Efficient Continuous Skyline Computation on Multi-Core Processors Based on Manhattan Distance

Ehsan Montahaie, Milad Ghafouri, Saied Rahmani, Hanie Ghasemi, Farzad Sharif Bakhtiar, Rashid Zamanshoar, Kianoush Jafari, Mohsen Gavahi, Reza Mirzaei, Armin Ahmadzadeh, Saeid Gorgin*

School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran.
HPC@ipm.ir

***Abstract*— The continuous Skyline query has recently become the subject of the several researches due to its wide spectrum of applications such as multi-criteria decision making, graph analysis network, wireless sensor network and data exploration. In these applications, the datasets are huge and have various dimensions. Moreover, they constantly change as time passes. Therefore, this query is considered as a computation intensive operation that finding the result in a reasonable time is a challenge. In this paper, we present an efficient parallel continuous Skyline approach. In our suggested method, the dataset points are sorted and pruned based on Manhattan distance. Moreover, we use several optimization methods to optimize memory usage in comparison with naïve implementation. In addition, besides the applied conventional parallelization methods, we partition the time steps based on the number of available cores. The experimental results for a dataset that contains 800k points with 7 dimensions show considerable speedup.**

Keywords*— *Skyline computation; Manhattan distance; Multi-core processors

I. INTRODUCTION

The Skyline query is a useful operator in many data-intensive applications such as product or restaurant recommendations [1], route planning for road networks [2], graph analysis network [3] and data exploration [4]. For a simple dataset with just one dimension, the Skyline operation is equivalent to Maximum or Minimum operation. However, in a dataset with two or more dimensions, to find the best records with specific criteria, the impact of all dimensions should be considered. Selecting a hotel for a holiday, based on two parameters of cost and distance to the beach, is a classic example that used for explaining the Skyline operator. To choose the most suitable hotels (to be both cheap and close to the beach), the Skyline operator provides the best cases for trade-off in price and distance to the beach. For example, in Figure1, the hotel with tag "A" is more appropriate than the hotel with tag "B", since it is not only closer to the beach, but also it is cheaper than "B", in this case, said "*A dominates B*". In contrast, the hotel with tag "C" and all other points on the black line in Figure 1 (i.e. Skyline points) present the alternatives with lower cost, however, more distance to the beach. In this example, just two parameters are considered and the dataset contains two dimensions ($m = 2$, in this paper, the number of dimensions in a dataset is shown with m). However, more dimensions must be added to the dataset, if a vacationer wants to ponder about other aspects of the hotel (e.g., the hotel's service quality).

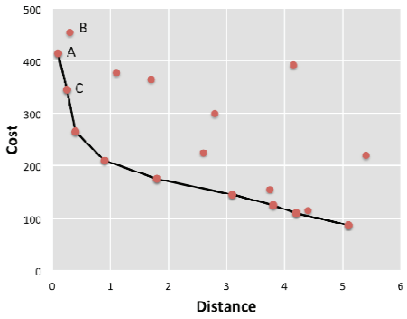

Fig. 1. The Skyline of hotels with two parameters [5]

In the aforementioned example, the dataset is static which means no data point will be added or removed over time. However, real datasets are very large and dynamic which are frequently changed over time. Finding the Skyline points for these types of datasets is called "continuous" or "continuous time-interval" Skyline computation.

The subject of MEMOCODE 2015 Design Contest challenge is to find the best approach (pure-performance or cost-adjusted-performance) for computing the Skyline of a multi-dimensional dynamic dataset. Based on the contest problem definition, the dataset D consisting of n elements $d_0, d_1, \dots, d_{n-1}$, each with m dimensions, that $n = 800K$ and $m = 7$. Elements in D can be represented as a 16-bit unsigned value. In addition, the time is modeled as a series of discrete time-steps and the dataset's changes are shown with activation time a and deactivation time v. Participants must find the Skyline, for each time-step t, among all data in D that have activation time $a \leq t$ and deactivation time $v > t$. The output of should contain the indices (in D) of the Skyline elements and the number of Skyline entries found per time-step.

In this contest, we chose multi-core CPU over GPU and FPGA. The data transfer between CPU and GPU is too high compared to the total running time of the algorithm on a multi-core system (it needs about 6 seconds for a copy from memory to GPU). Also, for FPGA, not only it has data transfer limitations, but also, it is hard to place the whole design on the FPGA hardware.

The rest of this paper is organized as follows: in Section II the previous works on designed implementation of Skyline computation are reviewed. We present our proposed methods in Section III and the experimental results are presented in Section IV. Finally, Section V contains our concluding remarks.

* S. Gorgin is also affiliated with Iranian Research Organization for Science and Technology (IROST), Tehran, Iran.

II. Related Works

The Skyline computation idea was first introduced in [6], which uses a divided-and-conquer approach. The first parallel implementation and high dimensional Skyline computation was proposed in [7] and [8], respectively.

In 2001, Borzsonyi et al. [9] introduced Skyline operator to database community that attracted significant research attention. They probed the impact of the standard indexing structures like B-tree and R-tree for computing Skyline queries.

Recently, many algorithms are presented to get the Skyline points, as mentioned in the previous section they can be categorized in two separate groups of static and dynamic or continuous. The static algorithms implemented in high performance platforms such as multi-core CPU [10], many core GPU and FPGA. There are different approaches for Skyline algorithm, which the simplest one was discussed in [9]. This approach, named block nested loop, explores all data points and computes Skyline to find non-dominated points. The GPU-based implementation of this method was proposed in [11], named GNL (GPU-based nested loop). Obviously, searching a large dataset consumes a huge amount of computing power. Therefore, many fast algorithms were introduced in order to decrease the computation time and complexity. One of the algorithms for this problem is Hashcube [12]. Also, Lattice and Skycube are data structures that used in [13], to manage complexity. For better access to memory in multi-core processors, the hybrid algorithm are presented that flattens tree structure into an array [14] [15]. A sorting based data-parallel Skyline algorithm was introduced that exploits the computational power of the GPU[16]. Bøgh et.al introduced a non-recursive partition-based algorithm which called SkyAlign [17]. Also, a hardware implementation of the Skyline algorithm on FPGA was introduced to allow multiple operations to be executed on the same working set in parallel [18].

On the other hand, Lin et al. [19] concentrate on continuous Skyline computation which incorporates a heap to remove elements that have slipped outside the sliding window. The sliding window involves data points which their arrival and expiration times are valid for a specific interval time. Morse et al. [20] improve approach of [19] and present an efficient and scalable algorithm for evaluating Skyline queries, called LookOut. The LookOut algorithm takes advantage of the quad-tree index since it is much more efficient index structure for evaluation. Also, the non-overlapping partitioning feature of quad-tree causes a natural decomposition of space that can more effectively prune the index nodes that must be searched.

III. Proposed Methods

In this section, we present our techniques that reduce the running time of the continuous Skyline computation. Profiling reference implementation shows computing comparison between points is the bottleneck of the algorithm. It is needed a huge amount of computation and memory access. These computations include four nested loops with the time complexity of $O(mtn^2)$ where m, t, and n are the number of dimensions, the number of time steps, and the number of points, respectively. Another challenge in this problem is memory access that can be relieved by optimizing memory usage. Therefore, by reducing the number of effective points and parallelization, the time complexity can be reduced. We describe our proposed techniques in the following sections.

A. Initialization step

The given dataset is sorted based on arriving time; we sort this dataset base on expiration time by quick sort algorithm. For working on data sets, we use the Set template class which is provided by the Standard Template Library (STL). This class is typically implemented via a Red-black tree. The Red-black tree does not have data race problem and can be used for sorted data with $O(n \log n)$ complexity.

We use sets of sorted points on arrival and expiration time for optimal detecting of changes of active points. In each time step, by traversing on the sorted arrays, the points that should be added (removed) to (from) the current set of active points are detected. It should mention that it has a linear complexity since each element of arrays has just one access. The Algorithm 1 shows the pseudo code of above mentioned procedure to obtain Skyline points. The details of "Update_Skyline" function are explained in following section.

Algorithm 1: Skyline Computation

```
1:  Arrival_p = 0, Expiration_p = 0;
2:  FOR (t = start_time TO end_time) DO
3:      While ( Arrival_time [Arrival[Arrival_p]] <= t){
4:          Arrival_nodes_list.add(Arrival[Arrival_p]);
5:          Arrival_p + + ; }
6:      While (Expiration_time [Expiration[Expiration_p]] <= t) {
7:          Expiration_node_list.add(Expiration[Expiration_p])
8:          Expiration_p + +; }
9:      Update_Skyline(Arrival_nodesss_list, Expiration_node_list);
10: END DO;
11: RETURN;
```

B. Updating Skyline method

Let S represent a set of Skyline points which are subsets of D ($S \subseteq D$). Also, we symbolize "x dominate y" by $x \prec y$. The "Update_Skyline" function has two main parts, adding and removing procedures, which are described in below:

1) Adding procedure

In a naïve implementation, for adding a new Point P to Skyline, it should be examined with all of D points.

$$P \text{ is new skyline point} \Leftrightarrow \nexists x \in D\ (x \prec P) \qquad (1)$$

$$P \text{ is not new skyline point} \Leftrightarrow \exists x \in D\ (x \prec P) \quad (2)$$

Using the above propositions is inefficient because it needs to check all points of dataset. Hence, to reduced search space, we can use following propositions.

$$S \subseteq D \Longrightarrow \exists x \in S\ (x \prec P) \Leftrightarrow \exists x \in D\ (x \prec P) \Longrightarrow$$

$$\exists x \in S\ (x \prec P) \Leftrightarrow P \text{ is not new skyline point}$$

On the other hand, point P is a Skyline point when it not dominated by any point in Skyline.

$$\forall x \in D\ ((x \in S) \vee (x \notin S)) \xrightarrow{base\ on\ eq.2}$$

$$\forall x \in D\ \ (x \in S) \vee \big(\exists y \in S\ (y \prec x)\big) \Longrightarrow$$

$$(x \prec P) \Leftrightarrow \Big((x \in S) \vee \big(\exists y \in S\ (y \prec x)\big)\Big) \wedge (x \prec P) \Longrightarrow$$

$$(x \prec P) \Leftrightarrow \Big(\big(x \in S \wedge (x \prec P)\big) \vee \big(\exists y \in S\ (y \prec x) \wedge (x \prec P)\big)\Big)$$

Based on transitive property, we can write:

$$(x \prec P) \Leftrightarrow \Big(\big(\exists x \in S\ (x \prec P)\big) \vee \big(\exists y \in S\ (y \prec P)\big)\Big) \Longrightarrow$$

$$(x \prec P) \Leftrightarrow \exists y \in S\ (y \prec P) \Longrightarrow$$

$$\exists x \in D\ (x \prec P) \Leftrightarrow \exists y \in S\ (y \prec P) \overset{Negation}{\Longrightarrow}$$

$$\nexists x \in S\ (x \prec P) \Leftrightarrow \nexists x \in D\ (x \prec P) \Longrightarrow$$

$$\nexists x \in S\ (x \prec P) \Leftrightarrow P \text{ is new skyline point}$$

This method helps us to decrease search space from all dataset D points to quite smaller one (just Skyline S points).

2) Removing procedures

For removing a point, two different situations are occurred. First, point P is not a member of S; so it can be easily removed without any effect. Second, point P is a member of Skyline set. In this situation, the Skyline set may change since the dominated points by the recent removed point find the chance of being a new Skyline member. We use sorted data based on Manhattan distance to reduce the search space.

According to definition of "dominate" function the element A dominates element B when:

- $A \neq B$
- $A[i] \leq B[i]$ for $i = 0$ to $m - 1$ (in m dimensions)

Therefore, based on Manhattan distance:

$$A \text{ dominate } B \overset{if}{\Rightarrow} \sum_{0}^{m-1} A[i] < \sum_{0}^{m-1} B[i]$$

And obviously:

$$\sum_{0}^{m-1} A[i] \geq \sum_{0}^{m-1} B[i] \Rightarrow A \text{ does not dominate } B$$

Regarding the above equations, the dominated points by P, have greater Manhattan distance than P and the point which dominates P, have smaller Manhattan distance than P. Thus, by sorting the points based on their Manhattan distance, we can find the pruning boundary points by a logarithmic search and start tracing from this points. For more clarification, we represent this subject as an example in a two dimension dataset in Fig. 2. Based on Manhattan distance the points which are placed in the triangle area are pruned. In addition, by removing the point P from the Skyline, the candidate points for being new Skyline member are just placed in the rectangle area of Fig. 2 and the other points are pruned. By using this pruning, we need check only some points which are placed next to the point P in our data structure.

We can summarize above explanation to make a candidate set in below propositions.

$$\text{candidate} = \{x \in D \mid P \prec x\}$$

$$newS = S \cup \{x \in candidate \mid \nexists x \in S\ (x \prec P)\}$$

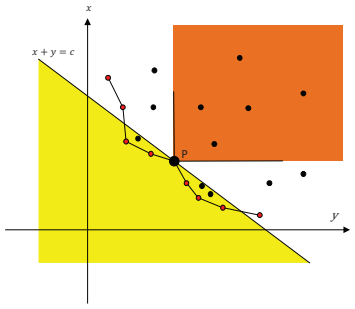

Fig. 2. Using Manhattan distance for pruning points.

C. Parallel implementation details

Multi-core platforms can help to achieve a lower running time for continuous Skyline computation. We study optimization techniques related to the distribution of computation over multi-core processors. We investigate the effect of different load-balancing strategies. In our suggested implementation, the workloads are divided among the cores; when the computation of one core is finished, it takes a block of the other remaining tasks. Therefore, an efficient load balancing is provided. In addition, according to the number of available cores, we parallelize suggested approach over time-step, with sufficient overlap and two different solutions are proposed two handling the overlaps.

1- Static solution: In this solution, the overlap value is fixed.

2- Dynamic solution: In this solution, the overlap value is computed based on dataset elements.

In our implementation, the most widely used function is "dominate" that return a Boolean value based on status of two points, by considering all dimensions. To improve running time of this function, we add a mask bit to the most significant position of each dimension value, and represent all dimension value beside the mask bits in a 128-bit vector. After that instead of the comparison operator for each dimension, we can subtract two points' vectored data. The subtraction result in mask bits position is used to define the dominate function result. To simplify implementation, we used two 64-bit operators instead of a 128-bit operator. These optimization techniques have an important effect on performance, especially on parallel implementation since data locality is increased.

IV. Experimental Result

In this section, we first overview our hardware platforms and then present our experimental results on these platforms. We implemented our algorithm on multi-core systems. Here is the list of our multi-core platforms.

- Intel Core i5, 2410M with two cores @ 2.30GHz.
- Intel Core i7, 3540M with two cores @ 3.00GHz.
- Intel Core i7, 960 with four cores @ 3.20GHz.
- Intel Xeon x5650 with six cores @ 2.66GHz.
- Intel Xeon E5-2650 with ten cores @ 2.00GHZ.
- AMD Opteron 6386 SE with sixteen cores @ 2.80GHz

We took advantage of multi-core platforms via OpenMP programming model. To gain better efficiency, we used Intel compiler and OpenMP4.4.

The platforms, execution time, prices, and performance × cost on each platform for the large dataset with $8{,}00K$ element that each element has 7 dimensions ($N = 8{,}00K$, $m = 7$) are shown in Table I. The 2.66 GHz Intel X5650 is the best in performance × cost, while the best performance is provided by the 2.80GHz AMD Opteron 6386 SE processor.

TABLE I. EXECUTION TIME, PRICES, AND PERFORMANCE×COST FOR EACH PLATFORM

Platform	Time (sec)	Cost in stock ($)	Performance×cost
Intel Corei5-2410M	22	28	616
Intel Corei7-3540M	15	150	2250
Intel Corei7-960	7.8	90	702
Intel Xeon X5650	3.5	81	**283.5**
Intel Xeon E5-2650	2.5	700	1750
AMD Opteron 6386 SE	**1.4**	791.11	1107.5

Table II shows execution time of naïve implementation on the 3.00GHz Intel Corei7-960 processor. In addition, the static and dynamic solutions (See Section III, part C) are compared that their difference is negligible.

TABLE II. EXECUTION TIME OF STATIC AND DYNAMIC SOLUTION FOR EACH PLATFORM

Design	Platforms	Time Dynamic (Sec)	Time Static (Sec)
Naive	Corei7-960	604800	604800
Proposed Solution	Corei5-2410M	23.1	22.0
	Corei7-960	8.6	7.8
	Xeon 5650	3.9	3.5
	Xeon 2650	3.1	2.5
	AMD Opteron 6386 SE	1.9	1.4

V. CONCLUSION

In this paper, we presented a parallel algorithm for Skyline computation with the dynamic dataset. In our proposed method, instead of examining all dataset points for updating Skyline, in add cases just the Skyline points are considered and in remove cases, by using an effective pruning algorithm, a limited subset of dataset is checked. We used the Set template class from the STL that provides several facilities for add/delete operation since it keeps the dataset sorted based on Manhattan distance. Moreover, we utilized the SIMD feature of processors by loop unrolling and bitwise operations for comparing two points. For more parallelization, we partitioned the time steps based on the number of available cores, with dynamic and static overlaps. Finally, we tested our proposed method on six different multi-core platforms. The experimental results are shown significant speedup.

REFERENCES

[1] T. Lappas and D. Gunopulos, "Efficient Confident Search in Large Review Corpora," in *Machine Learning and Knowledge Discovery in Databases*. vol. 6322, pp. 195-210, 2010.

[2] H. P. Kriegel, M. Renz, and M. Schubert, "Route Skyline queries: A multi-preference path planning approach," *in the Proc. of 26th International Conference on Data Engineering (ICDE)*, pp. 261-272, 2010.

[3] L. Zou, L. Chen, M. T. Özsu, and D. Zhao, "Dynamic Skyline Queries in Large Graphs," in *Database Systems for Advanced Applications*. vol. 5982, pp. 62-78, 2010.

[4] S. Chester, M. L. Mortensen, and I. Assent, "On the suitability of Skyline queries for data exploration," *in the Proc. of 1st International Workshop on Exploratory Search in Databases and the Web*, pp. 161-166, 2014.

[5] P. Milder. (2015). *MEMOCODE 2015 Design Contest: Continuous Skyline Computation.* Available: http://www.ece.stonybrook.edu/~pmilder/memocode/

[6] H. T. Kung, F. Luccio, and F. P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM,* vol. 22, pp. 469-476, 1975.

[7] I. Stojmenović and M. Miyakawa, "An optimal parallel algorithm for solving the maximal elements problem in the plane," *Parallel Computing,* vol. 7, pp. 249-251, 1988.

[8] J. Matoušek, "Computing dominances in E^n ," *Journal Information Processing Letters*, vol. 38, pp. 277-278, 1991.

[9] S. Borzsony, D. Kossmann, and K. Stocker, "The Skyline operator," *in the Proc. of 17th International Conference on Data Engineering*, pp. 421-430, 2001.

[10] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "APSkyline: Improved Skyline Computation for Multicore Architectures," Database Systems for Advanced Applications Lecture Notes in Computer Science, vol. 8421, pp 312-326, 2014.

[11] C. Wonik, L. Ling, and Y. Boseon, "Multi-criteria decision making with Skyline computation," *in the Proc. of 13th International Conference on Information Reuse and Integration (IRI)*, pp. 316-323, 2012.

[12] K. S. B, S. Chester, Darius, idlauskas, and I. Assent, "Hashcube: A Data Structure for Space- and Query-Efficient Skycube Compression," *in the Proc. of 23rd ACM International Conference on Conference on Information and Knowledge Management*, 2014.

[13] J. Lee and S.-W. Hwang, "Toward efficient multidimensional subspace Skyline computation," *The VLDB Journal,* vol. 23, pp. 129-145, 2014.

[14] H. Im, J. Park, and S. Park, "Parallel Skyline computation on multicore architectures," *Information Systems,* vol. 36, pp. 808-823, 2011.

[15] S. Chester, D. Sidlauskas, I. Assent, and K. S. Bogh, "Scalable parallelization of Skyline computation for multi-core processors," *in the Proc. 31st International Conference on, Data Engineering (ICDE)*, pp. 1083-1094, 2015.

[16] K. S. B, I. Assent, and M. Magnani, "Efficient GPU-based Skyline computation," *in the Proc. of the 9th International Workshop on Data Management on New Hardware*, 2013.

[17] K. S. B, S. Chester, and I. Assent, "Work-efficient parallel Skyline computation for the GPU," *in the Proc. of VLDB Endow.,* vol. 8, pp. 962-973, 2015.

[18] L. Woods, G. Alonso, and J. Teubner, "Parallel Computation of Skyline Queries," *in the Proc. of 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1-8, 2013.

[19] L. Xuemin, Y. Yidong, W. Wei, and L. Hongjun, "Stabbing the sky: efficient Skyline computation over sliding windows," *in the Proc. of 21st International Conference on Data Engineering,* pp. 502-513, 2005.

[20] M. Morse, J. M. Patel, and W. I. Grosky, "Efficient continuous Skyline computation," *Information Sciences,* vol. 177, pp. 3411-3437, 2007.