
- stream length

2 Sequential Implementation

The main idea behind the implementation is to sort each new tuple generated to either the current skyline or to a list, that we decided to call *rest*, that contains the tuples that may end up in the skyline later.

We can simplify the implementation in four phases to be repeated for each tuple contained in the stream.

1. Generate a new tuple t and fill it with m random integers;
2. Compare t with all the tuples ts_i present in the skyline list;
3. Compare t with all the tuples tr_j present in the rest list;
4. Print all the elements of the skyline list.

The first phase is quite easy to understand. The only thing required is the size of the tuple and a seed to generate the pseudo random numbers.

The second phase scans the whole skyline list and compare each tuple ts_i with t. This means that for each couple of tuples a pairwise comparison of the elements is performed. After this comparison three results are possible: either t dominates ts_i, the opposite or none is dominating the other. If t is dominated it means that it should be put in the skyline and the other should be removed. If it's the opposite then t is put in the rest list with a tag that signals when the [TODO]

The third phase [TODO]

The forth phase can be also inserted in the other two in the sense that once a tuple is not removed from the skyline it can be already printed without having to scan the list twice.

The skyline and the rest list are implemented through a simple linked list. Indeed we needed a data structure that allowed to append a new element at the top of the list and to delete at any point of the list with just O(1) cost. The implementation of the linked list was loosely taken from `https://www.geeksforgeeks.org/data-structures/linked-list/#singlyLinkedList`

This structure makes sense especially if a large portion of the window is maintained between steps. If this is not true we could discard every thing and simply start again at the next step. We can perform some experiments to see if this is the case by fixing the window size to 10,000 and varying the sliding factor using the values 1,000 2,500 5,000 and 10,000. And then estimate for which values which approach is more convenient.

2.1 Execution Times

Before implementing the parallel version we need to measure the execution time of each phase in order to decide which part could benefit more from having more workers.

Our goal is to evaluate the differences between the sequential and the parallel execution exploiting different metrics. The variables are the number of workers for each component of the parallel implementation.

In order to measure some executions time we have to set some values for the variables:

- The type of the elements in the tuple is fixed to int, so we can measure how much it takes to compares to elements. Indeed if the type is something more complex, such as an object, the distribution of the computational complexity could be spread differently;
- We try different values for the window size such as 100 and 10,000;
- We can do the same for the sliding factor using respectively 50 and 2,000. This way we can also compare the performance if the window at each step is discarded half or a smaller fraction (one over 5 for instance);

3 Parallel Implementation

There are many ways to parallelize this application.

The most natural one would be to build a pipeline in order to process more tuples at the same time. However this approach is not the easiest one. Indeed the problem is that each tuple must be compared with all the previous ones and therefore it should be implemented a kind of collector that compares the tuples exiting the pipeline.

The skeleton in this case, considering the phases *f1*, *f2*, *f3* and *f4* described in the previous chapter and an additional phase *col* that compares the elements currently in the pipe, it could be shaped like this: pipe(seq(f1), seq(f2), seq(f3), seq(col), seq(f4)). Of course the phases could also be joined using a composition of the phases. For example we could join the last two steps since they shouldn't take too much time: pipe(seq(f1), seq(f2), seq(f3), comp (col, f4)).

The first phase shouldn't take too long since for each tuple has just to generate m random integers, where m is the tuple size.

The phases f2 and f3 are instead more computational complex. Especially if the window w or the tuple size m are large, they have to perform in the worst case w * m comparisons. So, instead of focusing on stream parallel pattern we could try to optimize this phases using a data parallel approach. Since we are dealing with lists one option could we to split the lists among the workers in order to perform the comparison, hence doing something like seq(seq(f1), map(f2), map(f3), seq(f4)). This way we could get rid of the collector since we are not processing multiple elements at the same time but instead simply the computation for each element would take less time.

3.1 Parallel Model

As we have seen the phases that takes the longest are the scanning of the two

3.2 Implementation using C++ Threads

3.3 Implementation using FastFlow

4 Project Structure

The project is structured in 3 folders: sequential, parallel and fastflow. Each contains a full implementation of the problem with its own make file. Some functions useful in all the three versions are placed in the main folder.

4.1 How to Test

5 Performances

5.1 Sequential

n = 100k els fixed. Total time for each phase

tuple size	window size	sliding factor	generate (f1)	scan sky (f2)	scan rest (f3)
10	1,000	100	2K μs	1.1M μs	90k μs
100	1,000	100	30K μs	1.8M μs	10 μs
10	1,000	500	2K μs	1.1M μs	90K μs
100	1,000	500	35K μs	1.9M μs	10 μs
10	10,000	1,000	3K μs	9M μs	2M μs
100	10,000	1,000	165K μs	30M μs	50 μs
10	10,000	5,000	5K μs	9.2M μs	2.2M μs
100	10,000	5,000	170K μs	30Mμs	100 μs

6 Bibliography