On the Streaming Model Augmented with a Sorting Primitive

Gagan Aggarwal* Stanford University
Mayur Datar† Google
Sridhar Rajagopalan‡ IBM Almaden
Matthias Ruhl§ Google

Abstract

The need to deal with massive data sets in many practical applications has led to a growing interest in computational models appropriate for large inputs. The most important quality of a realistic model is that it can be efficiently implemented across a wide range of platforms and operating systems.

In this paper, we study the computational model that results if the streaming model is augmented with a sorting primitive. We argue that this model is highly practical, and that a wide range of important problems can be efficiently solved in this (relatively weak) model. Examples are undirected connectivity, minimum spanning trees, and red-blue line segment intersection, among others. This suggests that using more powerful, harder to implement models may not always be justified.

Our main technical contribution is to show a hardness result for the "streaming and sorting" model, which demonstrates that the main limitation of this model is that it can only access one data stream at a time. Since our model is strong enough to solve "pointer chasing" problems, the communication complexity based techniques commonly used in showing lower bounds for the streaming model cannot be adapted to our model. We therefore have to employ new techniques to obtain these results.

Finally, we compare our model to a popular restriction of external memory algorithms that access their data mostly sequentially.

1. Introduction

Recently, massive data sets have appeared in an increasing number of application areas. The sheer size of this data, often in the order of terabytes, means that "polynomially computable" is no longer synonymous with "efficiently computable"; in fact, any problem that requires significantly super-linear computation time is practically impossible to solve on these inputs. In this paper, we address the question of "what is efficiently computable on massive data sets".

The main bottleneck for massive data set computations on modern computing hardware is the cost for I/O operations, and usually not the cost for in-memory computations. It is well-known that modern computing hardware is optimized for sequential access to data, and there are substantial penalties for non-sequential data access, manifested for example as seek times, cache misses, and pipeline stalls. This has, in the past, led to an interest in the streaming model of computation [20, 5, 16], and practical applications built on the streaming primitive [18, 10].

It is really hard to write code that accesses massive data sets non-sequentially without a substantial degradation in performance compared to sequential access. Usually, this code would have to be platform and operating system specific. Consequently, there are very few general-purpose primitives available that access data in a non-local fashion, and still obtain maximal throughput from the I/O subsystem. Of these, "sorting" is the most readily available and most researched primitive [3, 6, 26, 11].

Consequently, streaming computations with an added sorting primitive are a natural and efficiently implementable class of massive data set computations. In this paper, we study this computational model and present results of two kinds. First, we establish that many natural problems solvable for massive data sets can already be solved in our model. This indicates that it is not necessary to consider more powerful models, as these do not seem to enable the solution of more problems, but run the risk of being not efficiently implementable. Second, we show hardness results for our model, i.e. demonstrate that there are clear limitations to the computational power of this class.

1.1. Our Contributions

First, we formally define the "streaming and sorting" model in section 2. We then demonstrate in section 3 that "streaming and sorting" admits efficient solutions to a number of natural problems, such as undirected connectivity,

* Supported in part by a Stanford Graduate Fellowship and NSF Grant EIA-0137761. E-Mail: gagan@cs.stanford.edu
† E-Mail: datar@cs.stanford.edu
‡ E-Mail: sridhar@almaden.ibm.com
§ E-Mail: ruhl@google.com

minimum spanning trees, suffix array construction, and even some geometric problems. Moreover, all problems solvable in NC with a linear number of processors can also be solved in our model. These problems are all known to be hard in the streaming model, suggesting that the addition of a sorting operation extends that model in a meaningful way.

The fact that most of the problems studied under more powerful models (e.g. [1]) can be solved in our model demonstrates that the additional power of those models is not strictly necessary, and that by studying the "weaker" class of "streaming and sorting", we do not lose any real computational power, while gaining a simpler and provably efficient model.

In section 4 we turn to hardness results for the "streaming and sorting" model, asking the question what *cannot* be computed in our computational model. As far as we are aware, there has been no past work on hardness results on computational models that are strict extensions of the streaming model. Our results might, therefore, be the first of their kind.

Intuitively, a problem is solvable in our model if access to data can be split into several linear passes over the data, with a known pattern of data reordering (sorting) between successive passes. A natural candidate for a problem that is hard for our model is therefore a problem that requires a non-predictable order of data access. One such candidate is "pointer chasing", in particular since it has a high communication complexity (see e.g. [21]). Surprisingly, the k-round pointer chasing problem can be solved in $O(\log k)$ passes in the "streaming and sorting" model, and is therefore not a "hard" problem for this model. This demonstrates that communication complexity techniques used to show lower bounds in the streaming model (see e.g. [7]) cannot be adapted to show hardness results for our model.

In the main technical result of this paper, we introduce a new problem called "alternating sequence" (a variant of pointer chasing), and show its hardness for the "streaming and sorting" model. To this end, we develop a new lower-bounding technique based on purely combinatorial arguments.

In section 5, we discuss the relationship between the streaming and sorting model and another model previously discussed in the literature, which we name "linear external memory algorithms" (LEMA). Intuitively, this model is equivalent to a streaming and sorting model enhanced with the ability to concurrently access multiple streams and not just one. It turns out that the LEMA model is strictly more powerful than streaming and sorting (without simultaneous access to multiple streams); in particular, it can efficiently solve the "alternating sequence" problem.

We conclude the paper in section 6 by discussing further research directions and open problems.

1.2. Related Work

The "streaming model" was defined implicitly in the work of Munro and Paterson [20], and even earlier, in the context of algorithms for systems with tape-based storage and little memory. The growing interest in massive data set computations has led to numerous publications on this topic in recent years; a comprehensive survey of this area is beyond the scope of this introduction.

The streaming primitive has been studied in the past few years in the graphics community (see [10, 22, 17, 18], which also list further references), and this application area recently also received some interest in the theory community [15, 2].

Borodin, Nielsen and Rackoff [9] study a computational model in which the data is accessed sequentially, but the as-yet-unread part of the input can be re-sorted depending on the already read input. This way of combining streaming and sorting is much more powerful than our approach, e.g. the problem ALTERNATING SEQUENCE from section 4.1 that is hard in our model is easy in theirs. However, their model was not meant for computations on massive data sets, and in fact does not seem practical in that setting.

The other computational models for massive data set computations studied in the literature are mostly restrictions of the "external memory algorithms" (EMA) model introduced by Aggarwal and Vitter [4, 24]. Many of these models (such as the one by Abello et al [1]) inherit the potential to access data in a non-sequential fashion, making them potentially harder to implement efficiently in practice.

The "mostly-sequential" EMA model studied in section 5 has been considered by several researchers [13, 14, 8], who emphasized that it is a particularly efficient subclass of EMAs.

2. Streaming and Sorting

We will first define the notion of a "streaming and sorting" algorithm. A *stream* is a sequence $\mathcal{S} = x_1 x_2 \ldots x_n$ of items $x_i \in \Sigma$, where Σ is some problem-specific universe. A *memory m streaming pass* is a function computed by a Turing machine M with a local memory of m bits that reads an input stream $\mathcal{S}$ and writes an output stream $Str_M(\mathcal{S})$. The machine M is allowed to move only left-to-right on both streams, i.e. it can only read $\mathcal{S}$ in a single linear pass, and it can only append to the output string, never erase what it has already written. A *memory m sorting pass* is a function defined using a Turing machine M with memory m that computes a partial order on Σ, i.e. given two items of Σ, it returns which one is greater, or whether they are equal or incomparable. If $\mathcal{S}$ is the input of a sorting pass, then the output $Sort_M(\mathcal{S})$ is $\mathcal{S}$ reordered according to the partial ordering defined by M. Note that the result is not uniquely defined if there are incomparable items in $\mathcal{S}$; in practice, this problem is easy to avoid. Also note that M's computation in the sorting pass is side-effect-free, i.e. no state is maintained between comparisons.

Definition 1 (StrSort)
We let $\mathbf{StrSort}(p_{\mathrm{Str}}, p_{\mathrm{Sort}}, m)$ *be the class of functions computable by the composition of up to* p_{Str} *streaming passes and* p_{Sort} *sorting passes, each with memory* m*, where we assume that*

- *the local memory is maintained between streaming passes, and*
- *streams produced at intermediate stages are of length* $O(n)$*, where* n *is the length of the input stream.*

We set $\mathbf{StrSort}(p,m) := \cup_{p'+p''\leq p}\mathbf{StrSort}(p', p'', m)$. $\square$

Note that $\mathbf{StrSort}(p,0,m)$ is not streaming as it is usually defined, since customarily the p passes are performed on the input stream, without ever writing any intermediate streams. However, one can easily see that by constructing the streams only implicitly, $\mathbf{StrSort}(p,0,m)$ can be simulated by a p-pass streaming algorithm with memory $m \cdot p$ that does not produce any intermediate streams.

3. Algorithms

As stated in the introduction, adding the sorting primitive greatly enhances the computational power of the streaming model, and many natural problems are efficiently solvable in the **StrSort**-model. By "efficiently", we mean "using little memory and few passes". In particular we are interested in algorithms that use poly-logarithmic memory and a poly-logarithmic number of passes. For this purpose, we define $\mathbf{PL\text{-}StrSort} := \cup_k \mathbf{StrSort}(O(\log^k n), O(\log^k n))$.

Clearly, tasks like computing the median, or computing frequency moments, while hard for the traditional streaming model, are trivial in **PL-StrSort**, requiring only a single sorting pass to order the input elements. We will now discuss some non-trivial examples of problems solvable in **PL-StrSort**, but not in the traditional streaming model.

3.1. Undirected Connectivity

First, we give a **PL-StrSort**-algorithm for undirected s-t-connectivity. For this problem we assume that the input stream consists of edges (u,v) of an undirected graph and two distinguished vertices s and t, and the question is whether there is a path from s to t in the graph.

Lemma 2
Undirected s-t*-connectivity can be solved in randomized* $\mathbf{StrSort}(O(\log n), O(\log n))$.

Proof: Our algorithm is very similar to the one commonly used in **RNC** to solve this problem. The algorithm proceeds as follows. We assign a random number, a $3\log n$-bit integer, to each vertex of the graph. Then each vertex gets labeled by the smallest number among the ones assigned to its neighbors and itself. We then merge all vertices that receive the same label. In expectation, this reduces the number of nodes in the graph by a constant factor. Thus, by repeating this process a logarithmic number of times, we obtain a graph without any edges, where each vertex represents a connected component of the original graph. By keeping track of which intermediate vertices s and t get merged into, we can answer the connectivity query by simply checking whether they end up in the same component.

The remainder of the proof consists of proving that the number of nodes decreases by a constant factor in each contraction phase, and that each such phase can be implemented by an algorithm in $\mathbf{StrSort}(O(1), O(\log n))$.

Correctness. We have to show that in expectation the number of nodes decreases by a constant factor in each contraction phase.

Suppose the number of nodes before a relabeling phase is n. Since with high probability all randomly assigned numbers are distinct, we can assume without loss of generality that the assigned numbers are actually $\{1,2,\ldots,n\}$, since only their relative order matters for the relabeling. Further, we can assume that there are no nodes with out-degree 0 in the graph, because if either s or t are mapped to such a node, we can decide the connectivity question immediately, and otherwise these nodes can simply be ignored.

A node v can get assigned a label greater than $n/2$ only if both itself and its neighbors get assigned random numbers greater than $n/2$. Since this only happens with probability $1/2$ per node, and v has at least one neighbor, v gets a label greater than $n/2$ with probability at most $1/4$. Thus, the expected number of nodes that receive a label greater than $n/2$ is only $n/4$. The expected total number of distinct labels in the relabeled graph is therefore at most $\frac{3}{4}n$ (assuming the worst case that all labels less than $n/2$ are actually used). This shows the desired reduction in size.

Implementation. Let us now sketch the implementation of a contraction phase. We assume that our stream is the list $\mathcal{V}$ of the vertices $v_1, v_2, \ldots, v_n$, followed by the list $\mathcal{E}$ of the vertex-pairs representing the edges. Since the graph is undirected, $\mathcal{E}$ contains both pairs (u,v) and (v,u) for an edge between u and v.

While both $\mathcal{V}$ and $\mathcal{E}$ are part of one stream, we will sometimes state operations on the two halves independently, assuming that in such a pass, the other half of the stream is passed through unchanged.

1. First, in a pass over $\mathcal{V}$, we produce a stream $\mathcal{R}$ that contains pairs of node names v_i and random distinct

numbers r_i, e.g. $3\log n$-bit random numbers:

$$(v_1,r_1)(v_2,r_2)(v_3,r_3)\dots(v_n,r_n)$$

2. Now we determine the new label for each node, i.e. the lowest value of r_i among its neighbors and itself.
   (a) First, we sort the pair of streams $\mathcal{R}$ and $\mathcal{E}$, so that the pairs (v_i,r_i) are directly followed by all edges whose first component is v_i:

   $$(v_1,r_1)(v_1,\_)(v_1,\_)\dots(v_2,r_2)(v_2,\_)\dots$$

   In one pass on this stream, we can produce a new stream of edges $\mathcal{E}'$, where for the first component of each edge, v_i is replaced by the corresponding r_i:

   $$(r_1,\_)(r_1,\_)\dots(r_2,\_)\dots$$

   (b) Now we sort $\mathcal{R}$ and $\mathcal{E}'$, so that the pairs (v_i,r_i) appear right before all edges whose second component is equal to v_i:

   $$(v_1,r_1)(\_,v_1)(\_,v_1)\dots(v_2,r_2)(\_,v_2)\dots$$

   In this stream, the number r_i assigned to a node v_i occurs right before the list of numbers assigned to the nodes incident to v_i. Thus, in one linear pass, we can determine the smallest number among them for each v_i, which yields a stream $\mathcal{L}$ of nodes v_i with their new labels ℓ_i:

   $$(v_1,\ell_1)(v_2,\ell_2)(v_3,\ell_3)\dots(v_n,\ell_n)$$

3. Now that we know the correct labels, all that remains is to relabel the edges in $\mathcal{E}$. This can be done by repeating step 2(a) for both the first and second component of $\mathcal{E}$, using $\mathcal{L}$ instead of $\mathcal{R}$. While producing this new stream, we can also eliminate all edges of the form (ℓ,ℓ), yielding a new stream of edges $\mathcal{E}_{new}$.

   The new list of vertices can be obtained from $\mathcal{L}$ by outputting only the second component ℓ_i of each pair, sorting the result, and removing duplicates. □

3.2. Minimum Spanning Tree

Using the above algorithm for undirected connectivity as a subroutine, it is not hard to compute minimum spanning trees in the “streaming and sorting” model.

We are going to use a divide and conquer approach to compute a minimum spanning tree of a graph $G=(V,E)$. The algorithm is as follows.

1. Sort the edges $E=\{e_1,e_2,\dots,e_m\}$ by increasing weight.
2. Let $E_0=\{e_1,e_2,\dots,e_{m/2}\}$ be the “lighter” half of the edges.
3. Compute the connected components of (V,E_0).
4. We now recursively compute minimum spanning trees for
   (a) each connected component in (V,E_0), and
   (b) the graph (V',E') where V' is the set of connected components in (V,E_0), and E' contains the edges in $E\setminus E_0$ connecting pairs of components.

It is not hard to see that the union of the edges in the individual spanning trees yields the answer to the minimum spanning tree problem for G.

In the **StrSort**-implementation, the current stream always contains a concatenation of all sub-problems currently being considered. During a divide step, the edges corresponding to the newly created sub-problems might appear in an arbitrary order in the input stream. By applying a sorting pass, we can rearrange these edges so that edges belonging to the same sub-problem appear consecutively.

The algorithm is similar to Kruskal’s algorithm, but reorders computations in a way that is compatible with streaming computations. The sorting passes are used to rearrange the data so that they appear in the correct order for subsequent streaming passes.

3.3. Red-Blue Line Intersection

We now consider a geometrical problem, motivated by geometric range queries, called RED-BLUE-INTERSECTION. The input is a list of red and blue line segments in the plane. The red line segments are parallel to the x-axis, the blue segments parallel to the y-axis. The output is the number of intersection points between red and blue line segments. (In many applications, one is actually interested in a list of intersection points, but that might require an output greater than the input, which our streaming model does not allow.)

Lemma 3
RED-BLUE-INTERSECTION *can be computed in deterministic* **StrSort**$(O(\log n),O(\log n))$.

Proof: We assume that no endpoint of a blue line segment lies on a red line segment. This assumption simplifies the presentation, but can be removed by standard techniques.

Let the *slab* of a line segment $(x_1,y_1)-(x_2,y_2)$ (with $x_1<x_2$) be the vertical strip $[x_1,x_2]\times\mathbb{R}$ of the plane. Then the following claim follows by a simple case analysis.

Claim 4
The number of intersections of red and blue line segments is equal to $L-U$, where

- *$L :=$ the number of pairs of red line segments s and lower end-points of blue line segments p, such that p is in the slab of s and* below *s, and*

- *U := the same for upper end-points of blue line segments.* □

The streaming algorithm computes the two values L and U and outputs their difference. For symmetry reasons, it is sufficient to show how to compute L. For this, we create a stream of all red line segments and blue lower endpoints. We use a divide-and-conquer approach, also known as distributional sweep [24].

The basic idea of the approach is the following. The input is divided into m vertical slabs $[x_i, x_{i+1}] \times \mathbb{R}$ where $-\infty = x_0 < x_1 < \cdots < x_m = +\infty$, such that each slab contains roughly the same number of blue lower endpoints. This division can be easily accomplished by a sort operation. Now we sort all these points by increasing y-coordinate. As we process this stream, for each of the m slabs, we keep track of the number of blue lower endpoints seen so far. When we encounter a red line segment, then for each slab that it crosses completely, we add the number of blue lower endpoints seen in that slab to our running sum. The (possibly) two ends of a segment that do not completely cross a slab are recursively passed down to the subproblem in the slab it is in. We then recursively solve the m slab problems. In $\log_m n$ passes, this finds all possible intersections, proving Lemma 3. □

We note that a generalization of the above algorithm can be used to solve the red-blue-intersection problem for arbitrary orientations of red and blue line segments in **StrSort**$(O(\log^2 n), O(\log n))$. More details will be in the full version of this paper.

3.4. Simulation of Circuits

Many other problems can be solved in **PL-StrSort** because they also admit algorithms that access data in a series of linear passes, interleaved with a known pattern of data reordering. Examples are substring matching, suffix array computations, undirected graph connectivity, computing maximal independent sets, finding a minimum cut in a undirected graph and many other problems (see [23] for a more comprehensive overview).

Some of these results are consequences of the fact that in our computational model, it is quite straightforward to evaluate uniform linear width, poly-logarithmic depth circuits. Since problems in **NC** that require only a linear number of processors can be solved by such circuits, this gives us a systematic way of constructing streaming algorithms for these problems. Examples of such problems are the undirected connectivity algorithm mentioned above, and the computation of a maximal independent set [19].

Lemma 5
A uniform bounded fan-in circuits with width $O(n)$ and depth $d(n)$ can be evaluated in deterministic **StrSort** *using $d(n)$ streaming and sorting passes, and $O(\log n)$ memory.*

Proof Sketch: This Lemma can be shown similar to PRAM simulations in the external memory model [12]. To evaluate a circuit, we inductively generate for each level ℓ of the circuit a stream $\mathcal{S}_\ell$ that contains a list of the inputs taken by the circuit nodes on that level, ordered by node. (Note that $\mathcal{S}_1$ can easily be computed from the input.) One streaming pass on $\mathcal{S}_\ell$ can compute the outputs of all nodes on level ℓ. To go from these outputs to $\mathcal{S}_{\ell+1}$, all we have to do is rearrange them according to the input pattern of the next level. This can be done by labeling the outputs with the numbers of the gates that take them as inputs (and creating duplicates if an output is input to multiple gates). Sorting on these labels yields the desired order. □

4. Hardness

The "streaming and sorting" model is clearly much more powerful than the traditional streaming model, as evidenced by the algorithms mentioned above. So it is only natural to ask what *cannot* be computed in this model.

Intuitively, problems hard for the streaming and sorting model would require data access in unpredictable patterns, i.e. the data accesses cannot be rearranged in a poly-logarithmic number of fixed-order passes over the data. At first glance, a good candidate for a hard problem might therefore be "pointer-chasing". In this problem, one is given an array of n numbers in the range $\{1, \ldots, n\}$. If the i-th array element is j, then we say that the i-th element points to the j-th element. Given a number k, we start at the first element of the array, and then repeatedly move from the current element to the element it points to. The output is the element we reach after k steps.

However, while the pointer-chasing problem seems to require data access in arbitrary patterns, and is indeed very hard from a communication complexity point of view [21], it can be solved in **PL-StrSort**. This is because it is possible to square a graph in the **StrSort** model in a constant number of passes, i.e. we can compute the result of following pointers 2,4,8,... times. This allows us to solve the pointer-chasing problem in $O(\log k)$ passes in the **StrSort** model.

4.1. ALTERNATING SEQUENCE

We now state a problem that builds on the pointer-chasing paradigm, but while remaining conceptually simple, is provably intractable in the **StrSort** model. The problem is called ALTERNATING SEQUENCE and is defined as follows.

Input: A stream of pairs
$(a_1,a'_1)(a_2,a'_2)\dots(a_n,a'_n)(b_1,b'_1)(b_2,b'_2)\dots(b_n,b'_n)$.

Output: The sequence $a_{i_1}b_{j_1}a_{i_2}b_{j_2}a_{i_3}b_{j_3}\dots$, satisfying

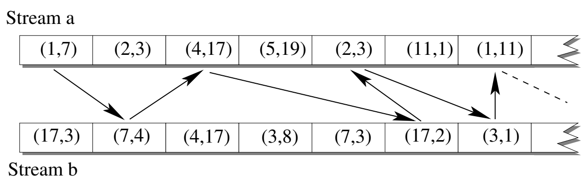
(i) $i_1 = 1$

(ii) $i_k = \min\{i > i_{k-1} \mid a_i = b'_{j_{k-1}}\}$ for $k \geq 2$

(iii) $j_k = \min\{j > j_{k-1} \mid b_j = a'_{i_k}\}$ for $k \geq 1$, using $j_0 = 0$.

The sequence ends as soon as either $i_k = n$, $j_k = n$ or the minima in equations (ii) or (iii) do not exist.

This problem is best explained by an example. Consider the following sequence, where the (a_i,a'_i)-pairs are in the top stream, and the (b_j,b'_j)-pairs are in the bottom stream. For this example, the output would start with 1,7,4,17,2,3,1,11,....



Theorem 6
The problem ALTERNATING SEQUENCE *can be solved in* **StrSort**$(O((n/m)^{1/2}),m)$. *However, it cannot be solved in deterministic* **StrSort**(p,m) *unless* $pm = \Omega(n^{1/3})$. *We assume that elements are indivisible, and that only comparisons between elements are allowed.*

4.2. Upper bound

Proof: We first give an algorithm to compute ALTERNATING SEQUENCE in **StrSort**$(O((n/m)^{1/2}),m)$. The algorithm proceeds in $2(n/m)^{1/2}$ phases. In the first phase, we construct a stream containing $(n/m)^{1/2}$ copies of the sequence

$$(*)\quad A_1A_2\dots A_{\sqrt{nm}}B_1B_2\dots B_{\sqrt{nm}},$$

where A_i and B_i are short for (a_i,a'_i) and (b_i,b'_i), respectively (we will continue to use this notation for the remainder of this proof). Clearly, the stream has length $O(n)$, and can be constructed by outputting elements multiple times, indexed by their desired position on the tape, followed by a sorting pass.

We can use this stream to output the beginning of the answer upto the point where either of the indices i_k or j_k becomes greater than $\sqrt{nm}$. This is done as follows. When reading the first sequence of A's, we keep $A_1,A_2,\dots,A_m$ in memory. This enables us to construct the answer upto $i_k \leq m$, $j_k \leq \sqrt{nm}$ on the following sequence of B's. When we exhaust the A's in our memory, we continue to the next stretch of A's, and put $A_{m+1},\dots,A_{2m}$ in memory, use that with the following stretch of B's, and so on.

Since we process m of A-elements for each copy of $(*)$, after reading the whole stream, the A-elements have been processed upto index $m\cdot(n/m)^{1/2} = \sqrt{nm}$. This shows that we will make a progress of $\sqrt{nm}$ on one of the two indices.

In the second (and later) phases, we repeat the same pattern, but the A- and B-subsequences of length $\sqrt{nm}$ start where we left off in the previous phase. Since one of the indices advances by $\sqrt{nm}$ in each phase, we will have constructed the whole output in at most $2n/\sqrt{nm} = O((n/m)^{1/2})$ phases, which yields the claimed number of passes.

4.3. Lower bound

We now show the harder part of the Theorem, the lower bound of $p = \Omega(n^{1/3}/m)$ passes for the **StrSort** model. We show this bound by fixing an arbitrary algorithm, and then adversarially choosing its input such that it cannot output the correct solution unless the number of passes meets the claimed bound. In this adversarial model we only decide on the equality of two items when the algorithm compares them.

The proof consists of two main parts. First, we show that in a single pass, we cannot make too much progress towards the solution of the problem. This is because whatever $O(n)$-size input stream we use in that pass, there will be a roughly $\sqrt{n/m}$-length alternating sequence that we cannot output based on linearly scanning the stream.

In the second part of our proof, we apply this construction to a sequence of passes, which slightly weakens the $\sqrt{n/m}$-bound as the algorithm gains more information about the processed data. In the end, it yields the $\Omega(n^{1/3}/m)$ lower bound on the number of passes.

The key lemma for the first half of the proof is the following. It shows that a string that contains all possible ALTERNATING SEQUENCE answers as subsequences must be very long (at least n^2+1 symbols). The converse of this statement is what we will need: a string of length $O(n)$ can only contain all ALTERNATIVE SEQUENCE answers for instances of length $O(\sqrt{n})$. Here and in the following, we will use A_i and B_j to stand for (a_i,a'_i) and (b_j,b'_j), respectively.

Lemma 7
Let S be the set of alternating increasing sequences of the alphabet $\Sigma = \{A_1,\dots,A_n,B_1,\dots,B_n\}$, i.e. strings of the form $[A_{i_1}]B_{j_1}A_{i_2}B_{j_2}\dots A_{i_k}[B_{j_k}]$ where $i_\ell < i_{\ell+1}$ and $j_\ell < j_{\ell+1}$ for $1 \leq \ell < k$, and at most the very last symbol is equal to A_n or B_n. By $[\dots]$ we mean that the symbol is optional. Let $\mathfrak{s}$ be a Σ-string that contains all strings in S as subsequences (i.e. the elements of each $s \in S$ occur in order in $\mathfrak{s}$, but not necessarily consecutively). Then $\mathfrak{s}$ has length at least n^2+1.
$\square$

Although we will not need this later, it is interesting enough to note that the bound in the lemma is actually tight, i.e. there are strings of length n^2+1 of the desired form. Let $\mathfrak{a}_i$ to be the string $A_iA_{i+1}\dots A_{n-1}$ and $\mathfrak{b}_i$ be the string $B_iB_{i+1}\dots B_{n-1}$. Then the following string of length n^2+1 has the desired properties:

$$\mathfrak{b}_1\mathfrak{a}_1\mathfrak{b}_1\mathfrak{a}_2\mathfrak{b}_2\mathfrak{a}_3\mathfrak{b}_3\mathfrak{a}_4\mathfrak{b}_4\dots\mathfrak{a}_{n-1}\mathfrak{b}_{n-1}A_nB_n$$

We leave the details to the reader, and concentrate on the lower bound.

Proof (Lemma 7): Fix $\mathfrak{s}\in\Sigma^*$, and let $\mathfrak{s}_i$ be the suffix of $\mathfrak{s}$ that begins at position i (e.g. $\mathfrak{s}_1=\mathfrak{s}$). We will now define sets of strings S_i such that all strings in S_i have to appear as subsequences in $\mathfrak{s}_i$. We set $S_1 := S$, and define the other S_i inductively. If the i-th symbol of $\mathfrak{s}$ is A_k, then we set

$$S_{i+1}=\{s \mid \text{“}s \text{ does not start with } A_k\text{” and } (s\in S_i \vee A_ks\in S_i)\}.$$

Here “A_ks” stands for the string obtained by prepending A_k to s. Replacing A_k with B_k gives the definition in the case that the i-th symbol of $\mathfrak{s}$ is B_k.

It is not hard to see that for each i the strings in S_i necessarily have to be contained as subsequences in $\mathfrak{s}_i$. So to show the desired lower bound on the length of $\mathfrak{s}$, it suffices to show that $S_i\neq\emptyset$ for $i\leq n^2+1$.

Let us define predicates $\alpha(i,k,\ell)$ and $\beta(i,k,\ell)$ as “S_i contains a string with the substring A_kB_ℓ” and “S_i contains a string with the substring $B_\ell A_k$”, respectively. If at least one of these predicates is true, it implies that $S_i\neq\emptyset$. We will now show that going from i to $i+1$, not too many of the predicates change from true to false. The intuition behind this is that some predicates imply other predicates. For example $\alpha(i,k,\ell)$ implies $\beta(i,k',\ell)$ for all $k'>k$, since an alternating sequence which contains A_kB_ℓ can be continued as $A_kB_\ell A_{k'}$ for any $k'>k$. These implications limit the number of predicates that can become false. We will show the following.

Claim 8
Let k,k',ℓ,ℓ' be numbers between 1 and n with $k\neq k'$ or $\ell\neq\ell'$. If $\alpha(i,k,\ell)$, $\beta(i,k,\ell)$, $\alpha(i,k',\ell')$ and $\beta(i,k',\ell')$ are all true, then at least three of $\alpha(i+1,k,\ell)$, $\beta(i+1,k,\ell)$, $\alpha(i+1,k',\ell')$ and $\beta(i+1,k',\ell')$ are true.

Proof: Let us consider the case where the i-th element of $\mathfrak{s}$ is an A-element, say A_j (the “B-case” is similar). Then the β-predicates will be unchanged from i to $i+1$. And the α-predicates can only change if $j=k$ or $j=k'$. In fact, the only way that both these predicates could become false is if $j=k=k'$ holds. This implies $\ell\neq\ell'$, wlog $\ell<\ell'$. But the truth of $\beta(i,k,\ell)$ then implies that S_i and therefore S_{i+1} contain strings that contain $B_\ell A_kB_{\ell'}$, and thus $\alpha(i+1,k,\ell')$ remains true, which proves the claim. □

Thus, going from i to $i+1$ there will be at most one $\alpha(\dots,k,\ell)$, $\beta(\dots,k,\ell)$ pair for which one of the predicates becomes false, after both having been true so far. Since at the beginning, all n^2 such α,β-pairs are true, it takes at least n^2 elements of $\mathfrak{s}$ to hit all these pairs once. And for the last pair hit, it takes one more character to satisfy the remaining true predicate in it, so $\mathfrak{s}$ has to contain at least n^2+1 characters. □

Lemma 7 shows that is hard to interleave the two streams $A_1A_2\dots A_n$ and $B_1B_2\dots B_n$ into one stream such that all possible answers to ALTERNATING SEQUENCE appear as subsequences in the interleaved stream. For our application of streaming passes, we are however interested in the minimum stream length such that a memory m algorithm could produce all possible answers to ALTERNATING SEQUENCE. This is answered by the following corollary.

Corollary 9
Let $\mathfrak{s}$ be a string, such that all alternating sequences of the form stated in Lemma 7 can be output by a linear scan of $\mathfrak{s}$ by a machine of memory m. Then the length of $\mathfrak{s}$ is at least $(\frac{n}{m+1})^2+1$. □

Note that Corollary 9 is not tight, consider e.g. the case $m=\Omega(n)$, where the lower bound is just a constant, but clearly each A_i and each B_i has to appear at least once in $\mathfrak{s}$, giving a trivial lower bound of $2n$. But the corollary is still strong enough for our purposes.

Proof (Corollary 9): Group the elements of $A_1,A_2,\dots,A_n$ into $\frac{n}{m+1}$ groups of $m+1$ consecutive elements each (i.e. $A_1,\dots,A_{m+1}$ form the first group, $A_{m+2},\dots,A_{2m+2}$ the second, and so on), and do the same for the B's. By “interleaving an A-group with a B-group” we mean that we alternate the $m+1$ elements in the A-group in ascending order with the $m+1$ elements of the B-group, for example $A_1B_1A_2B_2\dots A_{m+1}B_{m+1}$ for the first two groups. Now consider only the strings in S that are concatenations of such interleaved groups.

Any memory m algorithm outputting these strings upon reading $\mathfrak{s}$ must for each interleaved group pair read at least one A-element and one B-element from $\mathfrak{s}$. This is because the groups have size $m+1$ each, so they could not possibly have been entirely in memory before outputting the interleaved group pair.

By restricting our view to “representatives of groups”, not distinguishing the individual elements of groups, the previous observation implies that $\mathfrak{s}$ must actually be an interleaved string in the sense of Lemma 7 on the group representatives. Since there are $\frac{n}{m+1}$ A- and B-groups each, Lemma 7 therefore implies that $\mathfrak{s}$ has to have length at least $(\frac{n}{m+1})^2+1$. □

For our adversarial argument, we now fix a particular streaming algorithm. We allow the algorithm to construct arbitrary input streams for each of its passes. Even in this more powerful model, we can still prove the lower bound.

To fix the input we will now inductively construct a sequence of numbers $1 = t_1 < t_2 < t_3 < \cdots < t_k = n$, and alternating sequences $s_1 \subset s_2 \subset s_3 \subset \cdots \subset s_k$ (where by $s \subset s'$ we mean that s is a prefix of s'), such that for all $1 \leq i < k$:

(i) s_k is the correct output to the ALTERNATING SEQUENCE problem,

(ii) all elements of s_i not in s_{i-1} are from the set $\{A_{t_i}, \ldots, A_{t_{i+1}-1}, B_{t_i}, \ldots, B_{t_{i+1}-1}\}$, and

(iii) s_i is chosen so that, even based on the comparisons the algorithm performed in the first $i-1$ passes, it is not possible for the algorithm to output the elements of $s_i \setminus s_{i-1}$ in order during pass i.

The last point implies in particular that the algorithm cannot output the correct solution within k passes.

For the first step of the inductive construction, we choose t_2 such that $(t_2/m+1)^2+1$ is greater than the length of the stream in pass 1. This means that t_2 can be chosen as $O(m\sqrt{n})$, where the constant depends only on the constant in the $O(n)$ bound we imposed on the maximal stream length. By Corollary 9 this implies that there will be an alternating sequence of the A_i and B_i with indices bounded by t_2 that cannot be output by a memory m algorithm upon reading the stream. We let s_1 be one such sequence.

For the inductive step, we have to modify our argument to account for the fact that the algorithm already has made some comparisons in the previous passes, and our choice of s_i must be consistent with them.

In the previous $i-1$ passes, the algorithm can have performed a total of $O((i-1)nm)$ comparisons – each element in the memory could have been compared to every element in the first $i-1$ streams. We are enforcing the policy that whenever a comparison during pass j involves an element with index greater than t_{j+1}, then we will always return "unequal". Also, we enforce that $a_i \neq a_j$, $a'_i \neq a'_j$, $b_i \neq b_j$ and $b'_i \neq b'_j$ for all $i \neq j$.

For pass i, we therefore want to choose t_{i+1} such that there is an alternating sequence $s_{i+1} \setminus s_i$ with indices between t_i and t_{i+1} not contained in a stream of length $O(n)$, and such that the sequence does *not consecutively contain* any of the $O((i-1)nm)$ pairs for which we already answered "unequal". If one excludes a set of ℓ (A,B)-pairs from appearing consecutively in the strings of S, then a simple modification of the proof of Lemma 7 yields a lower bound on $|\mathfrak{s}|$ of $n^2-\ell+1$.

So we have to choose t_{i+1} such that $((t_{i+1}-t_i)/(m+1))^2 - nm(i+1)+1 > O(n)$, which can be satisfied by $t_{i+1}-t_i = \Omega(\sqrt{nm^2+nm^3(i+1)})$, in particular by $t_{i+1}-t_i = \Omega(\sqrt{nm^3}\sqrt{i+1})$, where the constant only depends on the constant used for the $O(n)$ upper bound on stream lengths.

In summary, we can accomplish the selection of t_i's and s_i's as long as $t_{i+1}-t_i = \Omega(\sqrt{nm^3(i+1)})$ for all i. This gives the following upper bound on k:

$$\sum_{i=1}^{k} \Omega(\sqrt{nm^3 i}) \leq n \implies \sum_{i=1}^{k} \sqrt{i} = O(\sqrt{n/m^3})$$

Since $\sum_{i=1}^{k} \sqrt{i} = O(k^{3/2})$, this means the construction is possible for k up to $(\sqrt{n/m^3})^{2/3} = n^{1/3}/m$, which concludes the proof of Theorem 6. □

4.4. A decision version of ALTERNATING SEQUENCE

A polynomial separation analogous to Theorem 6 can also be shown using the following decision problem: the input is just like ALTERNATING SEQUENCE, but now every pair (a_i, a'_i) and (b_j, b'_j) has a *color* which is either red or blue. The desired answer is whether the last element of the output of ALTERNATING SEQUENCE is a red or a blue element.

We briefly sketch how the computation of the ALTERNATING SEQUENCE function can be reduced in **StrSort** to this decision problem. We will do this with a factor $O(\log^2 n)$ increase in the number of required passes. Thus, the hardness of the function implies the hardness of the decision problem.

So suppose we have an oracle for the decision problem (a bit of care shows that each invocation of this oracle can be replaced by a corresponding **StrSort** algorithm in the following). First, this allows us to find the last element of the ALTERNATING SEQUENCE output (and not just its color) in $O(\log n)$ streaming passes. This is easily done using binary search: color half the input pairs red and half the pairs blue, and invoke the oracle. This shows us in which half of the pairs the last element is. We then recurse on this half (color it half red/half blue), and so on. After $O(\log n)$ applications of the oracle, we know the identity of the last element.

The algorithm producing the last element of the ALTERNATING SEQUENCE output can be used to give a divide-and-conquer solution to ALTERNATING SEQUENCE itself. For this, we consider the first half of the input streams $(a_1, a'_1)(a_2, a'_2)\ldots(a_{n/2}, a'_{n/2})$ and $(b_1, b'_1)(b_2, b'_2)\ldots(b_{n/2}, b'_{n/2})$. Given the above algorithm, we can compute the last two elements in the ALTERNATING SEQUENCE output for these half-problems. (The second-to-last element can be computed by first computing the last element, and then deleting it and the part of the stream following it from the input, and again computing the last element.)

The knowledge of these elements "in the middle" of the output allows us to split the input streams in half, and construct solutions for both halves independently. Note that in each divide step, at least one of the streams gets split exactly in half, showing that only $O(\log n)$ recursions are necessary. Thus, ALTERNATING SEQUENCE can be reduced to the red/blue decision problem, while multiplying the number of passes by $O(\log^2 n)$.

5. Linear Access External Memory Algorithms

5.1. Definitions

In this section, we will elucidate the relationship of the **StrSort** model to the well-studied external memory model. External memory algorithms (EMAs) study the effect that block-oriented external data storage has on the efficiency of algorithms (see [24] for a recent survey). An external memory algorithm can access the external storage (disk) only in units of blocks (each containing B items), and performance is measured in terms of the total number of disk accesses (reads and writes). Thus, a good data storage scheme and exploitation of locality are necessary for efficient algorithms. The *parallel disk model* introduced by Vitter and Shriver [25] has the following parameters:

$N =$ problem size (in data items),

$M =$ internal memory (in data items),

$B =$ block transfer size (in data items), and

$D =$ number of independent disk drives.

In this paper we are only interested in the single processor case, so we will ignore the (sometimes studied) additional parameter of the number of processors.

As mentioned in the introduction, reading data items sequentially from a disk achieves a substantially higher throughput than performing random accesses. Thus, external memory algorithms that only perform sequential reads or writes are a particularly efficient subclass of EMAs. We therefore define a linear access EMA (LEMA) as an EMA with the following parameter:

$P =$ number of out-of-sequence reads/writes on the disks ("passes")

A read or write operation is considered "out-of-sequence" if it does not act on the data block following the one last read or written on the corresponding disk. We are interested in algorithms for which P and M are small, i.e. poly-logarithmic in N. The class of these problems is denoted by **PL-LEMA**.

5.2. Relation to StrSort

While LEMA is somewhat similar to the **StrSort** model (in that the data is read and written sequentially), the **PL-LEMA** model turns out to be strictly more powerful than the **PL-StrSort** model. Since we can sort in **LEMA** using a logarithmic number of passes, we can simulate any algorithm for the **PL-StrSort** model in the **PL-LEMA** model while increasing the number of passes by at most a logarithmic factor. On the other hand, the ALTERNATING SEQUENCE problem which needs at least a polynomial number of passes in **StrSort** (as shown in Theorem 6) can be solved very efficiently (using only two passes) in **PL-LEMA**. The picture in section 4.1 suggests how to solve it using concurrent access to two disks. First, one divides the input onto two disks, one containing the a-pairs, and the other the b-pairs. Then, we begin by reading a'_1, and scanning the second disk until a matching b_j is found. Then we scan the first disk to find an a_i matching b'_j, and so on, alternating between the two disks. In this sense, the solution makes maximal use of the fact that it can read the two disks independently.

6. Conclusion

In this paper, we have studied a model for massive data set computations – which we think can be efficiently implemented – by extending the streaming model with a sorting primitive. This model allows for the efficient solution of a variety of natural problems known to be hard for the streaming model, but computable using stronger models which are harder to implement in practice. These problems include undirected connectivity, MST, red-blue line intersections, and several others. We have also proved one of the first hardness results for models of massive data set computations that are strict extensions of the streaming model. The communication complexity based techniques which were previously used in showing lower bounds for the streaming model do not generalize to our model, and consequently, new methods had to be developed.

Since this computational class is a quite natural extension of the traditional streaming model, and efficiently implementable in practice, it deserves further study. For example, can a breadth first or depth first traversal of a graph be computed efficiently in this model? Can our lower-bound result be extended to randomized algorithms or the bit model, where operations other than comparisons are allowed? And are there more natural problems that are hard to solve in this model, yet can be realistically solved on massive data sets? A candidate problem could be SUBSEQUENCE, in which given two strings, the problem is to decide whether one is a subsequence of the other.

We have also compared the "streaming and sorting" model to a model we called **LEMA**, which is a restriction of external memory algorithms to mostly linear accesses to external storage. We showed that the ability to read input from two disks simultaneously makes **LEMA** strictly more powerful than the "streaming and sorting" model. As mentioned above, it would be interesting to find more natural problems than ALTERNATING SEQUENCE that separate the two classes. It is also an interesting open problem whether increasing the number of independently accessible disks in **LEMA** always leads to a polynomial increase in computational power. We saw such an increase in the case of one versus two disks as demonstrated by ALTERNATING SEQUENCE.

Acknowledgments

We thank Jon Feldman for his help in proving Lemma 7, the anonymous reviewers for their comments, and David Karger, T.S. Jayram, Piotr Indyk, Ron Fagin, and the participants of the DIMACS Working Group on Streaming Data Analysis for helpful discussions.

References

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A Functional Approach to External Graph Algorithms. *Algorithmica*, 32:437–458, 2002.

[2] P. Agarwal, S. Krishnan, N. Mustafa, and S. Venkatasubramanian. Streaming Geometric Optimization Using Graphics Hardware. Technical Report TD-5HL2NX, AT&T, 2003.

[3] R. C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *Proceedings SIGMOD*, pages 240–246, 1996.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(8):1116–1127, 1988.

[5] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings SIGMOD*, pages 243–254, 1997.

[7] Z. Bar-Yossef, T. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proceedings FOCS*, 2002.

[8] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and Traversing: Maintaining Data for Traversals in a Memory Hierarchy. In *Proceedings ESA*, pages 139–151, Sept. 2002.

[9] A. Borodin, M. N. Nielsen, and C. Rackoff. (Incremental) Priority Algorithms. *Algorithmica*, 37(4):295–326, Sept. 2003.

[10] I. Buck and P. Hanrahan. Data Parallel Computation on Graphics Hardware. Manuscript, 2003.

[11] L. Chavet. FSORT, 2002. Available online at `www.fsort.com`.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-Memory Graph Algorithms. In *Proceedings SODA*, pages 139–149, 1995.

[13] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffix Tree Construction. In *Proceedings FOCS*, pages 174–183, Nov. 1998.

[14] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the Sorting-Complexity of Suffix Tree Construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[15] S. Guha, S. Krishnan, K. Mungala, and S. Venkatasubramanian. Application of the Two-Sided Depth Test to CSG Rendering. In *Proceedings of SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.

[16] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[17] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of SIGGRAPH*, pages 129–140, 2001.

[18] G. Humphreys, M. Houston, Y.-R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters. In *Proceedings of SIGGRAPH*, pages 693–702, 2002.

[19] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings STOC*, pages 1–10, 1985.

[20] J. I. Munro and M. S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980.

[21] S. J. Ponzio, J. Radhakrishnan, and S. Venkatesh. The communication complexity of pointer chasing. *Journal of Computer and System Sciences (JCSS)*, 62(2):323–355, 2001.

[22] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *Proceedings of SIGGRAPH*, pages 703–712, 2002.

[23] M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, Sept. 2003.

[24] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[25] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[26] J. Wyllie. SPsort: How to Sort a Terabyte Quickly. Technical report, IBM Almaden Research Center, 1999. Available online at `www.almaden.ibm.com/cs/gpfs-spsort.html`.