Internal Masked Prefix Sums and Its Connection to Fully Internal Measurement Queries

Rathish Das[1], Meng He[2], Eitan Kondratovsky[1](✉), J. Ian Munro[1], and Kaiyu Wu[1]

[1] Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
{rathish.das,eitan.kondratovsky,imunro,k29wu}@uwaterloo.ca
[2] Faculty of Computer Science, Dalhousie University, Halifax, Canada
mhe@cs.dal.ca
https://cs.uwaterloo.ca/~imunro/, https://web.cs.dal.ca/~mhe/

Abstract. We define a generalization of the prefix sum problem in which the vector can be masked by segments of a second (Boolean) vector. This problem is shown to be related to several other prefix sum, set intersection and approximate string match problems, via specific algorithms, reductions and conditional lower bounds. To our knowledge, we are the first to consider the fully internal measurement queries and prove lower bounds for them. We also discuss the hardness of the sparse variation in both static and dynamic settings. Finally, we provide a parallel algorithm to compute the answers to all possible queries when both vectors are fixed.

1 Introduction

The prefix sums problem (also known as *scans*) is one in which one seeks to preprocess an array of n numbers to answer prefix sum queries. This problem is widely known and has been motivated by many fields such as parallel algorithm, graph theory, and more [5,10]. Pătraşcu et al. [23] proved tight lower bounds for the query time when using linear space. Later, Bille et al. [4] provided tight lower bounds in the dynamic model where insertion, deletion, and modifications of the array's numbers are allowed. On the implementation side, Pibiri and Venturini [24] give an overview of current techniques and how well they perform in practice.

The internal model has received much attention in recent years. In this setting, the input is given as a sequential string and the queries are asked on different substrings of the input. One of the first problems in this research field was the *internal pattern matching problem*. In this problem, a string S is preprocessed to answer queries of the form: "report all occurrences where a substring of S is located in another substring of S" [15,17]. Since then, many internal query problems were introduced. Examples include the longest common prefix of two substrings of S, computing the periods of a substring of S, etc. We refer the interested reader to [16], which contains an overview of the literature.

The *internal masked prefix sum problem* takes as input an m-bit mask B and an array A of n numbers, where $m \geq n$, and supports the query: "report the prefix sum of the numbers against some substring of the mask". It is easy to see that the subtraction of two such prefix sums supports any masked sum of a substring of A. That is, subtracting the prefix sum until position i from the prefix sum until position $j > i$, where i, j are the substring positions. A simplified case of the problem is when A consist of binary values, i.e. $A \in \{0,1\}^*$. In this case, the masked prefix sums are the inner products of the substrings.

Clifford et al. [7] introduced the problem of dynamic data structure that supports the inner product (and other measurements) between an m-length pattern and any m-length substring of the text, where m is fixed. Here dynamic means that substitutions of letters in the pattern and in the text are allowed. The additional measurements that were considered are the Hamming distance and exact matching with wildcards. It was shown that in the dynamic setting, both query and update cannot be done in $O(n^{\frac{1}{2}-\varepsilon})$ time, for some $\varepsilon > 0$, unless the OMv (Online Boolean Matrix-Vector Multiplication) conjecture is false.

Our contributions are the following.

1. We give a preprocess-query time trade-off that matches the lower bound of the batched problem up to logarithmic factors. The trade-off algorithm works in $O(\frac{nm}{f(n)} \log f(n))$[1] preprocessing time and $O(\frac{nm}{f(n)})$ space and answers queries in $O(f(n))$ time, for any $1 \leq f(n) \leq n$.
2. We prove a lower bound for the batched problem in which the algorithm preprocesses the data to answer a batch of n queries, where n is the length of the input. We show a lower bound of $p(n) + nq(n) = \Omega(n^{\frac{3}{2}-\varepsilon})$, for any $\varepsilon > 0$, where $p(n)$ is the preprocessing time and $q(n)$ is the query time. This lower bound illustrates that queries that consist of substrings from both A and B even in a static setting, have similar hardness as the dynamic setting but on substrings with fixed length.
3. We show a $(1 + \varepsilon)$-approximation algorithm for the *internal masked prefix sums* that works in near constant time for any $\varepsilon > 0$.
4. We give a parallel algorithm that computes all the internal prefix sums in $O(\log n + \log m)$ span and $O(nm)$ work to answer all the possible queries that are stored explicitly, or $O(\log n + \log m)$ span and $O(\frac{nm}{\log n})$ work for (implicit) constant-time queries.
5. We consider the *sparse internal inner product* and we show conditional lower bounds from SetDisjointness and 3SUM in the static and dynamic settings, respectively.

The paper is organized as follows: In Sect. 2 we give the formal definition of our problems and other data structures that we will be using as building blocks. In Sect. 3 we study the *internal masked prefix sum*. We give the preprocess-query trade-off data structure for the problem along with the conditional lower bound. Furthermore, we study an approximate form of the problem and finally give a

[1] We will use log to denote $\log_2$, though as our log all eventually end up in asymptotic notation, the constant bases are irrelevant.

parallel algorithm that can be used in the preprocessing of the data structures. In Sect. 4 we study the *sparse internal inner product* and give conditional lower bounds for the problem. Finally, in Sect. 5 we discuss a related problem, of calculating Hamming distances and how it can be reduced to the *internal masked prefix sum* problem.

2 Preliminaries

Let Σ be an alphabet. A *string* S over Σ is a finite sequence of letters from Σ. By $S[i]$, for $1 \leq i \leq |S|$, we denote the i^{th} letter of S. The *empty string* is denoted by ε. By $S[i..j]$ we denote the string $S[i] \cdots S[j]$, called a *substring* of S (if $i > j$, then the substring is the empty string). A substring is called a *prefix* if $i = 1$ and a *suffix* if $j = |S|$.

In the paper, we assume that Σ consists of nonnegative numbers. The nonnegativity ensures that binary search on the partial prefix sums is well defined.

Definition 1 (`MaskedPrefixSumProblem`).
Input: *A bit vector (mask) B and an array of numbers A of lengths m and n, respectively. The goal is to preprocess a data structure that answers the masked prefix sum queries of the following form.*
Query: *Given i and k, where $1 \leq k \leq |A|, 1 \leq i \leq |B| - k + 1$. We wish to report the sum of the first k numbers of A that corresponds to 1s in the k-length submask of B located at position i. Formally,*

$$\sum_{j=1}^{k} A[j] \cdot B[i+j-1] = \langle A[1 \dots k], B[i \dots i+k-1] \rangle.$$

Example 1. Let $A = [1, 0, 2, 0, 0, 0, 4, 8]$ and $B = 10100111001$. For query $k = 7$ and $i = 5$ the answer is 6. To see this, we look at the first 7 numbers of A against the 7-length submask starting at position 5. After masking out the elements of A, which are against 0s, we are left with 2 and 4 which sum to 6.

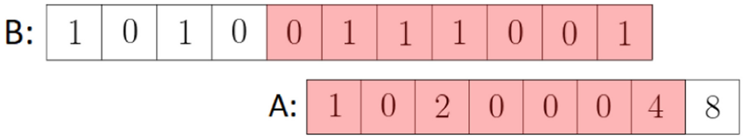

Fig. 1. Highlighted cells are those used in the query.

Definition 2 (`SparseMaskedPrefixInnerProduct`).
Input: *A bit vector (mask) B and a bit vector (numbers) A such that the sum of the number of 1s in A and the number of 1s in B is n, i.e. the problem can be represented by writing down the n positions of the ones. The goal is to*

preprocess a data structure that answers the sparse internal inner product queries of the following form.
Query: *Given i and k, where $1 \leq k \leq |A|, 1 \leq i \leq |B| - k + 1$. We wish to report the inner product of the first k bits of A against the k-length submask of B located at position i.*

In the paper, we consider the RAM model with word size $w = \log n$. We assume that each integer fits into one $\log n$-length RAM word. Thus the sum of n integers is at most n^2, and each prefix sum fits into constant number of RAM words (more specifically 2). Notice that in some problems $w = \log U$, where U is the universe size of the problem. When the context is clear, we write a capital U to indicate that this is the case.

In our solutions to the above `MaskedPrefixSumProblem` and the `SparseMaskedPrefixInnerProduct` problems, we will extensively use data structures for the predecessor/successor problem.

The predecessor problem is to preprocess a set S of N integers from a universe of size U, to answer queries of the form: "Given an input i, return the largest integer in S smaller than i". The successor problem is analogous. We only state the results for the predecessor query but all of the following data structures can handle the successor query as well with the same complexities.

Willard's y-fast tries [25] gives a $O(N)$ word space solution to the problem with $O(\log\log U)$ query time. Another approach is Fredman and Willard's fusion trees [11], which also uses $O(N)$ words of space and supports the predecessor query in $O(\sqrt{\log N})$ amortized time. Andersson [1] showed how to make the query in worst case time $O(\sqrt{\log N})$.

Combining these results, we obtain the following lemma. The decision on which data structure to use follows from the values of U and N.

Lemma 1 (Predecessor Query). *There is a data structure for the predecessor problem that uses $O(N)$ words of space and query time $O(\min\{\log\log U, \sqrt{\log N}\})$.*

We note that the data structure of Beame and Fich [3] gives a better running time of $O(\min\{\frac{\log\log U}{\log\log\log U}, \sqrt{\frac{\log N}{\log\log N}}\})$, at the cost of increasing the space to $O(N^{O(1)})$ words of space.

3 Data Structures for Masked Prefix Sum

In this section, we design data structures for masked prefix sum and some of its variants. More specifically, we present a time-space trade-off for this problem (Sect. 3.1) and prove a conditional lower bound (Sect. 3.2). We also design data structures for the dynamic version (Sect. 3.3) and the approximate version (Sect. 3.4) of the masked prefix sum problem. Finally, we consider a related problem of designing a parallel algorithms to compute the answers to all possible queries over a given instance of this problem (Sect. 3.5).

3.1 Time-Space Trade-off

We now present a data structure for the masked prefix sum problem. With it, we achieve a trade-off between space/preprocessing cost and query time. We note that the data structure can be immediately generalized to solve other internal measurements - such as those studied by Clifford et al. [7] by replacing the inner product by any linear function, i.e., any function g over two strings that can be written as

$$g(S, T) = \sum_i g(S[i], T[i]).$$

The main result can be stated as:

Theorem 1. *Given a bit vector B of length m and an array A of length n, there is a data structure that uses $\frac{mn}{f(n)}$ words of space that can answer masked prefix sum queries in $O(f(n))$ time, for any function $f(n)$ with $0 < f(n) < n$. The preprocessing time is $O(\frac{mn}{f(n)} \log f(n))$.*

Furthermore, for any $c > 0$, there is a data structure that uses $\frac{mn}{f(n)} + \frac{n^{1+c}}{c \log n}$ words of space and can answer masked prefix sum queries in $O(\frac{f(n)}{c \log n})$ time. The preprocessing time is $O(\frac{mn}{f(n)} \log f(n) + \frac{n^{1+c}}{c \log n})$.

Proof. We do this by writing down the answer to the queries for each ending position in A that is a multiple of $f(n)$ and each offset in B, using $\frac{mn}{f(n)}$ words of space. That is, we create a table D such that

$$D[i, k] := \sum_{j=1}^{kf(n)} A[j] \cdot B[i + j - 1]$$

For every query length k not a multiple of $f(n)$, we find the largest multiple of $f(n)$ and write $k = k'f(n) + \ell$, where $k' = \lfloor k/f(n) \rfloor$. We then break the sum into

$$\sum_{j=1}^{k} A[j] \cdot B[i + j - 1] = \sum_{j=1}^{k'f(n)} A[j] \cdot B[i + j - 1] + \sum_{j=k'f(n)+1}^{k} A[j] \cdot B[i + j - 1]$$

The first term is $D[i, k']$, and we compute the second term by computing $A[j] \cdot B[i + j - 1]$ one by one, which requires $O(f(n))$ time. Thus, the query time is $O(f(n))$.

To slightly improve the time, we consider all bit masks of length $c \log n$, of which there are n^c such bit masks. For each index that is a multiple of $c \log n$ in A and each possible bit mask M of length $c \log n$, we write down

$$D'[M, j] := \sum_{k=0}^{c \log n - 1} A[j + k] \cdot M[k]$$

and we store these values in the table D', using $\frac{n^{1+c}}{c \log n}$ space.

This allows us to compute the sum of $c\log n$ elements at a time. To see this, consider the second term in the previous sum $\sum_{j=k'f(n)+1}^{k} A[j]\cdot B[i+j-1]$, which we summed one term at a time. Let $h_1 = \lfloor \frac{k'f(n)+1}{c\log n} \rfloor + 1$ and $h_2 = \lfloor \frac{k}{c\log n} \rfloor$. Then we have $k'f(n)+1 < h_1 c\log n < (h_1+1)c\log n < \cdots < h_2 c\log n \leq k$. Create the masks $M_{h_1-1} = ..000B[k'f(n)+1, h_1 c\log n - 1]$, $M_{h_1} = B[h_1 c\log n, (h_1+1)c\log n - 1], \ldots, M_{h_2} = B[h_2 c\log n, k]000..$, where M_{h_1-1} and M_{h_2} are padded with 0s so that their lengths are $c\log n$. We can then write the sum as

$$\sum_{j=k'f(n)+1}^{k} A[j]\cdot B[i+j-1] = \sum_{i=h_1-1}^{h_2} D'[M_i, ic\log n]$$

Thus the time to sum the remainder would be $O(\frac{f(n)}{c\log n})$.

We now show how to build these data structures. Computing all $D[i,k] = \sum_{j=1}^{kf(n)+1} A[j]\cdot B[i+j-1]$, where $i \in [m - kf(n)]$, $1 \leq k \leq \frac{n}{f(n)}$ can be done in $O(\frac{mn}{f(n)}\log f(n))$ time. The computation is done in two phases. First, for every $1 \leq k \leq \frac{n}{f(n)}$, we consider the subarray $A[(k-1)f(n)+1 \ldots kf(n)]$, we call it A_k for short. In the first stage of the computation, we wish to compute the following sum, for every $1 \leq k \leq \frac{n}{f(n)}$, and for every $1 \leq i \leq m - f(n) + 1$.

$$\sum_{j=1}^{f(n)} A_k[j]\cdot B[i+j-1]$$

Such sum is called a convolution of A_k and B as one can think of A_k slides over B. It is well known that performing the *generalized fast Fourier transform* (see for example, [8,14]) between each A_k and B computes such convolution. For short, we refer to such an algorithm as FFT. The FFT computes the masked sum of each A_k against every $f(n)$-length submask of B. Each of these FFTs is done in $O(m\log f(n))$ time. Thus, $O(\frac{nm}{f(n)}\log f(n))$ overall time is required. The second stage aggregates the sums of the smaller intervals for each offset in B. Let i be an offset in B. We know $\langle B[i\ldots i+f(n)-1], A_1\rangle, \langle B[i+f(n)\ldots i+2f(n)-1], A_2\rangle, \langle B[i+2f(n)\ldots i+3f(n)-1], A_3\rangle \ldots \langle B[i+n-f(n)\ldots i+n-1], A_{\frac{n}{f(n)}-1}\rangle$. Thus we partially sum the entries, i.e. for array of values P, we compute $\Sigma_{q=1}^{r} P[q]$, for each $1 \leq r \leq |P|$ in linear time. The first stage requires $O(\frac{nm}{f(n)}\log f(n))$ time and the second stage requires $O(\frac{nm}{f(n)})$ time. The preprocessing times given in the first half of this theorem thus follows.

Finally, as D' has $\frac{n^{1+c}}{c\log n}$ entries and each entry can be computed in $O(1)$ time, D' can be constructed in $O(\frac{n^{1+c}}{c\log n})$ time, and we obtain the preprocessing time in the second half of this theorem. To see this, fix a particular value of j and compute values for different masks by ascending number of 1s in the mask. Mask pattern M would then only introduce 1 new term over a previously computed mask pattern. $\square$

Finally we note that for $m = O(n)$, we may set $f(n) = \sqrt{n}\log n$ and $c = 1/2$ to obtain an $O(n^{3/2}/\log n)$-word data structure with $O(\sqrt{n})$ query time and $O(n^{3/2})$ preprocessing time.

3.2 A Conditional Lower Bound

We now give a conditional lower bound to show the hardness of this problem.

Theorem 2. *Let $p(n)$ and $q(n)$ respectively denote the preprocessing time and query time of a masked prefix sum data structure constructed over a bit vector mask of length n and an array of n bits. Then Boolean matrix multiplication over two $\sqrt{n/2} \times \sqrt{n/2}$ matrices can be solved in $p(n) + nq(n) + O(n)$ time.*

Proof. Let X and Y be two $\sqrt{n/2} \times \sqrt{n/2}$ Boolean matrices and $Z = X \times Y$. Let $x_{i,j}$, $y_{i,j}$ and $z_{i,j}$ denote the elements in row i and column j of matrices X, Y and Z, respectively. We then construct an array A of n bits and a bit mask B of length n as follows. A is obtained by storing the bits in X in row-major order in its first half, i.e., $A[(i-1)\sqrt{n/2} + j] = x_{i,j}$ for any $1 \le i, j \le \sqrt{n/2}$, and storing all 0s in its second half. The first $n/2$ bits of the mask B are also all 0s. Then we store the content of Y in $B[n/2 + 1..n]$ in column-major order, i.e., $B[n/2 + (j-1)\sqrt{n/2} + i] = y_{i,j}$ for any $1 \le i, j \le \sqrt{n/2}$.

To compute $z_{i,j}$ for any $1 \le i, j \le \sqrt{n/2}$, observe that the ith row of X form the content of $A[(i-1)\sqrt{n/2} + 1..i\sqrt{n/2}]$ while the jth column of Y is in $B[n/2 + (j-1)\sqrt{n/2} + 1..n/2 + j\sqrt{n/2}]$. It is then sufficient to answer the following two masked prefix sum queries: The first query uses $(i-1)\sqrt{n/2}$ and $n/2+(j-1)\sqrt{n/2}-(i-1)\sqrt{n/2}$ as the mask length and offset of B, respectively, while the second uses $i\sqrt{n/2}$ and $n/2 + (j-1)\sqrt{n/2} - (i-1)\sqrt{n/2}$. Note that by setting the offset of B to be $n/2+(j-1)\sqrt{n/2}-(i-1)\sqrt{n/2}$ in both queries, we ensure that $A[(i-1)\sqrt{n/2} + 1..i\sqrt{n/2}]$ (storing the ith row of X) is always masked by $B[n/2 + (j-1)\sqrt{n/2} + 1..n/2 + j\sqrt{n/2}]$ (storing the jth column of Y). Hence the answer to the second query subtracted by that to the first will give us the dot product of the ith row of X and the jth column of Y. Then, $z_{i,j} = 0$ if this product is 0, and $z_{i,j} = 1$ otherwise. Hence, we can compute Z by answering n masked prefix sum queries, and the theorem follows. $\square$

As the current best algebraic method of multiplying two $n \times n$ Boolean matrices has complexity $O(n^\omega)$ with $\omega < 2.3727$ [26], two $\sqrt{n/2} \times \sqrt{n/2}$ matrices can be multiplied in $O(n^{\omega/2})$ time. This implies that, with current knowledge, either the preprocessing time $p(n)$ must be $\Omega(n^{\omega/2}) = \Omega(n^{1.18635})$, or the query time $q(n)$ must be $\Omega(n^{\omega/2-1}) = \Omega(n^{0.18635})$. Furthermore, the running time of the best known combinatorial approach for multiplying two $n \times n$ Boolean matrices is only polylogarithmically faster than cubic [2,6,27]. Hence, by purely combinatorial methods with the current best knowledge, either the preprocessing time $p(n)$ must be $\Omega(n^{3/2})$ or the query time $q(n)$ must be $\Omega(\sqrt{n})$, save for polylogarithmic speed-ups. On the other hand, the specific trade-off given in Theorem 1 gives a data structure with $O(\sqrt{n})$ query time and $O(n^{3/2})$ preprocessing time, for any $m = O(n)$. Hence, it can be used to multiply two $\sqrt{n} \times \sqrt{n}$ Boolean matrices in $O(n^{3/2})$ time, matching the time required for the best known combinatorial algorithm for Boolean matrix multiplication within polylogarithmic factors.

3.3 Dynamic Masked Prefix Sum

In dynamic settings, we support the update to any entry of A or B by assigning a new value to it. The following theorem presents our result.

Theorem 3. *Given a bit vector B of length m and an array A of length n, there is a data structure that uses $O(\frac{mn}{f(n)} + m + n)$ words of space that can answer masked prefix sum queries in $O(f(n) + g(n))$ time and support updates in $O(\frac{mn \log f(n)}{g(n)f(n)} + g(n))$ time, for any functions $f(n)$ and $g(n)$ with $0 < f(n) < n$ and $0 < g(n) < m + n$.*

Alternatively, for any $c > 0$, there is a data structure that uses $O(\frac{mn}{f(n)} + \frac{n^{1+c}}{c \log n} + m + n)$ words of space and can answer masked prefix sum queries in $O(\frac{f(n)}{c \log n} + g(n))$ time and support updates in $O(\frac{mn \log f(n)}{g(n)f(n)} + \frac{n^{1+c}}{g(n)} + g(n))$ time. If $m = O(n)$, setting $f(n) = n^{2/3} \log n$, $g(n) = n^{2/3}$ and $c = 1/3$ yields an $O(n^{4/3}/\log n)$-word data structure with $O(n^{2/3})$ query and update times.

For the full proof, see Appendix A.

3.4 Approximate Masked Prefix Sum

To achieve faster query time and decrease the space cost, we consider the problem of building a data structure to answer the masked prefix sum problem approximately.

Theorem 4. *Given a bit vector B of length m and an array A of length n, there is a data structure that uses $O((m \log n)/\varepsilon)$ words of space and can answer $(1+\varepsilon)$-approximate masked prefix sum queries in $O(\min\{\log\log n, \sqrt{\log(\log n/\varepsilon)}\})$ time for any $\varepsilon \in (0, 1]$.*

Proof. We first consider the approximate prefix sum solution on just the integer vector A. Consider the n prefix sums $S[j] = \sum_{k=1}^{j} A[k]$. We build a mapping P such that $j \in P$ if $(1+\varepsilon)^i \leq S[j] < (1+\varepsilon)^{i+1}$ for some i and $S[j-1] < (1+\varepsilon)^i$. That is, whenever the prefix sum reaches a power of $(1+\varepsilon)$, we write down the index where the prefix sum reaches it. Furthermore, for these indices, we write down the actual prefix sum, so that $P[j] = S[j]$. We may store this mapping in linear space with a hash table. To answer the approximate query for an index ℓ, we find the predecessor of ℓ in P and report the prefix sum at the predecessor.

By construction, if the predecessor of ℓ is j and $(1+\varepsilon)^i \leq S[j] < (1+\varepsilon)^{i+1}$, then $S[\ell] < (1+\varepsilon)^{i+1}$, as otherwise there would be a predecessor of ℓ where the prefix sum reaches $(1+\varepsilon)^{i+1}$. Similarly, $S[\ell] \geq (1+\varepsilon)^i$. Thus our output gives a $(1+\varepsilon)$-approximation to the actual result.

We note that the universe of the integers for the predecessor problem is $U = n$, and the number of elements in the set is at most $N = \log_{(1+\varepsilon)} n$. Since $0 < \epsilon \leq 1$, by Taylor series expansion, $\log(1+\varepsilon) = \Theta(\varepsilon)$. Hence, $N = O((\log n)/\varepsilon)$. Thus by Lemma 1 the predecessor data structure takes $O((\log n)/\varepsilon))$ words of space. The query time is $O(\min\{\log\log n, \sqrt{\log(\log n/\varepsilon)}\})$.

We now apply this solution to solve the masked prefix sum problem approximately, by building the above data structure for each index i of B on the integer vector A_i defined as $A_i[j] = A[j] \cdot B[i+j-1]$ (we mask the integer vector by the length n bit vector obtained from B starting at index i). This gives a solution of $O((m \log n)/\varepsilon)$ words of space. $\square$

3.5 Parallel Algorithms

We now consider a related problem which is to design a parallel algorithm to answer all queries of an instance of the masked prefix sum problem in the PRAM model.

Lemma 2. *Let A be the array of numbers of length n and let B be the bit vector of length m in the masked prefix sum problem. Then there is an optimal span (parallel running time) and work parallel algorithm that stores explicitly the answers of all mn queries in $O(\log n + \log m)$ span and performs $\Theta(mn)$ work in the PRAM model. In the implicit model, the work can be improved to be $O(\frac{mn}{\log n})$.*

The proof will appear in the journal version.

4 Data Structures for Sparse Internal Inner Product

In this section, we study the `SparseMaskedPrefixInnerProduct` problem, in both static (Sect. 4.1) and dynamic (Sect. 4.2) settings.

4.1 Static Sparse Internal Inner Product

We first present conditional lower bounds for the sparse internal inner product problem, SparseIIP for short, by giving a reduction from the SetDisjointness problem, which is defined as follows.

Definition 3 (SetDisjointness Problem). *Preprocess a family F of m sets, all from universe U, with total size $n = \bigcup_{S \in F} |S|$ so that given two query sets $S, S' \in F$ one can determine if $S \cap S' = \emptyset$.*

The following conjecture addresses the hardness of this problem.

Conjecture 1 (SetDisjointness *Conjecture* [19]). Any data structure for the SetDisjointness problem with constant query time must use $\tilde{\Omega}(n^{2-\varepsilon})$ space, while any data structure for this same problem that uses $O(n)$ space must have $\tilde{\Omega}(n^{1/2-\varepsilon})$ query time, where ε is an arbitrary small positive constant. This conjecture is true unless the 3SUM conjecture is false. Where $\tilde{\Omega}(f(n))$ means $\Omega(\frac{f(n)}{polylog(n)})$.

Recently, a stronger conjecture was proposed. A matching upper bound exists for Conjecture 2 by generalizing the ideas from [9,18].

Conjecture 2 (*Strong* SetDisjointness *Conjecture* [12]). Any data structure for the SetDisjointness problem that answers the query in x time must use $S = \tilde{\Omega}(\frac{n^2}{x^2})$ space for any $x \in (0, n]$.

We now show our reduction from SetDisjointness to SparseIIP to prove the following conditional lower bound:

Theorem 5. *Unless the* SetDisjointness *Conjecture is false, any data structure for the* `SparseMaskedPrefixInnerProduct` *problem with constant query time must use* $\tilde{\Omega}(n^{2-\varepsilon})$ *space, while any data structure for this same problem that uses* $O(n)$ *space must have* $\tilde{\Omega}(n^{1/2-\varepsilon})$ *query time, where* ε *is an arbitrary small positive constant. Furthermore, unless the* Strong SetDisjointness Conjecture *is false, any data structure for the* `SparseMaskedPrefixInnerProduct` *problem with query time* x *must use* $\tilde{\Omega}(\frac{n^2}{x^2})$ *space for any* $x \in (0, n]$.

Proof. Let $U = \{1, \ldots, u\} \subseteq \mathbb{N}$, where $u = |U|$. Let $F = \{S_1, \ldots, S_m\}$, such that $n = \bigcup_{S_i \in F} |S_i|$. Each $S_i \in F$ is a set represented by a sparse bit vector B_i of size u, where $B_i[j] = 1$ if and only if $j \in S_i$. Let $A = B = B_1 \cdot B_2 \cdots B_m$, i.e. the concatenation of all B_i one after another. It is easy to see that A and B have n ones. We treat them as sparse bit vectors. A and B are the inputs to the `SparseMaskedPrefixInnerProduct`.

The query $S_i \cap S_j$ for any $i, j \in [m]$ with $i < j$ is done by performing two prefix sum queries. The first has offset $(j-i) \cdot u + 1$ and length $(i-1) \cdot u$, while the second has offset $(j-i) \cdot u + 1$ and length $(i) \cdot u$. We subtract the first query result from the second query result and check if the result is zero or not. The answer is zero if and only if $S_i \cap S_j = \emptyset$.

A and B are represented using the predecessor data structure. Thus, the reduction takes time linear in the number of ones. $\square$

We now design a quadratic space data structure with polylogarithmic query time. Thus it matches the conditional lower bounds proved under the Strong SetDisjointness Conjecture within a polylogarithmic factor in query time.

Theorem 6. *Let* A *and* B *be two sparse* U*-bit vectors, and let* n *represent the sum of the number of* 1*s in* A *and the number of* 1*s in* B*. There is a data structure that uses* $O(n^2)$ *words of space that can answer a* `SparseMaskedPrefixInnerProduct` *query in* $O(\min\{\log\log U, \sqrt{\log n}\})$ *time for any* $p \in [1, n]$.

Proof. For each position k of B, create the vector A_k as we defined in Sect. 3.4, with $A_k[j] = A[j] \cdot B[j+k]$. Since there are at most n 1s in A and at most n 1s in B, the positions of the 1 bits in A can only form at most n^2 pairs with the positions of the 1 bits in B. Therefore, all these bit vectors, $A_1, A_2, \cdots, A_U$ have $O(n^2)$ 1 bits in total, where U is the length of A and B. If a bit vector A_k does not have any 1s, we do not store it at all. Otherwise, we represent A_k using Lemma 1 to answer predecessor queries, by viewing the position of each 1 bit as an element of a subset of $\{1, 2, \cdots, U\}$. We also augment this data structure by storing the rank of each element present in the subset, so that, given an index i, we can compute the number of 1s in $A_k[1..i]$ in $O(\min\{\log\log U, \sqrt{\log n}\})$ time. Since there are at most n^2 1s in all A_k's, these data structures use $O(n^2)$ space in total. We further build a perfect hash table T of $O(n^2)$ space to record which of these bit vectors have at least a 1, and for each such bit vector, a pointer to its predecessor data structure.

With these data structures, we can answer a query as follows: Suppose we need to compute the inner product of the first j bits of A against the j-length submask of B starting at position k. Then we check whether A_k has at least a 1 bit using T. If it does not, we return 0. Otherwise, we find the predecessor of j in A_k and return its rank as the answer, which requires $O(\min\{\log\log U, \sqrt{\log n}\})$ time. Thus, we have an $O(n^2)$-space data structure with $O(\min\{\log\log U, \sqrt{\log n}\})$ query time. $\square$

4.2 Dynamic Sparse Internal Inner Product

In this section, we assume that the sparse bit vectors support updates. That is, we support the operation update(V, i, x) which sets the vector V (which is either A or B) at position $1 \le i \le |V|$ to value $x \in \{0, 1\}$. We prove conditional lower bounds and show tight upper bounds up to polylogarithmic factors.

Definition 4 (3SUM *[22]***).** *Let A, B, and C three sets of numbers in $[-n^3, n^3]$, where $|A| + |B| + |C| = n$. The goal is to determine whether there is a triple $a \in A, b \in B, c \in C$ such that $a + b = c$.*

The 3SUM conjecture claims that it is not possible to solve the 3SUM problem in $O(n^{2-\varepsilon})$ time, for any $\varepsilon > 0$. It is believed that even when relaxing the range to be $[-n^2, n^2]$ the problem has the same lower bound. It was shown that even if one can preprocess A or B (but not both [13,20]) the lower bound holds [21]. It follows that the lower bound that we will prove holds for the case in which updates are allowed in only one of the bit vectors.

Lemma 3. *Unless the 3SUM conjecture is false, the* SparseIIP *problem in the dynamic setting must have at least query or update in $\Omega(n^{1-\varepsilon})$ time, for $\varepsilon > 0$.*

Proof. We initialize two empty bit vectors A' and C' of length $N = 2n^3 + 1$ corresponding to the range $[-n^3, n^3]$. In order to handle negative number, we begin by setting bits in A' to 1s at positions $a + n^3 + 1$, for any $a \in A$. Similarly, we set $C[c + n^3 + 1]$ to 1 for any $c \in C$. For each $b \in B$, we perform an internal query in the following way. We ask for b as the offset and $N - b$ as the length of the inner product. If the inner product is not zero then we have a 3SUM triple. Finding such a triple $a + b = c$ is done by a binary search on the staring and ending positions of the interval until one triple is left. Note that b is known, thus, we only need to find the corresponding a and c. Overall the reduction uses $|A| + |C| = O(n)$ updates with $|B| = O(n \log n)$ queries. From the 3SUM conjecture we have that either query or update uses $O(n^{1-\varepsilon})$ time, for any $\varepsilon > 0$. $\square$

Lemma 4. *The lower bounds of the dynamic* SPARSEIIP *are tight up to polylogarithmic factors.*

The proof will appear in the journal version.

5 The Connections Between the Problems and the Internal Measurements

It follows directly from the definition, solving the internal prefix sums also solves the internal inner product problem. Thus, all the lower bound on the SPARSEIIP apply on the sparse internal prefix sum. Moreover, all upper bound algorithms for the internal prefix sums problem apply on the internal inner product problem. In this section, we emphasize the connection of these two problems to the internal measurements. The considered measurements are Hamming distance and Exact Matching with wildcards.

Definition 5 (INTERNALHAMMINGDISTANCE and INTERNALEMWW). *Let S, T be two strings of lengths n and m, respectively. The problem of* INTERNALHAMMINGDISTANCE *is to preprocess S and T to answer Hamming distance queries between any equal-length substrings of S and T, where Hamming distance counts the number of mismatches between the two substrings.*

Similarly, the problem of INTERNALEMWW *is to preprocess S and T to answer exact matching with wildcards queries between any equal-length substrings of S and T, where the query counts the number of mismatches between the two substrings. However, mismatches with wildcards are not counted.*

Lemma 5. *Assume a constant-size alphabet Σ. Then, there is a linear-time reductions from the* INTERNALHAMMINGDISTANCE *to the internal inner product problem, and vice versa. Moreover, there is a linear-time reductions from the* INTERNALEMWW *problem to the internal inner product problem, and vice versa.*

For the full proof, see Appendix B.

A Details Omitted from Sect. 3

Proof of Theorem 3. Given a bit vector B of length m and an array A of length n, there is a data structure that uses $O(\frac{mn}{f(n)} + m + n)$ words of space that can answer masked prefix sum queries in $O(f(n) + g(n))$ time and support updates in $O(\frac{mn \log f(n)}{g(n) f(n)} + g(n))$ time, for any functions $f(n)$ and $g(n)$ with $0 < f(n) < n$ and $0 < g(n) < m + n$.

Alternatively, for any $c > 0$, there is a data structure that uses $O(\frac{mn}{f(n)} + \frac{n^{1+c}}{c \log n} + m + n)$ words of space and can answer masked prefix sum queries in $O(\frac{f(n)}{c \log n} + g(n))$ time and support updates in $O(\frac{mn \log f(n)}{g(n) f(n)} + \frac{n^{1+c}}{g(n)} + g(n))$ time. If $m = O(n)$, setting $f(n) = n^{2/3} \log n$, $g(n) = n^{2/3}$ and $c = 1/3$ yields an $O(n^{4/3}/\log n)$-word data structure with $O(n^{2/3})$ query and update times.

Proof. We first present a data structure with amortized bounds on update operations. The main idea is to rebuild the data structures from Theorem 1 every $g(n)$ updates. Since Theorem 1 presents multiple trade-offs, in the rest of the proof, we use $s(m, n)$, $p(m, n)$ and $q(n)$ to represent the space cost, preprocessing time and query time of the data structures in that theorem. Before a rebuilding is triggered, we maintain two copies of the array and the bit mask: A and B store the current content of this array and the bit mask, respectively, while A' and B' store their content when the previous rebuilding happened. Thus, the data structure, D, constructed in the previous rebuilding, can be used to answer masked prefix sum queries over A' and B'. For the updates arrived after the previous rebuilding, we maintain two lists: a list L_A that stores a sorted list of the indexes of the entries of A that have been updated since the previous rebuilding, and a list L_B that stores a sorted list of the indexes of the entries of B that have been updated since the previous rebuilding. Since the length of either list is at most $g(n) < m + n$, all the data structures occupy $O(s(m, n) + m + n)$ words.

We then answer a masked prefix sum query as follows. Let k and i be the parameters of the query, i.e., we aim at computing $\sum_{j=1}^{k} A[j] \cdot B[i + j - 1]$. We first perform such a query using D in $q(n)$ time and get what the answer would be if there had been no updates since the last rebuilding. Since both L_A and L_B are sorted, we can walk through them to compute the indexes of the elements of A that have either been updated since the last rebuilding, or it is mapped by the query to a bit in B that has been updated since the last rebuilding. This uses $O(g(n))$ time. Then, for each such index d, we consult A, A', B and B' to compute how much the update, to either $A[d]$ or $B[d + i - 1]$, affects the answer to the query compared to the answer given by D. This again requires $O(g(n))$ time over all these indexes. This entire process then answers a query in $O(q(n) + g(n))$ time.

For each update, it requires $O(1)$ time to keep A and B up-to-date. It also requires an update to the sorted list L_A or L_B, which can be done in $O(g(n))$ time. Finally, since the rebuilding requires $O(p(m, n))$ time and it is done every $g(n)$ updates, the amortized cost of each update is then $O(p(m, n)/g(n) + g(n))$.

The bounds in this theorem thus follows from the specific bounds on $s(m, n)$, $p(m, n)$ and $q(n)$ in Theorem 1.

Finally, to deamortize using the global rebuilding approach, instead of rebuilding this data structure entirely during the update operation that triggers the rebuilding, we rebuild it over the next $g(n)$ updates. This requires us to create two additional lists L'_A and L'_B: Each time a rebuilding starts, we rename L_A and L_B to L'_A and L'_B, and create new empty lists L_A and L_B to maintain indexes of the updates that arrive after the rebuilding starts. To answer a query, we cannot use the data structure that is currently being rebuilt since it is not complete, but we use the previous version of it and consult L_A, L_B, L'_A and L'_B to compute the answer using ideas similar to those described in previous paragraphs. $\square$

B Details Omitted from Sect. 5

Proof of Lemma 5. Assume a constant-size alphabet Σ. Then, there is a linear-time reductions from the InternalHammingDistance to the internal inner product problem, and vice versa. Moreover, there is a linear-time reductions from the InternalEMWW problem to the internal inner product problem, and vice versa.

Proof. **The reduction from the InternalHammingDistance to the internal inner product**. For each letter $\sigma \in \Sigma$, we change S and T to be bit vectors: σ in T become 1 and $\Sigma \setminus \{\sigma\}$ become 0, while in S, σ become 0 and $\Sigma \setminus \{\sigma\}$ become 1. That is, the Hamming distance query sums a constant number of internal inner products in order to answer the query.
The reduction from the internal inner product problem to the InternalHammingDistance. Assume we have two bit vectors A and B. Every 1 in A is transferred to 001, and 0 to 010, while in B, each 1 is transferred to 001, and 0 to 100. Let S and T be the transformed strings from A and B, respectively. It is easy to see that only 1 against 1 in A against B causes 0 mismatches between the corresponding substrings of S and T and any of the other 3 combinations results in 2 mismatches. Where corresponding substrings means that the starting and ending positions of the substrings are chosen to fit the original query, i.e. by multiplying the query indices by 3. Note that this reduction transfers the internal inner product to the InternalEMWW, as well.
The reduction from the InternalEMWW problem to the internal inner product. In a similar way, the inner product solves the exact matching with wildcards problem. We repeat the same process as described previously for Hamming distance but this time, wildcards are always transferred to 0 in both S and T. It is easy to see that when the sum over all the inner products is 0, there is an exact match with wildcards.

□

References

1. Andersson, A.: Faster deterministic sorting and searching in linear space. In 37th Annual Symposium on Foundations of Computer Science, FOCS 1996, Burlington, Vermont, USA, 14–16 October 1996, pp. 135–141. IEEE Computer Society (1996)
2. Bansal, N., Williams, R.: Regularity lemmas and combinatorial algorithms. Theory Comput. **8**(1), 69–94 (2012)
3. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. **65**(1), 38–72 (2002)
4. Bille, P., et al.: Dynamic relative compression, dynamic partial sums, and substring concatenation. Algorithmica **80**(11), 3207–3224 (2017). https://doi.org/10.1007/s00453-017-0380-7
5. Blelloch Guy, E.: Prefix sums and their applications. In: Synthesis of Parallel Algorithms, vol. 1, pp. 35–60. M. Kaufmann (1993)
6. Chan, T.M.: Speeding up the four Russians algorithm by about one more logarithmic factor. In: SODA, pp. 212–217 (2015)

7. Clifford, R., Grønlund, A., Larsen, K.G., Starikovskaya, T.: Upper and lower bounds for dynamic data structures on strings. In: Niedermeier, R., Vallée, B. (eds.) 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, 28 February–3 March 2018, Caen, France, vol. 96, pp. 22:1–22:14. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
8. Clifford, R., Iliopoulos, C.S.: Approximate string matching for music analysis. Soft. Comput. **8**(9), 597–603 (2004). https://doi.org/10.1007/s00500-004-0384-5
9. Cohen, H., Porat, E.: Fast set intersection and two-patterns matching. Theor. Comput. Sci. **411**(40–42), 3795–3800 (2010)
10. Dhulipala, L., Blelloch, G.E., Shun, J.: Theoretically efficient parallel graph algorithms can be fast and scalable. ACM Trans. Parallel Comput. **8**(1), 1–70 (2021)
11. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**(3), 424–436 (1993)
12. Goldstein, I., Lewenstein, M., Porat, E.: On the hardness of set disjointness and set intersection with bounded universe. In: Lu, P., Zhang, G. (eds.) 30th International Symposium on Algorithms and Computation (ISAAC 2019), 8–11 December 2019, Shanghai University of Finance and Economics, Shanghai, China, vol. 149, pp. 7:1–7:22. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
13. Golovnev, A., Guo, S., Horel, T., Park, S., Vaikuntanathan, V.: Data structures meet cryptography: 3SUM with preprocessing. In: Makarychev, K., Makarychev, Y., Tulsiani, M., Kamath, G., Chuzhoy, J. (eds.) Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, 22–26 June 2020, pp. 294–307. ACM (2020)
14. Kalai, A.: Efficient pattern-matching with don't cares. In: Eppstein, D. (ed.) Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 6–8 January 2002, San Francisco, CA, USA, pp. 655–656. ACM/SIAM (2002)
15. Keller, O., Kopelowitz, T., Feibish, S.L., Lewenstein, M.: Generalized substring compression. Theor. Comput. Sci. **525**, 42–54 (2014)
16. Kociumaka, T.: Efficient data structures for internal queries in texts. PhD Thesis. University of Warsaw (2019)
17. Kociumaka, T., Radoszewski, J., Rytter, W., Walen, T.: Internal pattern matching queries in a text and applications. In: Indyk, P. (ed.) Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, 4–6 January 2015, pp. 532–551. SIAM (2015)
18. Kopelowitz, T., Pettie, S., Porat, E.: Dynamic set intersection. In: Dehne, F., Sack, J.-R., Stege, U. (eds.) WADS 2015. LNCS, vol. 9214, pp. 470–481. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21840-3_39
19. Kopelowitz, T., Pettie, S., Porat, E.: Higher lower bounds from the 3SUM conjecture. In: Krauthgamer, R. (ed.) Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, 10–12 January 2016, pp. 1272–1287. SIAM (2016)
20. Kopelowitz, T., Porat, E.: The strong 3SUM-INDEXING conjecture is false. arXiv preprint arXiv:1907.11206 (2019)
21. Green Larsen, K.: Personal communication
22. Patrascu, M.: Towards polynomial lower bounds for dynamic problems. In: Schulman, L.J. (ed.) Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5–8 June 2010, pp. 603–610. ACM (2010)

23. Patrascu, M., Demaine, E.D.: Tight bounds for the partial-sums problem. In: Ian Munro, J. (ed.) Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, 11–14 January 2004, pp. 20–29. SIAM (2004)
24. Pibiri, G.E., Venturini, R.: Practical trade-offs for the prefix-sum problem. Softw. Pract. Exp. **51**(5), 921–949 (2021)
25. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space θ(N). Inf. Process. Lett. **17**(2), 81–84 (1983)
26. Williams, V.V.: Multiplying matrices faster than Coppersmith-Winograd. In: STOC, pp. 887–898 (2012)
27. Huacheng, Yu.: An improved combinatorial algorithm for Boolean matrix multiplication. Inf. Comput. **261**, 240–247 (2018)