



Aggregating over Dominated Points by Sorting, Scanning, Zip and Flat Maps

Jacek Sroka  

University of Warsaw, Poland

Jerzy Tyszkiewicz  

University of Warsaw, Poland

Abstract

Prefix aggregation operation (also called scan), and its particular case, prefix summation, is an important parallel primitive and enjoys a lot of attention in the research literature. It is also used in many algorithms as one of the steps.

Aggregation over dominated points in \mathbb{R}^m is a multidimensional generalisation of prefix aggregation. It is also intensively researched, both as a parallel primitive and as a practical problem, encountered in computational geometry, spatial databases and data warehouses.

In this paper we show that, for a constant dimension m , aggregation over dominated points in \mathbb{R}^m can be computed by $O(1)$ basic operations that include sorting the whole dataset, zipping sorted lists of elements, computing prefix aggregations of lists of elements and flat maps, which expand the data size from initial n to $n \log^{m-1} n$.

Thereby we establish that prefix aggregation suffices to express aggregation over dominated points in more dimensions, even though the latter is a far-reaching generalisation of the former. Many problems known to be expressible by aggregation over dominated points become expressible by prefix aggregation, too.

We rely on a small set of primitive operations which guarantee an easy transfer to various distributed architectures and some desired properties of the implementation.

2012 ACM Subject Classification Theory of computation \rightarrow Massively parallel algorithms; Theory of computation \rightarrow Parallel computing models; Theory of computation \rightarrow Database query processing and optimization (theory)

Keywords and phrases Aggregation over dominated points prefix sums sorting flat map range tree parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.96

Acknowledgements We would like to thank Filip Murlak for encouragement and discussions on the topic. We also would like to thank anonymous referees for comments, which helped us improve the presentation of the paper.

1 Introduction

In this paper we derive a tight relation between two computing problems: prefix aggregation and aggregation over dominated points. We first describe the problems alone, in a framework which covers them both.

The input data is assumed to be a collection of tuples composed of sortable atomic elements. We are interested in aggregation in general. We assume the data consists of two sets: a set D of data points and a set Q of queries where for each query there is a subset $\hat{q} \subseteq D$ of data points it matches.

A nonempty set A is the domain of weights. We are also given an associative and commutative function $\oplus : A \times A \rightarrow A$ with unit e , i.e., neutral element of \oplus .



© Jacek Sroka and Jerzy Tyszkiewicz;
licensed under Creative Commons License CC-BY 4.0
31st Annual European Symposium on Algorithms (ESA 2023).

Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 96;
pp. 96:1–96:13



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Data points have weights in A , defined by a function $w : D \rightarrow A$. We use notation $\bigoplus_{d \in S} w(d)$ and the like for the application of \bigoplus to a multiset of weights of elements $S \subseteq D$, the same way as Σ is used as a generalisation of $+$ to a multiset of numbers.

The goal is to compute $\bigoplus_{d \in \hat{q}} w(d)$ for all $q \in Q$. The two problems we consider in the paper are specific cases of the above schematic outline. We assume D and Q to be large.

1.1 Prefix aggregation and its role as a parallel primitive

Prefix aggregation (also called scan) arises when data and queries are linearly ordered, say are both elements of \mathbb{R} , and $\hat{q} = \{d \in D \mid d < q\}$. Then we wish to compute $\bigoplus_{d < q} w(d)$. The eponymous case is when $D = Q$, so the aggregations are applied to all prefixes of the whole sequence of data points.

Concerning the importance of prefix aggregation as a computing primitive, it is well-known that, no matter what practical or theoretical parallel computation model is considered, sorting and scan are among the very first algorithms to be developed. The importance of the latter has even led to patents around the idea of including a prefix sum in the instruction set of a microprocessor [27], attempts of hardware implementations [16, 17] or adapting the actual algorithm to the architecture of the processor, sequential [14] or parallel [9]. Attempts of formal verification have also been undertaken [20, 10].

Scans were also researched as a primitive to implement parallel variants of many algorithms, including radix sort, quicksort, lexical analysis, polynomial evaluation, stream compaction, histograms and string comparison [4, 5], and also geometric partitioning algorithms [13].

1.2 Aggregation over dominated points

In this problem, data and queries are points in the m -dimensional space \mathbb{R}^m . For two such points we write $(x_1, \dots, x_m) < (y_1, \dots, y_m)$ when inequalities hold coordinate-wise, i.e., $x_1 < y_1, \dots, x_m < y_m$. Then let $\hat{q} = \{d \in D \mid d < q\}$.

The result of the algorithm should therefore consist of $\bigoplus_{d < q} w(d)$ for each query point q , which is the result of applying \bigoplus to the set of all weights of points in D which are *dominated* by q , i.e., coordinate-wise smaller than q .

Aggregation over dominated points can obviously be considered as a multidimensional generalisation of 1-dimensional prefix aggregation. Note however that typically prefix aggregation uses an associative operation \bigoplus with unit, while in the multidimensional setting we also require it to be commutative. The reason is that prefix aggregation has a natural order in which the elements are aggregated. In multidimensional setting there is no such natural order and hence, for the sake of producing a deterministic result, we require commutativity.

Aggregation over dominated points, referred to as *general prefix computations*, has been shown to be a parallel primitive which allows expressing many computational problems [22]. This approach has been subsequently extended to a general parallel computation model called *Broadcast with Selective Reduction* PRAM (BSR for short). The multiple criteria variant of BSR, introduced by Akl and Stojmenović [2, 3] after a number of earlier papers about single criterion BSR, is the one whose only parallel primitive is aggregation over dominated points. Many computational problems have been then shown to have constant-round BSR algorithms including: counting intersections of isothetic line segments, vertical segment visibility, maximal elements in m dimensions, ECDF searching, 2-set dominance counting and rectangle containment in m dimensions, rectangle enclosure and intersection counting in m dimensions [2], all nearest smaller values [28], all nearest neighbours and furthest pairs of points in a plane in L_1 metric, the all nearest foreign neighbours in L_1 metric and the all furthest foreign pairs of points in the plane in L_1 metric [18]. All of them therefore can be expressed by aggregation over dominated points.

Other applications of this primitive include calculating Empirical Cumulative Distribution Functions (ECDFs) in statistics, which are required in multivariate Kolmogorov-Smirnov, Cramér-von Mises and Anderson-Darling statistical tests [15]. The problem is also intensively studied, under the name range-aggregate queries by the spatial database community [1, 26, 21] and data warehouse community [11]. However, in this area one typically does not assume queries to be known in advance, so the focus is rather on storing data points in a data structure which allows efficient querying.

1.3 Our contribution

We prove the following.

► **Theorem 1.** *Aggregation over dominated points in \mathbb{R}^m , where m is constant, can be computed in $O(1)$ basic operations: sorting of lists of tuples, zipping and computing prefix aggregations as well as flat maps over such lists. By using flat lists, our algorithm expands the initial size of the input data from n to $O(n \log^{m-1} n)$ tuples.*

Our work thus creates a direct link between so far separate parallel primitives. In this respect, we follow the paradigm presented by Blelloch [4, 5], who has shown that many computational problems can be expressed by prefix aggregation. Our result does not add just one more such problem, but, by transitivity, all problems which have been previously shown to be expressible by aggregation over dominated points. An interesting theoretical conclusion can be drawn, that prefix aggregation suffices to express aggregation over dominated points, i.e., its own multidimensional generalisation.

Seen from another perspective, our work significantly simplifies the algorithm from our earlier paper Sroka et al. [23]. It presents a constant-rounds algorithms for solving the counting variant of the problem, written for MapReduce and designed to be *minimal* in the sense proposed by Tao et al. [25], which guarantees it evenly distributes computation among worker nodes. It distributes range trees explicitly, and we borrow the method to do so from that paper. However, it uses recursion to deal with consecutive dimensions and minimal group-by method from [25] to aggregate the counts. We regard it as a new contribution that all data processing tasks of this algorithm are replaced by invocations of a very few simple primitives of well-understood behaviours, and that this result neatly connects two computational tasks, each one with its own history of research of algorithms it can express.

This switch from a particular parallel model to high-level parallel primitives makes the algorithm simpler to understand, reducing the number and level of details which must be taken care of. Another benefit is the fact that the algorithm now avoids any direct references to the mechanisms of the parallel hardware it is running on, like processors, messages, shared resources, etc. It requires exactly those, which are used by the underlying implementations of the primitives we rely on. Among them, prefix aggregation is the only one, which allows combining values from an unbounded number of data elements together. However, the role of flat maps is also crucial, because they distribute certain computations, the results of which prefix aggregation later reduces. Finally, scans allow distributing many algorithms which in the centralised setting are based on sorting and iterating over data. Such approach has many advantages similar to those pointed out in [25], e.g., the resulting algorithms have strong guarantees concerning the way they distribute work and load. The ideas we present this paper can be viewed as generalisation of such approach to multidimensional setting the same way as range tree generalises binary search tree.

There is also an algorithm by Yufei Tao [24][Theorem 5] that uses an entirely different idea. It is directly tailored for the MPC model and takes care of reducing the maximal amount of communication between processors. It is based on partitioning space into fragments,

recursively solving problem over them and finally aggregating partial results. It achieves the optimal load $m^{O(m)}N/p$ with p processors. As far as we understand, neither of them can be adapted to use prefix aggregation as its main mechanism.

2 Primitives

2.1 Data model

We assume the data to be stored in immutable but ordered lists. We are going to transform lists into new lists. Such approach and immutability is typical for distributed architectures, while ordering can be achieved by imposing some order on nodes in the cluster and distributing values such that successive nodes have increasing elements. Initial ordering of the input data is arbitrary, but we assume that input values are equipped with some numerical IDs that define it and can be used to break ties if needed.

The initial list of data points (vectors in \mathbb{R}^m) is going to be referred to as D and the list of queries as Q . The weights of data points are represented by weight function $w : D \rightarrow A$. To make the exposition simpler, we assume that weights are defined for queries, too, and $w(q) = e$ for $q \in Q$, so that they do not interfere with aggregation.

2.2 Primitives and macros

In this section we postulate primitive operations that are used to express our algorithms as well we define some convenient macros that combine them.

► **Definition 2 (Sort).** For $x = [x_0, \dots, x_n]$ and some linear order relation $\preceq \subseteq X \times X$

$$\mathbf{sort}(x, \preceq) = [x_{i_1}, \dots, x_{i_n}],$$

where the multisets $\{\{x_{i_1}, \dots, x_{i_n}\}\}$ and $\{\{x_1, \dots, x_n\}\}$ are equal, and $x_{i_j} \preceq x_{i_{j+1}}$ for all j . ◻

It is known that radix sort and quicksort are expressible by prefix aggregation [4, 5], hence we could theoretically eliminate sorting from the list of primitives we rely on.

► **Definition 3 (FlatMap).** For $x = [x_0, \dots, x_n]$ and $f : X \rightarrow [Y]$, which applied to an element produces a list of elements as the result:

$$\mathbf{flatmap}(x, f) = f(x_0) \& \dots \& f(x_n),$$

where $\&$ is list concatenation. ◻

For convenience we define \mathbf{map} , which expects $f : X \rightarrow Y$ to produce single elements, by using $\mathbf{FlatMap}$ and composing f with list constructor $\mathbf{list}()$:

$$\mathbf{map}(x, f) := \mathbf{flatmap}(x, \mathbf{list}() \circ f),$$

so that

$$\mathbf{map}([x_1, \dots, x_n], f) = [f(x_1), \dots, f(x_n)]$$

Zip is an operation which takes two (or more) lists of equal length and combines them into a single list of tuples, created from elements at the same positions.

► **Definition 4 (Zip).** For lists $x^1 = [x_1^1, \dots, x_n^1], \dots, x^k = [x_1^k, \dots, x_n^k]$:

$$\mathbf{zip}(x^1, \dots, x^k) = [(x_1^1, \dots, x_1^k), \dots, (x_n^1, \dots, x_n^k)]. \quad \text{◻}$$

As usually immediately after zip we want to do something with those tuples, we define macros, which map or flatmap a provided function $f : X^k \rightarrow Y$ or $f : X^k \rightarrow [Y]$ on the tuples immediately:

$$\mathbf{mapzip}(x^1, \dots, x^k, f) := \mathbf{map}(\mathbf{zip}(x^1, \dots, x^k), f),$$

$$\mathbf{flatmapzip}(x^1, \dots, x^k, f) := \mathbf{flatmap}(\mathbf{zip}(x^1, \dots, x^k), f).$$

We define three variants of prefix aggregation of a list $[a_1, \dots, a_n]$ of elements of A , known from literature.

► **Definition 5 (Scan).** For an aggregation operation $\oplus : A \times A \rightarrow A$, its natural element e and a list $[a_1, \dots, a_n]$ of elements of A :

$$\mathbf{scan}([a_1, \dots, a_n], \oplus) = [a_1, a_1 \oplus a_2, \dots, a_1 \oplus \dots \oplus a_n],$$

$$\mathbf{scan}^-([a_1, \dots, a_n], \oplus) = [e, a_1, \dots, a_1 \oplus \dots \oplus a_{n-1}]. \quad \lrcorner$$

As we need \mathbf{scan}^- only for aggregating nondecreasing lists of numerical values with $\oplus = \max$ and $e = -\infty$ we define a macro: $\mathbf{shift}([x_1, \dots, x_n]) := \mathbf{scan}^-([x_1, \dots, x_n], \max) = [-\infty, x_1, \dots, x_{n-1}]$, which is indeed a right-shift of its input if $[x_1, \dots, x_n]$ is nondecreasing.

Another useful macro for lists of numerical values with $\oplus = \max$ and $e = -\infty$ is the following: $\mathbf{broadcastmax}([x_1, \dots, x_n]) := \mathbf{scan}(\mathbf{sort}([x_1, \dots, x_n], \geq), \max) = [x, \dots, x]$ where $x = \max(x_1, \dots, x_n)$. This macro indeed broadcasts the maximal value in a list to all positions. By zipping this list with another list we assure that the maximal value can be used for local processing of the latter, by \mathbf{map} .

► **Definition 6 (Segmented scan).** For an aggregation operation $\oplus : A \times A \rightarrow A$ and two lists $[a_1, \dots, a_n]$ of elements of A and $[t_1, \dots, t_n]$ of elements of some other set T , where the latter list is sorted:

$$\mathbf{sscan}([a_1, \dots, a_n], [t_1, \dots, t_n], \oplus) = [\bigoplus_{t_i=t_1}^{i \leq 1} a_i, \bigoplus_{t_i=t_2}^{i \leq 2} a_i, \dots, \bigoplus_{t_i=t_n}^{i \leq n} a_i].$$

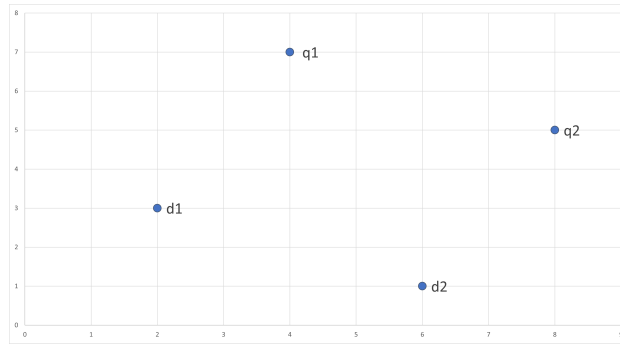
This means, that we essentially decompose the first argument list into maximal segments over which the corresponding elements of the second list remain identical, and then compute $\mathbf{scan}(s, \oplus)$ for each segment s separately, e.g., $\mathbf{sscan}([1, 2, 3, 4, 5, 6], [0, 0, 1, 1, 1, 2], +) = [1, 3, 3, 7, 12, 6]$.

Segmented scan can be expressed by standard scan (see [4]), but is typically designed and implemented independently, which gives a chance for better performance, in particular on complex, constrained architectures, such as GPU [8]. It can also be implemented as a data oblivious algorithm, whose memory access pattern is independent of the actual data being processed [19]. Note that scan is our only operation for combining unbounded number of elements, here by aggregation into a single value.

The primitives described in this section essentially define our model of hardware the algorithm is running on.

3 Checking dominance by polylogarithmic data expansion

In this section we present an important tool we need in our algorithm. It allows us to distribute the process of checking dominance relations later on. It can be viewed as a method to distribute a range tree that allows to query multidimensional data. It derives from the paper [23].



■ **Figure 1** Geometric visualization of the example data and queries. Note that the aggregation output in this case should be $w(d1)$ for $q1$ and $w(d1) \oplus w(d2)$ for $q2$.

We consider natural numbers written in binary notation, padded to some fixed length with leading 0's. Let x be a bitstring. Then let $P0(x)$ be the set of all bitstrings v such that $v0$ is a prefix of x , including the empty prefix, should x start with a 0, e.g., $P0(01010) = \{0101, 01, \varepsilon\}$. Similarly, let $P1(x)$ be the set of all bitstrings v , such that $v1$ is a prefix of x .

► **Lemma 7.** *Suppose x and y are natural numbers represented as bitstrings of equal length, perhaps with leading 0's.*

If $x < y$ then $P0(x) \cap P1(y)$ has exactly one element, and if $x \geq y$ then $P0(x) \cap P1(y) = \emptyset$.

Proof. $x < y$ iff their longest common prefix is followed by 0 in x and by 1 in y . Hence $P0(x) \cap P1(y)$ is nonempty iff $x < y$, which takes care of the $x \geq y$ part.

To rule out the possibility that $x < y$ and $P0(x) \cap P1(y)$ has more than 1 element, it is enough to observe that the longest element in $P0(x) \cap P1(y)$ is at the same time the shortest one because it has to be followed by different symbols in x and y . ◀

Let (x_1, \dots, x_n) be a tuple of bitstrings. Define $P0((x_1, \dots, x_n)) = P0(x_1) \times \dots \times P0(x_n)$, and similarly $P1((x_1, \dots, x_n)) = P1(x_1) \times \dots \times P1(x_n)$.

► **Lemma 8.** *Let $\vec{x} = (x_1, \dots, x_n)$ and $\vec{y} = (y_1, \dots, y_n)$ be two tuples of natural numbers encoded as bitstrings, coordinate-wise of equal lengths.*

If \vec{x} is coordinate-wise smaller than \vec{y} then $P0(\vec{x}) \cap P1(\vec{y})$ is a singleton, and otherwise it is empty.

Proof. Follows from Lemma 7. ◀

4 Algorithm

We present the algorithm in several groups of numbered instructions and for each add explanations. Each such group is followed by an example of its action on a very small 2-dimensional dataset with $D = [(2, 3)_{d1}, (6, 1)_{d2}]$ and $Q = [(4, 7)_{q1}, (8, 5)_{q2}]$, which is intended to help with the explanation of the algorithm. Value subscripts indicate their IDs, whose order is $d1, q1, d2, q2$. See Figure 1 for a geometrical visualization.

The algorithm works on data and query points together, so first the union $DQ = D \cup Q$ of those two lists is created.

```
0    $DQ = \text{flatmapzip}(D, Q, \lambda x, y. [x, y])$ 
```

$$0 \quad DQ = [(2, 3)_{d1}, (4, 7)_{q1}, (6, 1)_{d2}, (8, 5)_{q2}]$$

The next group of instructions is used to compute, for each dimension, the rank of each tuple's coordinate in that dimension and the total number of unique coordinates.

Let $less : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ be defined as

$$less(a, b) = \begin{cases} 1 & \text{if } a < b \\ 0 & \text{if } a \geq b \end{cases} .$$

For each dimension $i = 1, \dots, m$ we add the following instructions.

$$\begin{array}{ll} 5i - 4 & S_i = \mathbf{map}(DQ, \lambda x.x[i]) \\ 5i - 3 & T_i = \mathbf{sort}(S_i, \leq) \\ 5i - 2 & W_i = \mathbf{shift}(T_i) \\ 5i - 1 & R_i = \mathbf{scan}(\mathbf{mapzip}(W_i, T_i, less), +) \\ 5i & U_i = \mathbf{broadcastmax}(R_i) \end{array}$$

<p>for $i = 1$:</p> <ol style="list-style-type: none"> 1 $S_1 = [2_{d1}, 4_{q1}, 6_{d2}, 8_{q2}]$ 2 $T_1 = [2_{d1}, 4_{q1}, 6_{d2}, 8_{q2}]$ 3 $W_1 = [-\infty, 2_{d1}, 4_{q1}, 6_{d2}]$ 4 $R_1 = [1, 2, 3, 4]$ 5 $U_1 = [4, 4, 4, 4]$ 	<p>for $i = 2$:</p> <ol style="list-style-type: none"> 6 $S_2 = [3_{d1}, 7_{q1}, 1_{d2}, 5_{q2}]$ 7 $T_2 = [1_{d2}, 3_{d1}, 5_{q2}, 7_{q1}]$ 8 $W_2 = [-\infty, 1_{d2}, 3_{d1}, 5_{q2}]$ 9 $R_2 = [1, 2, 3, 4]$ 10 $U_2 = [4, 4, 4, 4]$
---	--

Line $5i - 4$ extracts the sequence of i -th coordinates of vectors from DQ , which is then sorted in line $5i - 3$, so that it can be shifted by one position to the right in line $5i - 2$. Then $less$ in line $5i - 1$ essentially compares each element of T_i with its predecessor and produces a list of 1s and 0s, with 1 on positions with a difference and 0 otherwise. Therefore prefix sum of that sequence computes ranks of the elements of T_i . In particular, on the last index there will be total number of unique elements. We need a list with this value present at every position. It is computed in line $5i$ with **broadcastmax**.

Now we transform each rank into its binary representation of fixed length which can be viewed as coordinates of that value in a binary search tree. Let $bin(n, k)$ for $n \leq k$ be defined as a binary expansion of $n - 1$ using exactly $\lceil \log k \rceil$ binary digits, i.e., with leading zeros if necessary.

Again for each dimension $i = 1, \dots, m$ we add the following instructions.

$$5m + i \quad BR_i = \mathbf{sort}(\mathbf{mapzip}(R_i, U_i, bin), ID)$$

<p>for $i = 1$:</p> <ol style="list-style-type: none"> 11 $BR_1 = [00_{d1}, 01_{q1}, 10_{d2}, 11_{q2}]$ 	<p>for $i = 2$:</p> <ol style="list-style-type: none"> 12 $BR_2 = [01_{d1}, 11_{q1}, 00_{d2}, 10_{q2}]$
--	--

The instructions in lines $5m + 1, \dots, 6m$ transform the original data in each dimension to the rank space, where values are replaced by their ranks within the whole dataset, written in binary. Ranks isomorphically preserve all inequalities of the original real values, and hence preserve the results of aggregations to be computed, too.

96:8 Aggregating over Dominated Points by Sorting, Scanning, Zip and Flat Maps

Let $P0(x)$ and $P1(x)$ be as in Lemma 8. Furthermore, let P be a function from m -tuples of binary strings to sets of m -tuples of binary strings, defined as follows:

$$P(b_1, \dots, b_m) = \begin{cases} P0(b_1, \dots, b_m) & \text{if } (b_1, \dots, b_m) \text{ is a data point;} \\ P1(b_1, \dots, b_m) & \text{if } (b_1, \dots, b_m) \text{ is a query point.} \end{cases}$$

$$6m + 1 \quad EDQ = \mathbf{flatmapzip}(BR_1, \dots, BR_m, P)$$

$$13 \quad EDQ = [(\varepsilon, \varepsilon)_{d1}, (0, \varepsilon)_{d1}, (0, \varepsilon)_{q1}, (0, 1)_{q1}, (1, \varepsilon)_{d2}, (1, 0)_{d2}, (\varepsilon, \varepsilon)_{q2}, (1, \varepsilon)_{q2}]$$

This results in a new, expanded sequence EDQ consisting of tuples of bitstrings. We assume they inherit ID and weights from their originating data and query points. This sequence will contain many duplicated tuples, but with different IDs. This operation increases the total size of data from $O(n)$ to $O(n \log^m n)$.

Let \leq_{lex} be doubly lexicographic ordering relation on tuples from EDQ : lexicographic for bitstrings in each coordinate and lexicographic between coordinates, with the (crucial) additional requirement, that in case of a tie of two tuples data points precede queries.

$$6m + 2 \quad SEDQ = \mathbf{sort}(EDQ, \leq_{lex})$$

$$14 \quad SEDQ = [(\varepsilon, \varepsilon)_{d1}, (\varepsilon, \varepsilon)_{q2}, (0, \varepsilon)_{d1}, (0, \varepsilon)_{q1}, (0, 1)_{q1}, (1, \varepsilon)_{d2}, (1, \varepsilon)_{q2}, (1, 0)_{d2}]$$

$SEDQ$ is composed of segments consisting of duplicates, differing only by IDs and weights. Data points come before queries among each segment of equal values. At this point we have prepared all data we need, it remains to perform several steps of aggregations.

$$6m + 3 \quad A_1 = \mathbf{sscan}(\mathbf{map}(SEDQ, \lambda x. w(x)), SEDQ, \oplus)$$

$$15 \quad A_1 = [w(d1)_{d1}, w(d1)_{q2}, w(d1)_{d1}, w(d1)_{q1}, e_{q1}, w(d2)_{d2}, w(d2)_{q2}, w(d2)_{d2}]$$

In line $6m + 3$ we compute segmented prefix sums of the weights of the sorted data and queries. All bitstring coordinates serve as tags for segmentation. This way a list is created, where each query point q corresponds (by position on the list) to the aggregation of all weights of data points which yielded the same tuple of bitstrings. By Lemma 8, each such identical tuple originates from a data point d such that $d < q$, and, moreover, for each fixed tuple t derived from q there is one-to-one correspondence between such tuples and data points which also produced t and are (therefore) dominated by q .

Queries have been assigned neutral weight, so they do not interfere with the scan. However, some aggregation happens at the positions of data points, too.

Let \leq_{ID} be ordering relation on tuples which compares their associated ID values.

$$6m + 4 \quad A_2 = \mathbf{sort}(A_1, \geq_{ID})$$

$$6m + 5 \quad A_3 = \mathbf{sscan}(A_2, \mathbf{map}(A_1, \lambda x. ID(x)), \oplus)$$

16	$A_2 = [w(d2)_{q2}, w(d1)_{q2}, w(d2)_{d2}, w(d2)_{d2}, e_{q1}, w(d1)_{q1}, w(d1)_{d1}, w(d1)_{d1}]$
17	$A_3 = [w(d2)_{q2}, (w(d2) \oplus w(d1))_{q2}, w(d2)_{d2}, (w(d2) \oplus w(d2))_{d2}, e_{q1}, w(d1)_{q1}, w(d1)_{d1}, (w(d1) \oplus w(d1))_{d1}]$

Line $6m + 4$ is a sort of partial aggregations by ID in nonincreasing sequence, which creates a continuous segment of aggregation values corresponding to each query. There are also separate segments of data points, which are irrelevant now.

After that in line $6m + 5$ we compute segmented prefix sums of the already partially aggregated weights of the sorted data and queries, whereby their ID values serve as tags for segmentation. This way partial aggregations for each query are further aggregated, so that the total aggregation of each query q corresponds, by position, to the last tuple which originated from q . This total aggregation for q comes from all d such that $P(q) \times P(d) \neq \emptyset$ (those sets are created in line $6m + 1$). By Lemma 8, $\{d | P(q) \cap P(d) \neq \emptyset\} = \{d | d < q\}$, and each such d in the r.h.s. is witnessed by exactly one element of $P(q) \cap P(d)$.

At this point the aggregation of each query is already computed, but the data contains many partial aggregations for the same query, too. Therefore the last task is to distribute the total aggregation of each query to all tuples with the same ID.

This can be significantly simplified if A is linearly ordered and $a \oplus b \geq a$ for all $a, b \in A$. In this case for each query we need the maximum aggregation among all partial ones. Therefore a segmented variant of **broadcastmax** with ID defining segmentation would do that. Otherwise the method to achieve the goal is more complex and described below.

$$\text{Let } neutral_if_eq(id_1, id_2, a) = \begin{cases} e & \text{if } id_1 = id_2 \\ a & \text{if } id_1 \neq id_2 \end{cases} .$$

$6m + 6$	$A_4 = \mathbf{sort}(A_3, \leq_{ID})$
$6m + 7$	$H = \mathbf{shift}(\mathbf{map}(A_4, \lambda x.ID(x)))$
$6m + 8$	$A_5 = \mathbf{mapzip}(A_4, H, \mathbf{map}(A_4, \lambda x.ID(x)), neutral_if_eq)$
$6m + 9$	$Out = \mathbf{sscan}(A_5, \mathbf{map}(\lambda x.ID(x), A_5), \oplus)$

18	$A_4 = [(w(d1) \oplus w(d1))_{d1}, w(d1)_{d1}, w(d1)_{q1}, e_{q1}, (w(d2) \oplus w(d2))_{d2}, w(d2)_{d2}, (w(d2) \oplus w(d1))_{q2}, w(d2)_{q2}]$
19	$H = [-\infty, d1, d1, q1, q1, d2, d2, q2]$
20	$A_5 = [(w(d1) \oplus w(d1))_{d1}, e_{d1}, w(d1)_{q1}, e_{q1}, (w(d2) \oplus w(d2))_{d2}, e_{d2}, (w(d2) \oplus w(d1))_{q2}, e_{q2}]$
21	$Out = [(w(d1) \oplus w(d1))_{d1}, (w(d1) \oplus w(d1))_{d1}, w(d1)_{q1}, w(d1)_{q1}, (w(d2) \oplus w(d2))_{d2}, (w(d2) \oplus w(d2))_{d2}, (w(d2) \oplus w(d1))_{q2}, (w(d2) \oplus w(d1))_{q2}]$

In line $6m + 6$ we reverse the sort order of A_4 . Now the complete aggregations come at the beginning of each segment, and, moreover, the ID values are nondecreasing, hence we can shift them in line $6m + 7$. Line $6m + 8$ resets all computed weights to the neutral e , except at the beginning of each segment, where the complete aggregation is present. Finally, segmented scan in line $6m + 9$ aggregates these values with neutral elements elsewhere, producing the desired output.

Now each query ID is accompanied by the aggregation of weights of its dominated points, which means that the desired output has been computed. In total it took $6m + 9$ instructions and at most that many intermediate lists of data created. As it can be seen, the output of the example agrees with the output determined from the geometric presentation in Figure 1.

5 Improvement by one logarithm

It is possible to reduce the amount of data generated by a factor of $\log n$.

One of the coordinates (say: the last one) is chosen. It is not replaced by ranks and left in the form of real numbers. The remaining ones are replaced by ranks and prefixes are generated, exactly as in the basic algorithm, from both data points and queries. This results in a multiset of $\leq n \log^{m-1} n$ tuples with $m - 1$ coordinates in the form of bitstrings and the last, m -th coordinate being real number. Identifiers are retained.

Now sort the data and queries EDQ into $SEDQ$ (see line $6m + 2$ and explanation thereof above) according to the doubly lexicographic order, with queries preceding data points in case of equality of all coordinates (including the m -th).

This results in segments of data elements with $m - 1$ first coordinates equal, sorted according to the last coordinate within the segment. Then the remainder of the algorithm is executed exactly as in the basic version.

6 Relation to range trees and complexity

The algorithm we have presented above is derived from our earlier MapReduce algorithm Sroka et al. [23], and is indeed a parallelisation of the common sequential algorithm, based on range trees. The move to the rank space with ranks expressed as binary expansions is equivalent to speaking about elements in terms of their positions in a balanced binary tree, whose leaves hold the sorted data. The binary encodings then correspond to branches in the tree, and their prefixes to the positions where attached trees of smaller dimensions are located.

Our algorithm inherits its total data complexity of $O(n \log^{m-1} n)$ from the range tree algorithm. This distributed data structure is generated by flat maps, while the parallelisation is achieved by expressing operations on the tree in terms of sorting, zipping and prefix aggregation.

Time complexity of our algorithm depends very much on the underlying architecture and complexity of the primitive operations, but its analysis is pretty straightforward in each case, since it is a fixed length sequence of operations of very well known properties, applied to lists of data of sizes easy to determine. In particular, no matter what architecture it is executed on, if the implementations of primitives do the same total work as their sequential variants, then the whole algorithm will also have the total work of the sequential algorithm using range trees.

Indeed, in the sequential case the only nonlinear (and thus dominating) operation is sorting, and the size of the data is $O(n \log^{m-1} n)$. One linearithmic sort takes time $O(n \log^{m-1} n \cdot \log(n \log^{m-1} n)) = O(n \log^m n)$, which is equal to the worst case of the standard sequential implementation, calculated as creating the range tree with $n/2$ data points and then processing $n/2$ queries by this tree. There is a one logarithm better variant of Chazelle [6], which however works only for counting.

For the MPC model, it is known that for $\delta > 0$, sorting and scanning of n values can be performed deterministically in a constant number of rounds using n^δ space per machine, $O(n)$ total space, and $\text{poly}(n)$ local computation, which follows directly from analogous bounds for MapReduce computation. The load can be made $O(N/p)$ [7]. This implies that our algorithm in dimension m can be implemented deterministically in $O(m)$ rounds, with n^δ space per machine, $O(n \log^{m-1} n)$ total space and $\text{poly}(n)$ local computation, with load $O(N \log^{m-1} N/p)$.

By comparison, the algorithm by Hu et al. [12] achieves load $O(Np^{-1} \log^{m-1} p)$ with p processors, which is better than what get in this paper. This is not surprising, since our reliance on high-level primitives gives us much less freedom in designing the computation mechanism and achieving low loads. In particular, the set of instructions we use does not permit shifting the computational effort into the local computation, which is the method to lower the loads. The algorithm of Tao [24] does it to an even higher extent, arriving at load $m^{O(m)}N/p$.

Our earlier algorithm from Sroka et al. [23] is quite similar, but at that time we did only tests of an implementation on MapReduce, for which it has been designed. The running times did not seem to scale linearly with the number of machines. Moreover, for its present form we can do complexity analysis, greatly simplified by known complexities of the primitives we use.

7 Summary and future research

In this paper we have presented algorithm for aggregating over dominated points in \mathbb{R}^m , where m is constant. Our algorithm is based on a limited set of primitive operations: sorting, prefix aggregations, zip and flat maps. All those primitive operations are well studied and their efficient implementations exist for essentially all distributed architectures.

This proves that one-dimensional prefix aggregation allows expressing its own multidimensional generalisation. The latter problem has many practical applications, as well as it is known to be a parallel primitive, allowing to express in turn further problems. By transitivity, our result expresses all those problems in terms of the above mentioned primitive operations.

We consider our result to be primarily of theoretical interest at present, before experimental tests are conducted.

We believe that our algorithm may turn out to be quite practical. First of all, it is absolutely transparent and does not hide any significant computation steps. The local computation is on the level of individual tuples, only. No large collections of data need to be broadcasted to computation nodes and there is no limit on number of such nodes. Otherwise we use high-level primitives of very well understood algorithmic properties, and whose highly optimised implementations exist for virtually all hardware platforms. Also the organisation of the algorithm into a sequence of functional operations (without any branch) on immutable ordered lists is very convenient for implementation. Last but not least, the choice of primitives guarantees that the algorithm has several desired properties, e.g., it is minimal in the sense of [25].

On the other hand, the $\log^{m-1} n$ memory footprint will be a problem in larger dimensions.

Therefore an obvious item on the “further research” list is undertaking experiments with the algorithm on diverse parallel platforms.

References

- 1 Pankaj K. Agarwal, Lars Arge, Sathish Govindarajan, Jun Yang, and Ke Yi. Efficient external memory structures for range-aggregate queries. *Computational Geometry*, 46(3):358–370, 2013. doi:10.1016/j.comgeo.2012.10.003.
- 2 Selim G. Akl and Ivan Stojmenović. Multiple criteria BSR: An implementation and applications to computational geometry problems. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994.
- 3 Selim G. Akl and Ivan Stojmenović. Broadcasting with selective reduction: A powerful model of parallel computation. *Parallel and distributed computing handbook*, pages 192–222, 1996.

- 4 Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11):1526–1538, 1989. doi:10.1109/12.42122.
- 5 Guy E. Blelloch. Prefix sums and their applications. In John H Reif, editor, *Synthesis of parallel algorithms*. Morgan Kaufmann Publishers Inc., 1993.
- 6 Bernard Chazelle. Lower bounds for orthogonal range searching II. the arithmetic model. *J. ACM*, 37(3):439–463, 1990. doi:10.1145/79147.79149.
- 7 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011. doi:10.1007/978-3-642-25591-5_39.
- 8 Mark Harris and Michael Garland. Optimizing parallel prefix operations for the Fermi architecture. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 29–38. Morgan Kaufmann, Boston, 2012. doi:10.1016/B978-0-12-385963-1.00003-4.
- 9 Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.
- 10 Ralf Hinze. An algebra of scans. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004. doi:10.1007/978-3-540-27764-4_11.
- 11 Seokjin Hong, Byoungso Song, and Sukho Lee. Efficient execution of range-aggregate queries in data warehouse environments. In Hideko S. Kunii, Sushil Jajodia, and Arne Sølvberg, editors, *Conceptual Modeling - ER 2001, 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001, Proceedings*, volume 2224 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2001. doi:10.1007/3-540-45581-7_23.
- 12 Xiao Hu, Ke Yi, and Yufei Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Trans. Database Syst.*, 44(2):6:1–6:36, 2019. doi:10.1145/3311967.
- 13 Y. Charlie Hu, Shang-Hua Teng, and S. Lennart Johnsson. A data-parallel implementation of the geometric partitioning algorithm. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997*. SIAM, 1997.
- 14 Hung-Ming Lai and Jenq-Kuen Lee. Efficient support of the scan vector model for RISC-V vector extension. In *Workshop Proceedings of the 51st International Conference on Parallel Processing, ICPP Workshops 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, pages 15:1–15:8. ACM, 2022. doi:10.1145/3547276.3548518.
- 15 Nicolas Langrené and Xavier Warin. Fast multivariate empirical cumulative distribution function with connection to kernel density estimation. *Computational Statistics & Data Analysis*, 162:107267, 2021. doi:10.1016/j.csda.2021.107267.
- 16 Rong Lin, Koji Nakano, Stephan Olariu, Maria Cristina Pinotti, James L. Schwing, and Albert Y. Zomaya. Scalable hardware-algorithms for binary prefix sums. *IEEE Trans. Parallel Distributed Syst.*, 11(8):838–850, 2000. doi:10.1109/71.877941.
- 17 Rong Lin, Koji Nakano, Stephan Olariu, and Albert Y. Zomaya. An efficient parallel prefix sums architecture with domino logic. *IEEE Trans. Parallel Distributed Syst.*, 14(9):922–931, 2003. doi:10.1109/TPDS.2003.1233714.
- 18 Robert A. Melder and Ivan Stojmenovic. Constant time BSR solutions to l_1 metric and digital geometry problems. *J. Math. Imaging Vis.*, 5(2):119–127, 1995. doi:10.1007/BF01250524.
- 19 Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 373–384. ACM, 2021. doi:10.1145/3409964.3461783.

- 20 Mohsen Safari and Marieke Huisman. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theor. Comput. Sci.*, 912:81–98, 2022. doi:10.1016/j.tcs.2022.02.027.
- 21 Yexuan Shi, Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Bolin Ding, and Lei Chen. Efficient approximate range aggregation over large-scale spatial data federation. *IEEE Trans. Knowl. Data Eng.*, 35(1):418–430, 2023. doi:10.1109/TKDE.2021.3084141.
- 22 Frederick N. Springsteel and Ivan Stojmenovic. Parallel general prefix computations with geometric, algebraic, and other applications. *Int. J. Parallel Program.*, 18(6):485–503, 1989. doi:10.1007/BF01381719.
- 23 Jacek Sroka, Artur Leśniewski, Mirosław Kowaluk, Krzysztof Stencel, and Jerzy Tyszkiewicz. Towards minimal algorithms for big data analytics with spreadsheets. In Foto N. Afrati and Jacek Sroka, editors, *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, pages 1:1–1:4. ACM, 2017. doi:10.1145/3070607.3075961.
- 24 Yufei Tao. Massively parallel entity matching with linear classification in low dimensional space. In Benny Kimelfeld and Yael Amerdamer, editors, *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, volume 98 of *LIPICs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ICDT.2018.20.
- 25 Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal MapReduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 529–540, New York, NY, USA, 2013. ACM. doi:10.1145/2463676.2463719.
- 26 Yufei Tao and Dimitris Papadias. Range aggregate processing in spatial databases. *IEEE Trans. Knowl. Data Eng.*, 16(12):1555–1570, 2004. doi:10.1109/TKDE.2004.93.
- 27 Uzi Vishkin. Prefix sums and an application thereof, April 1 2003. US Patent 6,542,918.
- 28 Limin Xiang and Kazuo Ushijima. ANSV problem on bsrs. *Inf. Process. Lett.*, 65(3):135–138, 1998. doi:10.1016/S0020-0190(97)00214-7.