The Polygonal Partial-Sum Problem

Yuan Tang [*] [§], Rezaul A. Chowdhury [†], Charles E. Leiserson [§]

[*] School of Computer Science

Fudan University, Shanghai 200433, P. R. China

yuantang@fudan.edu.cn

[†] Department of Computer Science

State University of New York at Stony Brook

rezaul@cs.stonybrook.edu

[§] MIT Computer Science and Artificial Intelligence Laboratory

32 Vassar Street, Cambridge, MA 02139

{yuantang, cel}@mit.edu

Abstract

Algorithms in the literature for performing range queries on multidimensional grid typically assume that the ranges are orthogonal. This paper extends the algorithm to cover non-orthogonal range queries. Our algorithm works for operations drawn from an arbitrary semigroup. Querying the "sum" of all the grid points within a given k-line bounded polygon in a 2-dimensional grid with N grid points can be accomplished in $\Theta(k\alpha^2(N))$ time using $\Theta(kN)$ preprocessing space and time, where $\alpha(N)$ is a functional inverse of Ackermann's function. Our algorithm extends A. Yao's one-dimensional data-reduction method to higher dimensions, as opposed to performing dimension reduction, as is done by Chazelle and Rosenberg.

Index Terms

polygonal partial sum problem, semi-group, multi-dimensional algorithm, parallel algorithm, meta-algorithm, higher-order function

The Polygonal Partial-Sum Problem

I. INTRODUCTION

We consider the ***polygonal partial-sum problem*** in which a static 2-dimensional grid A with N entries chosen from a commutative semigroup $(S,\oplus)$ is given and can be preprocessed for subsequent querying. Each query designates a polytopal region q of grid coordinates, and the problem is to compute the "sum" of the points in the region:

$$\sum_{(k_1,\ldots,k_d)\in q} A[k_1,\ldots,k_d] \ . \tag{1}$$

The polygonal query is restricted in that the normals to the sides of the polygon must be known at preprocessing time. Polygonal query regions generalize the assumption common in the literature of orthotope-shaped queries (boxes). For example, in two dimensions, a query region could be triangular with sides of known slope or even a complex polygon.

Since $(S,\oplus)$ is a commutative semigroup, $\oplus$ is an associative and commutative operator. No additional restrictions are imposed on this semigroup as is sometimes done for similar problems in the literature:

- We do not assume idempotence ($a\oplus a = a$ for $a\in S$).
- We do not assume any partial or total ordering of the elements of S.
- We do not assume the existence of an inverse operation (subtraction), since otherwise a trivial $\Theta(N)$-time, $\Theta(N)$-space preprocessing algorithm for solving this problem exists [1].

We use the ***arithmetic model*** for complexity analysis, the same complexity model used by A. Yao [1], [2] and Chazelle and Rosenberg [3], [4]. In this model, the space bound of the preprocessing algorithm is counted in units of semigroup elements. The preprocessing time and query time count the number of arithmetic (semigroup) operations performed, ignoring the time needed to find the proper memory cells [1][1].

Many results have been reported in the literature on 1-D and multidimensional orthogonal range-query problems, the special case of the polytopal partial-sum problem when the query is an orthotope. Yao's work [1], [2] on 1-D orthogonal queries laid down the foundation for later research on general partial-sums in semigroups. Chazelle and Rosenberg [3] extended Yao's 1D algorithm to multidimensional grids

[1]Actually, it's a simple LCA (Lowest Common Ancestor) problem for complete binary tree, which can be done by using bit-tricks in $O(1)$ time

using the method of ***dimension reduction***. The key idea behind the dimension reduction method for a d-dimensional orthogonal partial-sums query is as follows. During preprocessing of a d-dimensional grid, pick one grid dimension $i \in [1,d]$, and decompose the grid into a 1-D sequence of $(d-1)$-dimensional grids. Apply Yao's 1-D algorithm on the 1-D sequence by treating each $(d-1)$-dimensional grid as a semigroup element and applying $\oplus$ on them component-wise. The same procedure is then applied recursively on each $(d-1)$-dimensional grid. During the query stage, a query range is decomposed into a small number of slices by considering the problem as a 1-D grid of $(d-1)$-dimensional grids. This process produces a set of $(d-1)$-dimensional subproblems, each of which can be solved recursively. For the base case, the 1-D case can be solved using a "two-path" recursion — one solving "small" queries (intrablock queries), and the other solving "large" queries (interblock queries). In [4], they further proved that the asymptotic bound of Yao's 1-D algorithm is optimal.

To the best of our knowledge, all previous research assumes orthogonal query ranges. During the development of the Pochoir stencil compiler [5], [6], however, we encountered the problem of querying octagonally shaped ranges in a 2-D grid. Our study of that problem led us to this research.

In summary, our contributions are as follows:

- We extend the concept of range queries in a multidimensional grid from orthotopes to polytopes, the faces of which have normals known during the preprocessing stage.
- We introduce the notion of a ***meta-algorithm*** to handle range queries over a multi-dimensional grid, including orthogonal range queries in arbitrary d-dimensional grids and polygonal range queries in 2-dimensionals. Compared with dimension reduction method, the ***meta-algorithm*** treats the input algorithm as a black box, doesn't need to know any internal data structures created or used by the input algorithm, which has the advantage that any initial algorithm that just does the work can be plugged into the ***meta-algorithm*** and possibly hit the optimal asymptotic bound by undergoing several rounds of self-application. The key idea of ***meta-algorithm*** is established on the observation that all algorithms operate on data, by simply manipulating or compressing the input data to the algorithm, we can achieve the effect of reducing the asymptotic bound without any apriori knowledge of the input algorithm.
- We show that the sum of all the grid points within a given k-sided polygon in a 2-dimensional grid with N grid points can be computed in the same asymptotic bound as orthogonal range.
- We implemented the meta-algorithm for 1-D and 2-D grids. To the best of our knowledge, this is the first experimental study and performance results of static partial sum algorithm with α bound in

general semi-group, where $\alpha(n)$ stands for the functional inverse of Ackermann's function.

The main constraint of our current approach is that the normals of all sides of the query polygon must be given beforehand. Whether polygonal partial-sum queries can be efficiently answered when not all slopes are known in advance is left as an open problem. As for how to extend the methodology of processing 2-D polygonal range queries to even higher-dimensional polytopal range queries is another open problem.

The rest of the paper is organized as follows. Section II elaborates on the notion of meta-algorithm and illustrates its use to answer orthogonal range queries in d-dimensional grids. Section III explains how to answer polytopal queries using the meta-algorithm. Section IV presents some experimental results for 1D, 2D, and 3D grids.

II. Introduction to meta-algorithm

A. Introduction to meta-algorithms and its application to 2-D orthogonal range queries

In [1], [2], A. Yao introduced an algorithm to preprocess a 1-D grid recursively in $O(n\alpha(n))$ space and time bound and answer later online range queries in time $O(\alpha(n))$. A good graphical illustration of how the algorithm works can be found in [7]. Yao's algorithm [1], [7] can be viewed as 1-D meta-algorithm as defined below. We call an algorithm a ***meta-algorithm*** if it takes an algorithm (called an ***initial algorithm***) as input, applies it as a black box on some manipulated version of the input data, and effectively behaves like a new algorithm with different (ideally improved) asymptotic performance bounds. The idea similar to that of a higher-order function in a functional language.

In this section, we demonstrate how to use the notion of meta-algorithms to process 2-D orthogonal range queries. Extension to higher-dimensional grids is similar and omitted from this extended abstract due to space limitations. In Section III we will show how to extend it to support non-orthogonal range queries, such as triangular and polygonal queries in 2-D grids.

1) Initial algorithm for 2D grids: First we need an initial algorithm to kick off. The initial algorithm does not need to be very efficient as long as it solve the problem correctly. Our meta-algorithm only needs to know the complexity bound of the initial input algorithm, but no knowledge of any internal data structure or the internal workings of the algorithm is needed. One such algorithm is given in Figure 5 of Section A1. The preprocessing time and space of this initial 2-D algorithm $\Theta(N\log^2 N)$ for a grid of size N, and the query overhead is only 3 semigroup operations.

2) Meta-algorithm for 2D grids: The meta-algorithm takes one initial algorithm as input, compresses the data alternatively along dimension x or y to eventually hit the alpha bound. The description follows the notation in Figure 4(b) and the pseudo-code in Figure 6. (Figure 4(b) is also a graphical illustration of how the algorithm works)

1) We start data reduction on a longer dimension, w.l.o.g., suppose it's dimension x. For each line of dimension y, partition it along dimension x into segments of size *seg_size*, which is $f_1(n_1)$ assuming that the complexity of the input algorithm (*input_2D_algo*) is $\Theta(n_1 n_2 f_1(n_1) f_2(n_2))$.
2) Apply the $\oplus$ operator to reduce all data within each segment into a single value. All such values construct a new grid — *promoted_grid*. Apply *input_2D_algo* on *promoted_grid*.
3) For each vertical line (along dimension y), reduce the the data relative to the left and right end of each segment (of size *seg_size*) to construct *prefix_grid* and *suffix_grid*.
4) For each vertical line (e.g. line $\overline{l_1 l_1'}$ and line $\overline{l_2 l_2'}$ in Figure 4(b)) in *prefix_grid* and *suffix_grid*, apply the *input_1D_algo* on it. If the *input_2D_algo* has bound $\Theta(n_1 n_2 f_1(n_1) f_2(n_2))$, the corresponding *input_1D_algo* should have a bound of $\Theta(n f(n))$, where $f(n) \leq f_1(n)$ and $f(n) \leq f_2(n)$. Otherwise though the meta-algorithm will still be correct it won't have the desired performance bounds.
5) Recursively apply the meta-algorithm on segment blocks of size $seg\_size \times n_2$, e.g., on the colored segment block $\square efgh$ in Figure 4(b).
6) Apply the meta-algorithm described above alternatively on dimensions y and x.

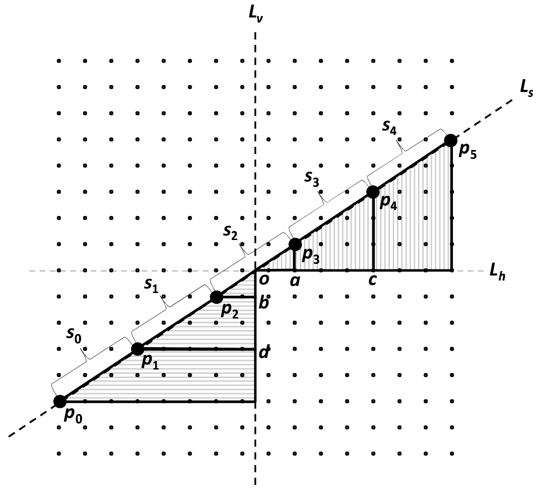
Theorem 1: Given an input $2D$ algrithm with a preprocessing complexity (in both space and time) of $\Theta(n_1 n_2 f(n_1) f(n_2))$, where $f(n) \leq n-2$, one round of application of the meta-algorithm reduces the bound to $\Theta(n_1 n_2 f^*(n_1) f^*(n_2))$.

Corollary 2: Given an input $2D$ algrithm with a preprocessing complexity (in space and time) of $\Theta(n_1 n_2 f(n_1) f(n_2))$, where $f(n) \leq n-2$, and a constant query overhead, $O(\alpha(n_1)\alpha(n_2)$ rounds of application of the meta-algorithm reduces the preprocessing bound to $\Theta(n_1 n_2 \alpha(n_1)\alpha(n_2))$ at the cost of increasing the the query overhead to $\Theta(\alpha(n_1)\alpha(n_2))$.

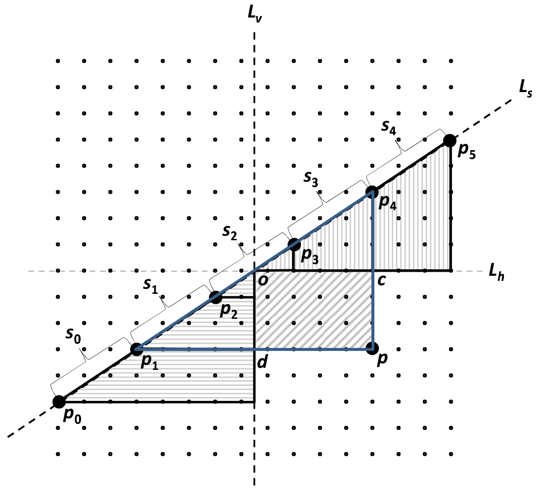
More technical details, including pseudo-codes, proofs of theorems and corollaries, can be found in Section A.

III. Polygonal Range Queries

In this section, we explain how use meta-algorithms to handle polygonal range query in 2-D grids efficiently, given all normals are known at preprocessing time. The examples in this section are for 2-D,



(a) Preparing for a 1D oblique query that returns partial-sums of trapezoidal regions



(b) Decomposing a right triangular query into oblique and rectangular queries

Fig. 1: Handling right triangular queries with horizontal bases and hypotenuses with a fixed slope.

but we believe that the methodology can be extended to higher-dimensions (will be our future work).

A. Triangular Query in 2-D grid

We will first consider right triangular queries with fixed slopes. We assume that the slope of the hypotenuse as well as that of the base (or perpendicular) are given during preprocessing time.

1) Right Triangular Queries with Horizontal Bases: In this section we show how to decompose a right triangular query with a horizontal base and a hypotenuse with a fixed slope into three queries: one 1D ***oblique query*** (to be explained shortly), one 2D rectangular query, and one 1D vertical range query. We already know how to preprocess the 2D grid to answer the last two queries using a constant number of semigroup operations. We introduce the notion of oblique queries below, and explain how to answer them in $O(1)$ semigroup operations, too.

We define oblique queries in a 2D grid G w.r.t. a given straight line and a given slope m. W.l.o.g. our explanations below assume a vertical line, say, L_v. We consider all lines with slope m that pass thru at least two grid points, and intersect L_v at a point inside G. Let L_s be such a line (see Figure 1(a)). If L_s passes thru k grid points (say, p_0, p_1, $\ldots$, p_{k-1}), we divide L_s into $k-1$ segments, where segment $i \in [0, k-2]$ is the section of the line starting from point p_i and ending right before point p_{i+1}. We will compute a semigroup sum s_i (explained below) for each segment i, and construct a range query data structure based on these s_i values to allow 1D partial-sum queries on them. The s_i values are computed as follows. Suppose L_s intersects L_v at o. Consider a horizontal line L_h through o. Now s_i is computed based on the location of p_i and p_{i+1} w.r.t. L_h :

- If p_i is on the right of L_v, then s_i is the sum of all semigroup elements inside the trapezoid containing all grid points on or below line p_ip_{i+1}, on or to the right of the vertical line through p_i, to the left of the vertical line through p_{i+1}, and on or above the horizontal line L_h. For example, in Figure 1(a), s_3 is the sum of all grid points on or inside the trapezoid defined by lines p_3p_4, p_3a, p_4c and ac, except the points on line p_4c.
- If p_{i+1} is on the left of L_v, then s_i is the sum of all semigroup elements inside the trapezoid containing all grid points on or to the right of p_ip_{i+1}, on or above the horizontal line through p_i, below the horizontal line through p_{i+1}, and on or to the left of L_v. For example, in Figure 1(a), s_1 is the sum of all grid points on or inside the trapezoid defined by lines p_1p_2, p_1d, p_2b and bd, but excluding the points on line p_2b.
- If p_i is on the left, but p_{i+1} is on the right of L_v, then s_i is the sum of grid points on or inside the right triangle defined by points p_i, o and line L_v, and the one defined by points p_{i+1}, o and line L_h, excluding the points on the vertical line through p_{i+1}. Sum s_2 in Figure 1(a) is an example.

Now suppose we are given a right triangular query p_ip_jp whose hypotenuse p_ip_j lies on L_s, but p_i and p_j are on the opposite sides of L_v. Then p_ip_jp can be decomposed into the following three queries (see, e.g., triangle p_1p_4p in Figure 1(b)).

- An oblique query on L_s from p_i to p_j,
- A 2D query for the rectangular region $ocpd$, where c is the intersection point of p_jp and L_h, and d is the point of intersection of lines p_ip and L_v, and
- A 1D vertical range query from point p_j to point c.

Since each of these queries has $O(1)$ complexity, query complexity of p_ip_jp is also $O(1)$.

An oblique query data structure for the entire grid G can be constructed as follows. First we vertically or horizontally partition G into two halves, and contruct a 1D oblique range query data structure for each line L_s of given slope m that passes thru at least two grid points and interesects the partition line inside the grid. We then preprocess the two halves recursively.

In order to answer a right triangular query we first check if the triangle is completely contained inside one of the two partitions of G. If not, it intersects the partitioning line, and we can answer the query as explained above. Otherwise, we move into the half that contains the triangle, and recursively find a partition in which the triangle intersects the partitioning line, and answer the query accordingly.

We will now analyze the preprocessing time and space required for constructing the right triangular range query data structure. Our analysis of preprocessing time will depend on the complexity of computing the sum of the grid points inside the smallest right triangle of slope m (where m is the slope of the hypotenuse) that can be specified on the given grid G. Observe that the two endpoints of the hypotenuse of such a triangle will correspond to two consecutive grid points on an oblique line L_s. Let Δ be the complexity of computing such a sum.

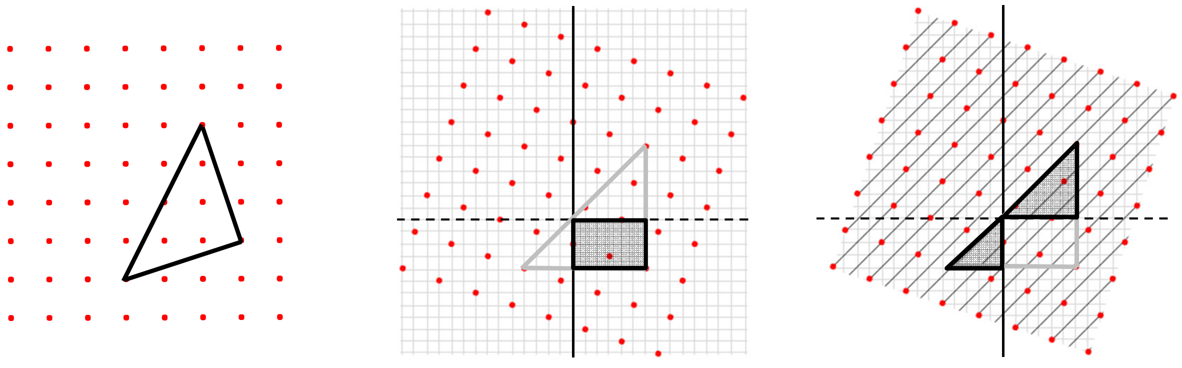
Theorem 3: The right triangular range query data structure for a 2D grid of size N can be constructed in $\Theta(N(\Delta+\log^2 N))$ time and $\Theta(N\log^2 N)$ space assuming that the hypotenuse of each query triangle has a fixed slope and the base is horizontal.

Proof: (sketch) We first construct a range query data structure for 2D rectangular queries on the given grid G. As we discussed earlier such a data structure can be constructed in $\Theta(N\log^2 N)$ time and space so that queries can be answered in $O(1)$ semigroup operations.

Let us now consider the space complexity of the data structure for oblique queries. We know that an oblique line passing through k grid points stores $k-1$ semigroup elements. Ignoring for the time being the complexity of computing these elements, we can preprocess such a line in $\Theta(k\log k)$ time and space so that 1D range queries on those elements can be answered in $O(1)$ time. If we are preprocessing a 2D grid G' of size N' at a particular level of recursion, then the preprocessing time and space for all oblique lines for G' in that level of recursion is $\Theta\left(\sum_{0\le i<r} k_i \log k_i\right)$, where r is the number of oblique lines and k_i is the number of grid points on the i-th oblique line. Observe that $\sum_{0\le i<r} k_i = N'$, and so the complexity above reduces to $\Theta(N'\log N')$. Now since G has N points, and it is preprocessed through $\log N$ levels of recursion, the preprocessing time and space for G is clearly $\Theta(N\log^2 N)$.

The total time required for computing the semigroup elements (i.e., the s_i values, see Figure 1(a)) can be derived as follows. For each oblique line L_s, we first compute the sum of all grid points inside each smallest triangle with hypotenuse on L_s. Since there are fewer than N such triangles across all oblique lines, the total time for precomputing all of them is $O(N\Delta)$. Now observe that the trapezoid below each s_i (see Figure 1(a)) can decomposed into one smallest triangle and a rectangle. The sum of the grid points inside the rectangle can be computed in constant time using the rectangular query data structure we already constructed. Hence, each s_i can thus be computed in $O(1)$ time. ■

Observe that if B and P are the lengths of the base and the perpendicular, respectively, of a smallest triangle, then $\Delta = O(min(B,P))$, as we can use our rectangular query data structure to perform either P horizontal 1D queries or B vertical queries to compute the sum inside the triangle. In the worst case,



(a) A right triangle with an oblique base

(b) Rotating the input grid to make the triangle base horizontal, and embedding it in a finer grid with axis-parallel sides

(c) Oblique queries in the fine grid

Fig. 2: Handling right triangular queries with a fixed hypotenuse slope but without horizontal bases.

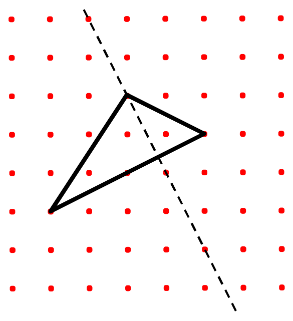
$\Delta = O(min(B,P)) = O(\sqrt{N})$.

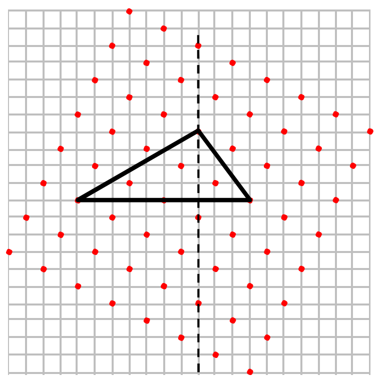
Meta algorithm: The meta-algorithm for right triangular query that rectilinear to axes is very similar to that for rectangular query. So we just use a triangular preprocessing/query algorithm as the *input_2D_algo*, then everything remains the same as in meta-algorithm for rectangular ranges. Section II-A2 explains more on how meta-algorithm works on 2D rectangles. Figure 4(b) and Figure D provides a graphical illustration of how the meta-algorithm works on rectangles and right triangles. More details are omitted from this extended abstract.

Theorem 4: If we input the above initial right triangular range preprocessing and query algorithm to our meta-algorithm and self-apply $\Theta(\alpha^2(N))$, times, it will achieve $\Theta(N(\Delta+\alpha^2(N)))$ time and $\Theta(N\alpha^2(N))$ space assuming that the hypotenuse of each query triangle has a fixed slope and the base is horizontal. The query overhead will become $\Theta(\alpha^2(N))$.

2) Right Triangular Queries without Horizontal Bases: In this section we consider right triangular queries that have fixed slopes for all sides, but the base is neither horizontal nor vertical. We define the ***feature size*** of a straight line passing thru a grid as the distance between two consecutive grid points lying on that line. Let δ_b, δ_p and δ_h be the feature sizes of base, perpendicular and hypotenuse, respectively, of any query triangle. For example, in Figure 2(a), $\delta_b = \delta_p = \sqrt{10}$ and $\delta_h = \sqrt{5}$. We assume that δ_b and δ_p are small constants.

Assuming that the original grid has N points, we can rotate the grid so that query base becomes horizontal, and embed it in an axis-parallel finer grid of size $\Theta(N\delta_b\delta_p)$ so that every point of the original


(a) An arbitrary triangle in a 2D grid


(b) Rotating the grid to make the largest side of the triangle horizontal, and embedding the input grid in a finer grid with axis-parallel sides

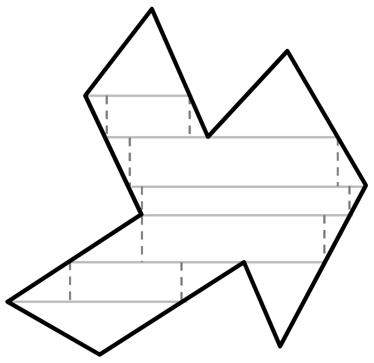

(c) Decomposing an arbitrary polygonal query into rectangles and triangles using only horizontal and vertical lines

Fig. 3: Handling arbitrary triangular and polygonal queries with fixed slopes.

grid sits on a grid point of the finer grid (see Figure 2(b)). Since δ_b and δ_p are assumed to be constants, we can use our algorithm from Section III-A1 to solve right triangular range queries on this grid within the performance bounds proved in Theorem 3.

3) Arbitrary Triangular Query in a 2-D grid: We consider arbitrary triangular queries on a 2D grid (see Figure 3(a)) with fixed slopes for all sides. We take the largest side b of such a triangle, and rotate the grid so that b becomes horizontal. We embed the rotated grid in a finer grid as in Section III-A2 (see Figure 3(a)). Let us draw a perpendicular p on b from the triangle vertex that does not lie on b. Now assuming that the feature sizes of b and p are small constants the size of the finer grid will be within a constant factor of the size of the original grid. The perpendicular p divides the query triangle into two right triangles, and thus the given triangular query can decomposed into two right triangular queries, and we can preprocess the finer grid to answer them within the performance bounds proved in Theorem 3.

B. Polygonal Query in a 2-D grid

We now consider arbitrary query polygons with sides that can have slopes from a fixed set of slopes known during preprocessing time. We assume that lines corresponding to these slopes have small constant feature sizes. The key issue is decomposing this polygon into rectangles and triangles without introducing any additional slopes. First we draw horizontal lines through each vertex of the polygon which decompose the polygon into a set of disjoin triangles and trapezoids (see Figure 3(c)). Then we use vertical lines to decompose each trapezoid into at most one rectangle and no more than two right triangles. The total number of triangles and rectangles thus created is within a constant factor of the total number vertices in the original polygon. We now create a query data structure for each triangle and rectangles. If the polygon has only a constant number of vertices the performance bounds given in Theorem 3 hold.

IV. Experimental Results

We have implemented our initial algorithm and meta-algorithm (pseudo-codes in appendix Section A1 Section C) for the orthogonal range partial-sum problem on commutative semigroups. There are empirical studies of the Range Minimum Query (RMQ) problem [8], which is a special case of the partial-sum problem. But to the best of our knowledge, ours is the first experimental study of the static partial-sum problem using an algorithm with an α bound. We implemented the initial algorithm, which has a complexity of $O(N \log^d N)$, and compared its performance with a couple of algorithms generated by our meta-algorithm, that is, an $O(N(\log^* N)^d)$ algorithm and an $O(N\alpha^d(N))$ algorithm.

We implemented all algorithms in C++ standard C++11 [9]. The use "unsigned long" as the datatype, and the partial sum operation was simply the general integer "+" operation wrapped in a lambda function and passed as a function object to all algorithms. All the query time is calculated as an arithmetic mean of 1000 random queries over the entire d-dimensional grid.

We ran our experiments on an Intel Core i7 (Nehalem) machine. [2] In all performance graphs, the vertical axis is "problem size" in "unsigned long", which is 64-bits long, and the horizontal axis is "measured time" in "milliseconds". So in all performance graphs, the closer a curve to the horizontal axis, the better the performance of the corresponding algorithm. To verify the functional correctness of the meta-algorithm and to compare the query overhead, we implemented a naïve scan through algorithm which scans through all the points in the query range and sums ($\oplus$) them up to answer any online query.

From all graphs in Figure 8, we see that the preprocessing time of the α bound algorithm is always better than the $\log^*$ bound algorithm, which, in turn, is better than the initial algorithm of log bound. So our experimental results match theoretical predictions, which confirms the efficiency of the meta-algorithm.

From all graphs in Figure 9, we see that compared with naïve scan algorithm, the query overhead of our initial algorithm or those transformed by meta-algorithm is negligible.

[2]Intel Xeon X5650, with clock frequency 2.66 GHz, 32KB L1 data cache/core, 256KB data cache/core, 12MB L3 cache/socket. Compiler is icc 12.1.0.

In the final version, we will include more performance data on triangular and polygonal preprocessing and query in 2-D grids.

References

[1] A. C. Yao, "Space-time tradeoff for answering range queries (extended abstract)," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, ser. STOC '82. New York, NY, USA: ACM, 1982, pp. 128–136. [Online]. Available: http://doi.acm.org/10.1145/800070.802185

[2] A. C.-C. Yao, "On the complexity of maintaining partial sums," *SIAM J. Comput.*, vol. 14, no. 2, pp. 277–288, 1985.

[3] B. Chazelle and B. Rosenberg, "Computing partial sums in multidimensional arrays," in *Proceedings of the fifth annual symposium on Computational geometry*, ser. SCG '89. New York, NY, USA: ACM, 1989, pp. 131–139. [Online]. Available: http://doi.acm.org/10.1145/73833.73848

[4] ——, "The complexity of computing partial sums off-line," *Int. J. Comput. Geometry Appl.*, vol. 1, no. 1, pp. 33–45, 1991.

[5] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[6] Y. Tang, R. A. Chowdhury, C.-K. Luk, and C. E. Leiserson, "Coding stencil computation using the pochoir stencil-specification language," in *3rd USENIX Workshop on Hot Topics in Parallelism*, ser. HotPar'11, 2011.

[7] R. Seidel, "Understanding the inverse ackermann function," in *Invited Talk : 22nd European Workshop on Computational Geometry*, Delphi, Greece, 2006. [Online]. Available: cgi.di.uoa.gr/~ewcg06/invited/Seidel.pdf

[8] J. Fischer and V. Heun, "Theoretical and practical improvements on the rmq-problem, with applications to lca and lce," in *CPM*, 2006, pp. 36–48.

[9] http://en.wikipedia.org/wiki/C%2B%2B11#cite_note-0.

Appendix

A. Meta-algorithm for 2-D grids

From the results in [1], [2], [7], we can establish following recurrence for space and time bound of 1-D preprocessing algorithm.

$$\mathcal{S}_{1,0}(n) = n\log n // \text{ initial algo} \tag{2}$$

$$\mathcal{S}_{1,1}(n) = \Theta(n) + \mathcal{S}_{1,0}(n/\log n) + n/\log n \cdot \mathcal{S}_{1,1}(\log n) // \text{ use } \mathcal{S}_{1,0} \text{ inter-block query} \tag{3}$$

$$\ldots = \ldots \tag{4}$$

$$\mathcal{S}_{1,k}(n) = \Theta(n) + \mathcal{S}_{1,k-1}(n/f(n)) + n/f(n) \cdot \mathcal{S}_{1,k}(f(n)) // \text{ Suppose the complexity of } \mathcal{S}_{1,k-1}(n) = nf(n) \tag{5}$$

$$\mathcal{S}_{1,\alpha(n)}(n) = \Theta(n\alpha(n)) \tag{6}$$

$$\tag{7}$$

For the notation $\mathcal{S}_{d,k}$, $\mathcal{S}$ stands for the space and time bound, the first subscript stands for dimensionality, the second is the recursion level. Similarly, we have a recurrence for query overhead, where $\mathcal{Q}$ stands for the query overhead, the semantics of subscript is the same as in $\mathcal{S}_{d,k}$:

$$Q_{1,0} = 1 \text{// initial algo} \tag{8}$$

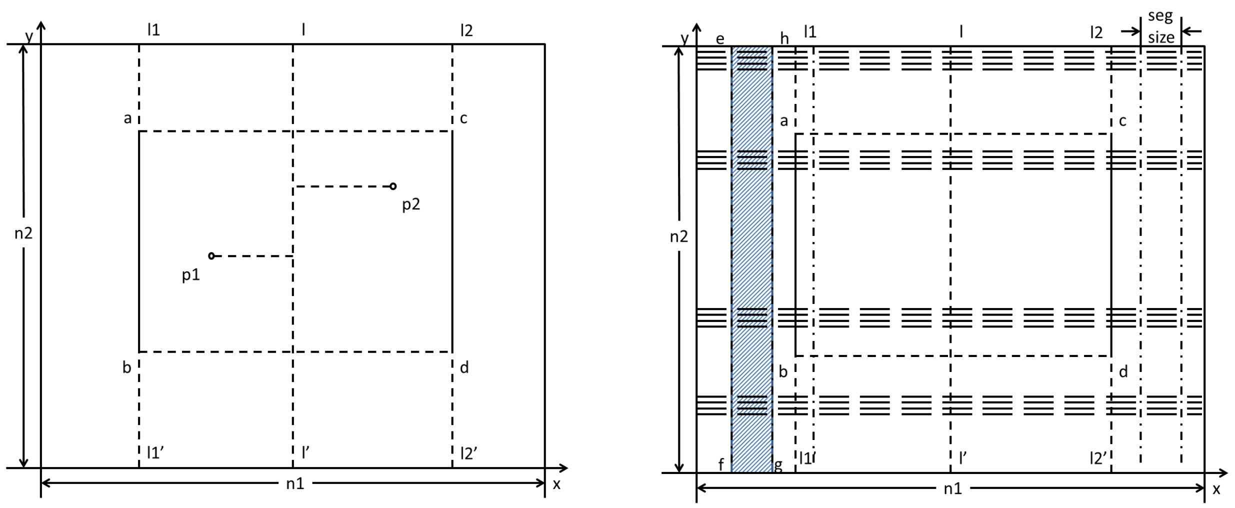
$$Q_{1,1} = 2 + Q_{1,0} \text{// any query can be decomposed into at most one inter-block query and one suffix/prefix} \tag{9}$$

$$\ldots = \ldots \tag{10}$$

$$Q_{1,k} = 2 + Q_{1,k-1} \tag{11}$$

$$Q_{1,\alpha(n)} = \Theta(\alpha(n)) \tag{12}$$

To extend the same kind of recurrence to multi-dimensional grids, we start from how to use meta-algorithm to process a 2-D grid and iteratively apply to itself to achieve alpha bound. First, we need an initial algorithm to kick off. The initial algorithm doesn't need to be very smart, it just needs to work. Our meta-algorithm will later transform it through a series of recursion levels and improves the asymptotic bound to eventually hit the alpha bound.



(a) Initial algorithm for 2D orthogonal range queries

(b) Meta-algorithm for 2D orthogonal range queries

Fig. 4: Initial and meta-algorithm for 2-D orthogonal range queries

1) Initial algorithm for 2*D grids:* As shown in Figure 4(a), the initial algorithm is a direct extension of A. Yao's initial algorithm [1] to 2-D grids. The algorithm is diagrammed in Figure 4(a). The description is as follows and uses the notation in Figure 4(a), the pseudo-code is in Figure 5

1) Select a longer dimension, without loss of generality, let's suppose it's dimension x.
2) Partition dimension x by a center line $\overline{ll'}$ into evenly two parts. Then all points in the grid resides either on the left (e.g. p_1) of the center line or right (e.g. p_2). For all the points, reducing (by

applying the $\oplus$ operator) the data from center line to itself by dynamic programming. We call the reduced data prefix or suffix value from the center line.

3) After data reduction on both prefix and suffix values, applying Yao's initial 1-D preprocessing algorithm [1], [4], which has complexity $O(n\log(n))$ on all lines along vertical dimension y.
4) Now we have the observation that all 2-D orthogonal range query that spans across the center line $\overline{ll'}$ can be answered by conducting two 1-D queries on both left-hand side and right-hand side of the center line. E.g. To query the rectangle $\square abdc$ in Figure 4(a), we perform one 1-D query on line $\overline{l_1l_1'}$ and another 1-D query on line $\overline{l_2l_2'}$. Combining these two results by $\oplus$ operator, we get the final query result of the orthogonal range query.
5) Recursively applying the above procedure to the left and right sub-grids of line $\overline{ll'}$.

Theorem 5: The preprocessing of Algoritm Figure 4(a) has complexity of $\mathcal{S}_{2,0}(n_1,n_2) = \Theta(n_1 \cdot n_2 \cdot \log(n_1)\cdot\log(n_2))$ in both time and space.

Proof: Apparently, at each level of recursion, it requires at least n_1n_2 space and time to hold prefix and suffix grids.

For each vertical lines, such as $\overline{l_1l_1'}$ and $\overline{l_2l_2'}$, we apply Yao's initial 1-D algorithm on it, and denote it as $\mathcal{S}_{1,0}(n) = \Theta(n\log n)$.

Since the partition occurs on dimension x, which will terminate when the number of vertical lines in the grid less than or equal to 1, there are in total $\log n_1$ levels.

Combining all above calculations we have the recurrence:

$$\begin{aligned}\mathcal{S}_{2,0}(n_1,n_2) &= n_1n_2 + \log n_1\{n_1\cdot\mathcal{S}_{1,0}(n_2)\} && (13)\\ &= \Theta(\log n_1 n_1 n_2 \log n_2) && (14)\end{aligned}$$

■

Corollary 6: The query overhead of Algorithm Figure 4(a) is $\mathcal{Q}_{2,0} = 3-\oplus$

Proof: For the query, we first need to locate which recursion level the query range resides, which is equivalent to finding the lowest common ancestor of its range on dimension x, the overhead of which is not counted in our simple arithmetic model. [3]

After locating the lowest common ancestor of the query range on dimension x, performing two 1-D queries on left and right-hand side of the center line of the level, each of which requires $\mathcal{Q}_{1,0} = 1-\oplus$

[3]In RAM model, by using bit-tricks, finding lowest common ancestor in a complete binary tree can be accomplished in $O(1)$ time

overhead. In the end, we need to combine both results from two 1-D queries into one, which requires another $1-\oplus$ operation. So in total, we have $Q_{2,0} = 2Q_{1,0}+1 = 3$. ∎

2) Meta-algorithm for 2D grids: The meta-algorithm takes an algorithm as input, such as the initial algorithm described in Figure 4(a) (Figure 5), compress the data alternatively on dimension x or y to eventually hit the α bound. The description follows the notation in Figure 4(b), the pseudo-code is in Figure 6

1) We start data reduction on a longer dimension, without loss of generality, suppose it's dimension x. For each line of dimension y, partition it along dimension x into segments of size $\log(x_1 - x_0)$, or $\log^*(x_1 - x_0)$, $\log^{**}(x_1 - x_0)$, ... (*seg_size* in Figure 6). The length of *seg_size* depends on the complexity of *input_2D_algo*, which can be a user's input parameter.
2) For each segment, apply $\oplus$ operator over it and reduce all data within the segment into one single value. All such values construct a new grid — *promoted_grid*. Applying the *input_2D_algo* to the *promoted_grid*.
3) For each vertical line (along dimension y), reduce the the data relative to the left and right end of each segment (of size *seg_size* to construct the *prefix_grid* and *suffix_grid*.
4) For each vertical line (e.g. line $\overline{l_1 l_1'}$ and line $\overline{l_2 l_2'}$ in Figure 4(b)) in *prefix_grid* and *suffix_grid*, apply the *input_1D_algo* to it. If the *input_2D_algo* has bound $\Theta(n_1 n_2 f_1(n_1) f_2(n_2))$, the corresponding *input_1D_algo* should be of bound $\Theta(nf(n))$, where $f(n) \le f_1(n)$ and $f(n) \le f_2(n)$.
5) Recursively apply the meta-algorithm itself on segment blocks of size *seg_size* $\times n_2$. E.g. on colored segment block $\Box efgh$ in Figure 4(b)
6) Repeat the above meta-algorithm alternatively on dimension y, x.

Theorem 7: The preprocessing of Algoritm Figure 4(b) has complexity of $\mathcal{S}_{2,\alpha(n_1)\alpha(n_2)} = \Theta(n_1 \cdot n_2 \cdot \alpha(n_1) \cdot \alpha(n_2))$ in both time and space.

Proof:

1) In Theorem 5, we proved that $\mathcal{S}_{2,0} = 3$.
2) For recursion level 1: we have $\mathcal{S}_{2,0}$ as *input_2D_algo*, and $\mathcal{S}_{1,0}$ as *input_1D_algo*. We apply *input_2D_algo* to *promoted_grid*, which is of size $n_1/\log n_1 \times n_2$, and apply *input_1D_algo* to *prefix_grid* and *suffix_grid*, each of which is of size $n_1 \times n_2$. So we have the recurrence:

$$\begin{aligned} \mathcal{S}_{2,1}(n_1,n_2) &= \mathcal{S}_{2,0}(n_1/\log n_1, n_2) + 2 \times n_1 \times \mathcal{S}_{1,0}(n_2) \frac{n_1}{\log n_1} \cdot \mathcal{S}_{2,1}(\log n_1, n_2) && (15) \\ &= \Theta(n_1 n_2 \log n_2 \log^* n_1) && (16) \end{aligned}$$

3) Solution in Equation (16) means we now have a 2-D preprocessing algorithm of complexity $O(n_1 n_2 \log^* n_1 \log n_2)$ in both space and time. Repeat the same procedure on dimension y, we will have the recurrence:

$$\begin{aligned} \mathcal{S}_{2,2}(n_1,n_2) &= \mathcal{S}_{2,1}(n_1,n_2/\log n_2) + 2\times n_2 \times \mathcal{S}_{1,1}(n_1) + \frac{n_2}{\log n_2}\cdot \mathcal{S}_{2,2}(n_1,\log n_2) && (17)\\ &= \Theta(n_1 n_2 \log^* n_1 \log^* n_2) && (18)\end{aligned}$$

4) Repeat above two procedures, more generally, if we assume *input_2D_algo* has complexity $\Theta(n_1 n_2 f(n_1) f(n_2))$, where $f(n) \le n-2$. We first segment the 2-D grid along x dimension into size of $f(n_1)$ and supply a 1-D algorithm of complexity $\Theta(nf(n))$, we will have the recurrence:

$$\begin{aligned} \mathcal{S}_{2,k}(n_1,n_2) &= \mathcal{S}_{2,k-1}(n_1/f(n_1),n_2) + 2\times n_1 \times \mathcal{S}_{1,k-1}(n_2) + \frac{n_1}{f(n_1)}\cdot \mathcal{S}_{2,k}(f(n_1),n_2) && (19)\\ &= \Theta(n_1 n_2 f(n_2) f^*(n_1)) && (20)\end{aligned}$$

In a second step, we supply the $\mathcal{S}_{2,k}$ as *input_2D_algo* and a 1-D algorithm of complexity $\mathcal{S}_{1,k} = nf^*(n)$ as *input_1D_algo*.

$$\begin{aligned} \mathcal{S}_{2,k+1}(n_1,n_2) &= \mathcal{S}_{2,k}(n_1,n_2/f(n_2)) + 2\times n_2 \times \mathcal{S}_{1,k}(n_1) + \frac{n_2}{f(n_2)}\cdot \mathcal{S}_{2,k+1}(n_1,f(n_2)) && (21)\\ &= \Theta(n_1 n_2 f^*(n_1) f^*(n_2)) && (22)\end{aligned}$$

5) We define the alpha function, i.e. the inverse Ackermann function to be: $\alpha(n) = min\{k | \log^{\overbrace{**\cdots*}^{ktimes}}(n) \le 2\}$ [7]

6) Combining above steps completes the induction.

■

Corollary 8: Without loss of generality, if we assume $n_1 = n_2 = n = \sqrt(N)$, where N is the total number of points in entire grid. The query overhead of Algorithm Figure 4(b) is $\mathcal{Q}_{2,k+1} = \Theta(\alpha^2(n))$

Proof: For the query, we just need to locate which recursion level the query range resides. At each level, the query will be decomposed into at most three sub-queries, one 2-D query on the *promoted_grid*, two 1-D queries on the *prefix/suffix_grid*. If we assume that the index calculation to locate the recursion level is free, we have following recurrence for the query overhead:

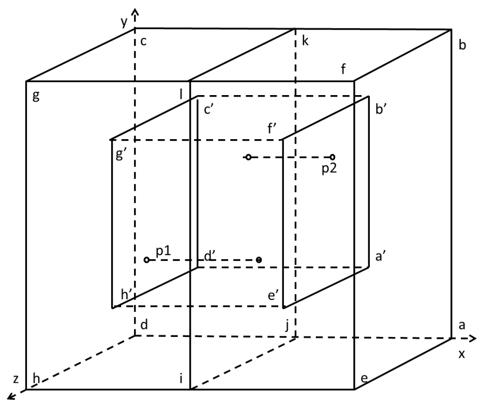
$$\begin{aligned} \mathcal{Q}(2,0) &= 2\times \mathcal{Q}_{1,0} + 1 = 3 \text{//initial algorithm} && (23)\\ \mathcal{Q}(2,1) &= \mathcal{Q}_{2,0} + 2\times \mathcal{Q}_{1,0} + 2 \text{//Decompose the query into one 2-D query and two 1-D queries} && (24)\\ \mathcal{Q}_{2,2} &= \mathcal{Q}_{2,1} + 2\times \mathcal{Q}_{1,1} + 2 && (25)\end{aligned}$$

$$\ldots \quad = \quad \ldots \tag{26}$$

$$Q_{2,k} \quad = \quad Q_{2,k-1} + 2 * Q_{1,k-1} + 2 \quad \text{//k is odd} \tag{27}$$

$$Q_{2,k+1} \quad = \quad Q_{2,k} + 2 * Q_{1,k} + 2 \tag{28}$$

Solve the recurrence in Equation (28), we have $Q_{2,k+1} = 2\Sigma_{i=0}^{k} Q_{1,i} + 2k$, where $Q_{1,k} = 2k+1$ is the query overhead of 1-D algorithm [7]. So $Q_{2,k+1} = 2k(k+1) + 2k = \Theta(k^2)$ is the query overhead of the 2-D meta-algorithm. Since the 1-D algorithm's query overhead is $\Theta(2\alpha(n)+1)$ [1], [7], so the 2-D meta-algorithm is $\Theta(\alpha^2(n))$ ■



3) Initial algorithm for 3D grids: The 3-D initial algorithm for orthogonal range query is pretty straightforward and diagramed in Figure A3. Instead of center line, we now have a center plane. In Figure A3, assuming x is the longest dimension and the first to reduce, we have a center face $\square ijkl$ to partition the entire grid into two parts. All the points on the left-hand/right-hand side will reduce from center plane to itself and store it as *prefix*/*suffix_grid*. All the query range spanning across the center plane can now be answered by two 2-D queries. Then recurse on both left and right parts of the center plane will cover all 3-D query ranges.

The meta-algorithm for 3-D is also similar to the case of 2-D, we just need to reduce on all three dimensions to bring the preprocessing complexity from $O(n_1 n_2 n_3 f(n_1) f(n_2) f(n_3))$ down to $O(n_1 n_2 n_3 f^*(n_1) f^*(n_2) f^*(n_3))$. The exact procedure is omitted here.

B. Pseudo-code for the 2-D initial algorithm

C. Pseudo-code for the 2-D meta algorithm

```
INIT-2D($grid, x_0, x_1, y_0, y_1$; INPUT_1D_ALGO, OP)
 1  $prefix\_grid$ = new $((x_1 - x_0) * (y_1 - y_0))$
 2  $suffix\_grid$ = new $((x_1 - x_0) * (y_1 - y_0))$
 3  // Assuming we apply divide-and-conquer on dimension $x$
 4  if $x_1 - x_0 <= 1$
 5      return // if the size of longer dimension $<= 1$ then return
 6  else
 7      $mid\_point = (x_0 + x_1)/2$
 8      // Copy the starting point of prefix (on the right side of $mid\_point$)
 9      // and suffix (on the left side of $mid\_point$)
10      for $y = y_0$ to $y_1$
11          $suffix\_grid[mid\_point - 1][y] = grid[mid\_pont - 1][y]$
12          $prefix\_grid[mid\_point][y] = grid[mid\_point][y]$
13      // Reduce on all suffix
14      for $x = mid\_point - 2$ downto $x_0$
15          for $y = y_0$ to $y_1$
16              $suffix\_grid[x][y]$ = OP$(grid[x][y], suffix\_grid[x+1][y])$
17      // Reduce on all prefix
18      for $x = mid\_point + 1$ to $x_1$
19          for $y = y_0$ to $y_1$
20              $prefix\_grid[x][y]$ = OP$(prefix\_grid[x-1][y], grid[x][y])$
21      // Apply the input 1D algorithm on reduced prefix/suffix grid
22      parallel for $x = x_0$ to $x_1$
23          INPUT_1D_ALGO$(suffix\_grid[x][], y_0, y_1, op)$
24          INPUT_1D_ALGO$(prefix\_grid[x][], y_0, y_1, op)$
25      // Recursively call Init-2D algorithm on the left side
26      spawn INIT-2D$(grid, x_0, mid\_point\text{-}1, y_0, y_1; input\_1D\_algo, op)$
27      // Recursively call INIT-2D algorithm on the right side
28      INIT-2D$(grid, mid\_point, x_1, y_0, y_1; input\_1D\_algo, op)$
29      sync
```

Fig. 5: Initial 2-D algorithm

D. Supplemental material for meta-algorithm for non-orthogonal range queries

E. Experimental results of static partial sum problem

```
META-2D($grid$,$x_0$,$x_1$,$y_0$,$y_1$;$REC$, INPUT_2D_ALGO, INPUT_1D_ALGO, OP)
1  // Asssuming we do the data reduction on dimension $x$
2  $seg\_size = \log(x_1 - x_0)$
3  // Depending on $REC$ level, we partition the seg_size into $\log(x_1 - x_0)$, or $\log^*(x_1 - x_0)$, $\log^{**}(x_1 - x_0)$, $\ldots$
4  for $i = 0$ to $REC$
5      $seg\_size = *seg\_size$
6  $n\_segs = (x_1 - x_0)/seg\_size$
7  $promoted\_grid$ = new $(n\_segs * (y_1 - y_0))$
8  $prefix\_grid$ = new $((x_1 - x_0) * (y_1 - y_0))$
9  $suffix\_grid$ = new $((x_1 - x_0) * (y_1 - y_0))$
10 for $i = 0$ to $n\_segs$
11     for $j = 0$ to $y_1 - y_0$
12         // REDUCE apply input partial sum operator OP over range $[i * seg_size, (i+1) * seg_size][j]$
           // and return the reduced result into $promoted\_grid[i][j]$
13         $promoted\_grid[i][j]$ = REDUCE($i, j$, OP)
14         // REDUCE_PREFIX apply input partial sum operator OP over range $[i * seg\_size, (i+1) * seg\_size][j]$
           // and reduce the data relative to the beginning point of the segment $(i * seg_size)$
15         $prefix\_grid[i][j]$ = REDUCE_PREFIX($i, j$, OP)
16         // REDUCE_SUFFIX apply input partial sum operator OP over range $[i * seg\_size, (i+1) * seg\_size][j]$
           // and reduce the data relative to the end point of the segment $((i+1) * seg_size)$
17         $suffix\_grid[i][j]$ = REDUCE_SUFFIX($i, j$, OP)
18 // Apply INPUT_2D_ALGO on the $promoted\_grid$
19 INPUT_2D_ALGO($promoted_grid, 0, n_segs, y_0, y_1$;$REC\text{-}1$, INPUT_1D_ALGO, OP)
20 parallel for $i = 0$ to $n\_segs$
21     // Apply META-2D itself onto the segments of $[i * seg\_size, (i+1) * seg\_size, y_0, y_1]$
22     META-2D($grid, i * seg\_size, (i+1) * seg\_size, y_0, y_1$;$REC$, INPUT_2D_ALGO, INPUT_1D_ALGO, OP)
23 parallel for $i = x_0$ to $x_1$
24     // Apply the input 1D algorithm on reduced prefix/suffix grid
25     INPUT_1D_ALGO($prefix\_grid[i], y_0, y_1$, OP)
26     INPUT_1D_ALGO($suffix\_grid[i], y_0, y_1$, OP)
```

Fig. 6: Meta 2-D algorithm

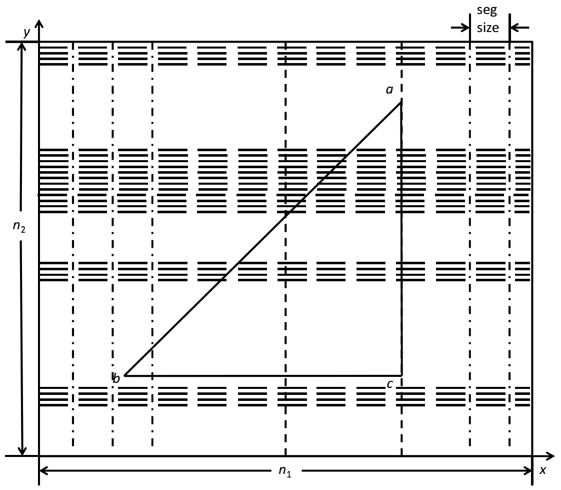
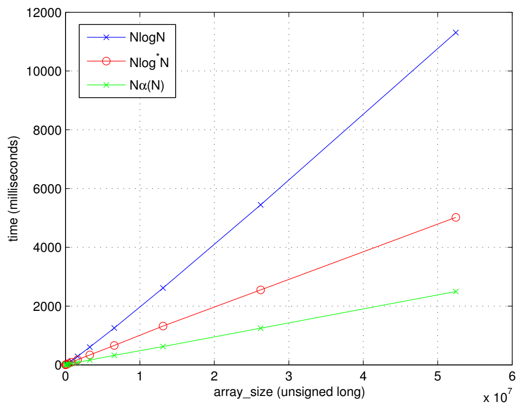
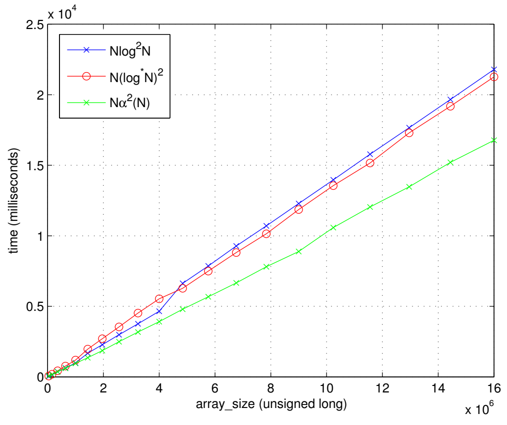


Fig. 7: Meta-algorithm for right triangular query in 2-D grid.

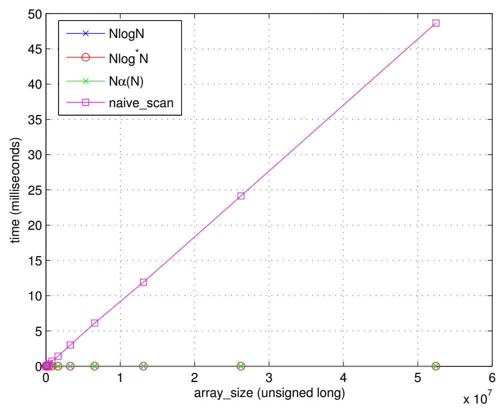


(a) Preprocessing time of meta-algorithm for 1-D grid.

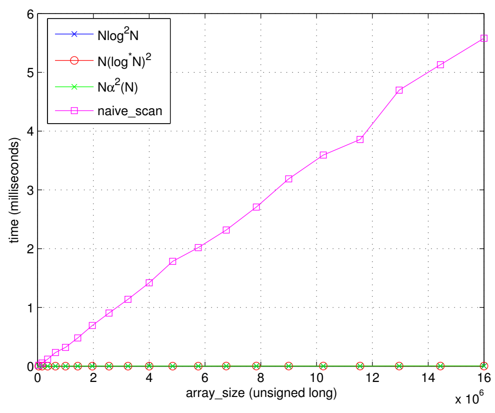


(b) Preprocessing time of meta-algorithm for 2-D grid

Fig. 8: Performance data of meta-algorithm's Preprocessing time for 1-D and 2-D grid



(a) Query time of meta-algorithm for 1-D grid



(b) Query time of meta-algorithm for 2-D grid

Fig. 9: Performance data of meta-algorithm's Query time for 1-D and 2-D grid