On Models for Massive Data Set Computations

Gagan Aggarwal  
Stanford

Mayur Datar  
Google

Sridhar Rajagopalan  
IBM Almaden

Matthias Ruhl*  
IBM Almaden

Abstract

Many models have been proposed that address the computation on massive data sets. Examples are the streaming model, external memory algorithms, and several variations on these models. A common theme among these models is the recognition that sequential access to data is faster than random access on modern computers. However, despite this common feature, the relationships between many of these models is neither obvious nor well understood.

This paper provides a first systematic study of the relative computational power of models for massive data set computations. We observe that the computational power of a model is characterized by both the amount of memory available and the number of streams/disks that can be read simultaneously. We show equivalence results for models that share these parameters, and separation results for models that do not.

We propose an attractive and practical new model based on this parameterization which unifies earlier models that were based on streaming primitives.

1 Introduction

Recently, there has been a lot of interest in massive data sets, since they are appearing in an increasing number of application areas. A natural question to ask in this context is what problems can be efficiently solved for large inputs. Clearly, for an input of several terabytes, “polynomially computable” is no longer synonymous with “efficiently computable.” In fact, any algorithm that requires significantly super-linear computation time will yield no results within practical time limits. And even linear-time algorithms can be inefficient in practice if they access the data in a random access fashion.

Several models [MP80, AMS99, HRR99, Vit01, FFM98, FCFM00, BCDFC02] have been proposed in the literature with the intent of giving meaning to the notion “efficiently solvable” for massive data sets. These include the streaming model, the external memory model and several variants. The models share a common characteristic – they all assume that sequential access to data is more efficient than random access.

It is therefore an important question to address what distinguishes these models from each other, and whether some of them are equivalent in terms of their computational power.

In the literature, the number of sequential passes over the data has been the relevant measure of efficiency. In this paper, we show that the power of “streaming” models of computation can be characterized in terms of two parameters,

(i) the amount of random access memory $m$ available, and

(ii) the maximum number $d$ of streams that can be read concurrently.

*Address: 650 Harry Rd K53/B2, San Jose, CA 95120, USA. Phone: +1-408-927-1406. Fax: +1-408-927-3030. E-Mail: ruhl@almaden.ibm.com
The first parameter, $m$ has been the focus of much of the past work. The second parameter, $d$ is much less well understood.

**Our Contributions.** We study computational models for massive data sets that access data in a stream fashion. We show that models that share the parameter $d$ are “equivalent”, and that models with $d = 1$ and $d > 1$ are “distinct.” By “equivalent” we mean that the required number of passes is within $O(\text{polylog} n)$ of each other (where $n$ is the length of the input stream), and by “distinct” we mean that there is a problem that requires polynomially more passes in the weaker model than in the stronger model. More precisely, we show the following hierarchy (see below for a discussion of the models).

$$\text{Classical Streaming} < \text{Streaming+Sorting} \approx \text{Tape}(1) \approx \text{LEMA}(1) < \text{Tape}(2) \approx \text{LEMA}(2)$$

$$\forall d : \text{Tape}(d) = \text{LEMA}(d)$$

It is surprisingly difficult to separate the models with $d = 2$ from the models with $d = 1$. This is due to two facts. First, the models with $d = 1$ are intuitively already very powerful. They allow for the efficient sorting of a stream, computing graph connectivity queries, minimum spanning trees, the minimum cut of a graph, finding substring matchings, and many other problems. Given this power, it is not clear that the reading of two independent streams cannot (for example) be simulated by an appropriate sequence of sorting operations.

From a technical point of view, the separation result is difficult, since the techniques commonly used to show hardness of streaming computations mostly come from communication complexity. Crucial for the application of these techniques, however, is the fact that the input stream is immutable. Being able to reorder the stream (almost) arbitrarily makes it impossible to apply similar arguments for our separation result.

In order to show the promised equivalences and separations, we first unify the models under a common framework, which we call “streaming circuits”. These are computational models that are characterized by the two parameters $m$ and $d$. In these models, streams are the basic objects that are manipulated, i.e. all operations take streams as inputs and produce streams as outputs. There are two kinds of operations allowed: first, two streams can be concatenated into one, and second, one can perform a streaming pass, where up to $d$ streams are read concurrently, and an output stream is written.

**Existing Computational Models.** We will now briefly discuss the previously proposed computational models for massive data sets, where the model does not allow random access to the data.

In the (classical) **Streaming Model** [MP80, AMS99, HRR99], the input is presented as a stream, i.e. a sequence of items. The algorithm is allowed one or several linear read-only passes over the stream, and the algorithm has access to a (usually small) local memory to perform computations. This model arises in practice for online problems where the input is generated in a sequential fashion and there is no desire or possibility of an intermediate data storage.

The **Streaming and Sorting** model also has the input given as a stream. An algorithm performs several passes. In each pass, the algorithm not only sequentially reads a stream, but also sequentially writes a stream. The stream written in pass $i$ is read in pass $i + 1$. In addition, instead of a streaming pass, the algorithm can sort the input stream according to “simple” comparison functions. This model is motivated by the observation that in practice, intermediate storage of streams is often possible, and that on modern computers, a stream can be sorted almost as fast as it can be read [Aga96, ADADC+97, Wyl99, Cha02].

**Linear Access External Memory Algorithms (LEMAs)** [FFM98, FCFM00, BCDFC02] are special cases of external memory algorithms [Vit01]. Here, an algorithm has access to a block-based external storage medium, e.g. a harddisk. That is, each read or write actually affects a whole block of data and
not just a single item. Several researchers have observed that external memory algorithms that access their data mostly sequentially tend to be particularly efficient in practice. In each pass of a linear access external memory algorithm (LEMA), data is read and written only sequentially from and to the disks.

Predating the other models are Tape-based Computations [Knu98]. Again, data is encountered as a stream. We can imagine the data being stored on tapes that can be read and written sequentially. The algorithm operates in passes. In each pass, one or more tapes are read sequentially, and one output tape is written. Although this model was motivated by hardware available in the early days of computing, it is true even for modern mass storage media that sequential access is faster than random access.

Related Work. The streaming model was defined implicitly in the work of Munro and Paterson [MP80], 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 paper.

Many results and references in the field of tape-based algorithms can be found in the third volume of Knuth’s “The Art of Computer Programming” [Knu98]. The computational powers of tape-based computations are very similar to the capabilities of graphics hardware for stream processing. These hardware architectures have been studied in the past few years in the graphics community (see [BH03, PBMH02], which also list further references), and have recently also received some interest in the theory community [GKMV03, AKMV03]. Our streaming circuits model is similar to the Chromium project [HEB+01, HHN+02] for distributed graphics rendering, where an input stream of OpenGL commands is processed by a network of nodes that communicate via streams.

The external memory algorithms (EMA) model was introduced by Aggarwal and Vitter [AV88] (for a recent survey on the topic see [Vit01]). Closely related variants of the linear access EMA model studied in this paper haven been considered by several researchers [FFM98, FCFM00, BCDFC02] to yield more practically meaningful analyses of algorithms.

Abello, Buchsbaum and Westbrook [ABW02] consider a subclass of single disk external memory algorithms. The algorithms given in their paper are built up from several primitives that can be viewed as functional transformations on streams. Restricted to the primitives actually used in the paper, which include streaming passes and sorting, their model is roughly equivalent to the “streaming and sorting” model discussed here. However, in principle, the model is not limited to this particular set of functional transformations, and its computational power will in general depend on what transformations are allowed.

Outline of the Paper. In section 2, we formally define the computational model of “streaming circuits”. In section 3, we discuss its relation to previously considered complexity classes, such as the streaming model, streaming and sorting, linear access external memory algorithms, and tape computations. In section 4, we show that the computational power of these models depends heavily on the number of concurrently accessible streams, and prove corresponding separation results. We conclude with a list of open problems in section 5.

2 Streaming Circuits

We begin our analysis of computational models for massive dataset computations by defining a model that captures the entire hierarchy of such models. This model is called streaming circuits. In section 3, we will show that the computational models that have previously been considered in the literature are in some sense special cases of the “streaming circuits” model.
Let $\Sigma$ be some alphabet. We call a sequence of symbols $S = x_1 x_2 \ldots x_n$, with $x_i \in \Sigma$, a stream. All operations in the streaming circuits model take one or more streams as inputs, and produce one stream as an output. These operations are the following (see also figure 1, page 18):

**Concatenation:** Taking two streams $S_1$ and $S_2$ as inputs, this operation outputs $S_1 S_2$, the concatenation of the two input streams.

**Streaming Pass:** Let $d \geq 1$ be an integer. Let $M$ be a RAM-machine with a local memory of $m$ bits. A $d$-way streaming pass is performed by $M$ by reading $d$ streams $S_1, S_2, \ldots, S_d$ sequentially, and outputting a new stream $S_M(S_1, S_2, \ldots, S_d)$.

In a streaming pass, the machine $M$ has to read the symbols in each $S_i$ sequentially, i.e. from beginning to end. The machine can, however, interleave the accesses to different streams in arbitrary ways, e.g. read some symbols from $S_1$, then some symbols from $S_4$, then again from $S_1$, etc. The output is also written in a sequential fashion, i.e. from beginning to end without “going back” to erase already written symbols.

A program in the streaming circuits model is defined by a circuit with the following properties (see figure 2 on page 18 for an example):

- the circuit has one input and one output,
- the circuit has constant width,
- the edges of the circuit represent streams, and the gates implement one of the operations defined above,
- each stream in the circuit has length $O(n)$, where $n$ is the length of the input stream, and
- all edges leaving a gate transmit the same stream.

The program, in the natural way, maps input streams (at the single input to the circuit) to output streams (the single output of the circuit). We will assume that the gates of a circuit are evaluated in some specific order (some specific topologically sorted order), and that the local memory of size $m$ is maintained between streaming passes.

**Definition 1 (StrC$(d,m,p)$, StrCS$(d,m,p)$)** Let $d \geq 1$ be an integer, and $m, p : \mathbb{N} \to \mathbb{N}$ monotonically increasing functions. Then StrC$(d,m,p)$ is the class of functions that can be computed with a streaming circuit that uses at most $p(n)$ gates, whose streaming passes use at most $m(n)$ bits of local memory and read at most $d$ streams concurrently for any input stream of length $n$.

We let StrCS$(d,m,p)$ be defined just like StrC$(d,m,p)$, except that the circuits do not contain any concatenation gates. □

Note that StrC$(d,m,p) = \text{StrCS}(d,m,p)$ for $d \geq 2$, since a concatenation is just a special case of a 2-way streaming pass.

**Practical considerations.** We note that both concatenation and streaming passes are operations that can be efficiently carried out on modern hardware. However, hardware limitations can impose restrictions on the in-degree of a streaming pass, i.e. the number $d$ of concurrently accessible streams. For example, if streams correspond to tapes, and a single tape reader is employed to perform streaming passes, then we have $d = 1$.

Requiring each stream to have length $O(n)$, and the circuit to have constant width limits the amount of intermediate storage required to evaluate the circuit to $O(n)$. 

4
3 Equivalences

In this section, we will show that the computational models discussed in the introduction are related to the streaming circuit models. The correspondences between the models will often not be exact, but we will say that two models $M_1$ and $M_2$ are equivalent (written as $M_1 \approx M_2$) if they can be simulated by each other with at most a $O(\text{polylog } n)$ factor increase in the number of passes.

The Streaming Model

**Definition 2 (Stream($m, p$))** Let $m, p : \mathbb{N} \rightarrow \mathbb{N}$ be monotonically increasing functions. Then Stream($m, p$) is the set of decision problems computable in the streaming model with $m(n)$ bits of memory and using $p(n)$ passes for an input stream of length $n$. □

Note that as opposed to streaming circuits, in the streaming model we do not allow the creation of intermediate streams, i.e. in each pass we read the input stream. One can argue that this model is somewhat unrealistic: since we obviously have to store the input stream to read it multiple times, why can’t we generate new streams as well? We are not aware of natural settings where this would be the case.

**Lemma 3 (Streaming Model)** Let $m, p : \mathbb{N} \rightarrow \mathbb{N}$ be monotonically increasing functions. Then, restricted to decision problems, we have

(a) Stream($m, 1$) = StrCS($1, m, 1$), and
(b) Stream($m, p$) ⊂ StrCS($1, m, p$) ⊂ Stream($mp, p$).

**Proof Sketch:** Part (a) follows from the definition. The first inclusion of part (b) is also immediate, by making the output of each gate equal to its input in the StrCS model. For the second inclusion, one can use a trick that is commonly used to compose space-bounded functions: instead of computing the intermediate streams explicitly, it suffices to recompute them as needed. We omit the details for space reasons. □

**Streaming and Sorting.** In the “streaming and sorting” model, we allow two kinds of passes that take a stream as input, and generate a stream as output. The first kind is a traditional streaming pass using $m(n)$ memory, reading a stream sequentially, and writing a stream at the same time. The second kind is a “sorting pass”. Given a comparison function computable with $m(n)$ memory, the input stream is re-sorted according to the comparison function. We assume that invocations of the comparison function are side-effect free, i.e. the local memory remains unaffected by a sorting pass.

**Definition 4 (StrSort($m, p$))** Let $m, p : \mathbb{N} \rightarrow \mathbb{N}$ be monotonically increasing functions. Then StrSort($m, p$) is the class of function problems computable with a “streaming and sorting” computation using a memory of at most $m(n)$ bits and at most $p(n)$ passes (counting both streaming and sorting passes) for inputs of length $n$. □

It is worth noting that the StrSort-model is a very powerful model that extends the power of the streaming model considerably. Being able to sort allows for trivial solutions of problems found to be hard for the streaming model, such as computing frequency moments, computing medians, $l_p$ norms of vectors in the cash-register model [Ind00], approximate V-OPT histograms [GKS01], and distinct element counting.

But repeated sorting, interleaved with streaming passes, also allows one to solve problems such as substring matching, suffix array computations, undirected graph connectivity, computing minimum spanning tress, maximal independent sets, finding a minimum cut in a undirected graph, red-blue line intersections and many other problems with $m, p = O(\text{polylog } n)$ (see e.g. [Ruh03] for an overview). Some of these results are consequences of the following Lemma.
Lemma 5  In \( \text{StrSort}(O(\log n), 2d) \), one can evaluate uniform bounded fan-in circuits that for \( n \) inputs have width \( O(n) \) and depth \( d(n) \). □

This can be shown similar to PRAM simulations in the external memory model [CGG’95, ABW02]. We sketch a proof in appendix A.1.

In fact, many of the algorithmic results for the \( \text{StrSort} \)-model are not “new”, in the sense that algorithms given for these problems in (possibly) more powerful models, such as the LEMA model, or the model of [ABW02], often do not really “need” advanced features of those models, and can already be implemented in the “streaming and sorting” model. This implies that the “streaming and sorting” model is really quite powerful, since there do not seem to be (m)any natural problems not efficiently solvable in it. We will now see how this model fits into the streaming circuit hierarchy.

Lemma 6 (Streaming and Sorting)  Let \( m, p : \mathbb{N} \to \mathbb{N} \) be monotonically increasing functions. Then we have

(a) \( \text{StrSort}(m, p) \subset \text{StrC}(1, O(m), O(p \log^2 n)) \), and

(b) \( \text{StrC}(1, m, p) \subset \text{StrSort}(m, p) \).

Proof Sketch: For part (a), it suffices to show that in \( \text{StrC}(1, m, O(\log^2 n)) \), one can sort a stream under comparison functions that can computed using \( m(n) \) bits. For space reasons we defer this result to appendix A.2.

To show part (b), we simulate a \( \text{StrC}(1, m, p) \) computation as follows in the \( \text{StrSort} \)-model. Consider the order in which the \( \text{StrC} \)-computation progresses. At each point, a constant number of streams will be “active” due to the constant width of the circuit. The stream in the \( i \)-th pass of the \( \text{StrSort} \)-computation is a concatenation of the active streams at the \( i \)-th pass of the \( \text{StrC} \)-computation. A concatenation in the \( \text{StrC} \)-model can be simulated by a sorting operation placing the two substreams next to each other, and a streaming pass can be simulated by a streaming pass that operates just on the affected stream, passing the other active streams through unchanged. □

Linear Access External Memory Algorithms (LEMAS).  External memory algorithms study the effect that block-oriented access to external data storage has on the efficiency of algorithms (see [Vit01] for a recent survey). An external memory algorithm (EMA) can access the external storage (think of it as a hard-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 arrangement scheme and exploitation of locality are necessary for efficient algorithms. The parallel disk model introduced by Vitter and Shriver [VS94] has the following parameters:

\[ N = \text{problem size (in data items)}, \]
\[ M = \text{internal memory (in data items)}, \]
\[ B = \text{block transfer size (in data items)}, \]
\[ D = \text{number of independent disk drives}. \]

It has been observed by many researchers (e.g. [FFM98, FCFM00, BCDFC02]) that external memory algorithms that access their data mostly sequential are particularly efficient, and deserve further study. We therefore define a linear access EMA (LEMA) as an EMA with the following additional parameter:

\[ P = \text{number of out-of-sequence reads/writes on the disks (“passes”).} \]
A read or write operation is considered “out-of-sequence” if it does not act on the data block following the block 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 $N$, and $D = O(1)$. More precisely, we define the model as follows.

**Definition 7 (LEMA*)** Let $m, p : \mathbb{N} \to \mathbb{N}$ be monotonically increasing functions. Then $\text{LEMA}(m, p)$ is the class of function problems computable with a LEMA with $M \cdot B = m(n)$, $P = p(n)$, and $D = O(1)$ for inputs of length $n$, where in each pass data is read from at most $d$ of the disks (there are no limitations for writing).

While exhibiting certain similarities to the streaming models considered earlier (in that data is read and written sequentially), the relationship between these classes is not entirely obvious. It is summarized by the following Lemma, whose proof is sketched in appendix A.3.

**Lemma 8 (LEMA)** Let $m, p : \mathbb{N} \to \mathbb{N}$ be monotonically increasing functions, and $d = O(1)$. Then we have

(a) $\text{LEMA}(d, m, p) \subset \text{StrC}(d, m, O(p))$, and

(b) $\text{StrC}(d, m, p) \subset \text{LEMA}(d, m, O(p))$. □

**Tape Computations.** Computations on tapes were studied extensively in computer science before the wide-spread availability of harddisks. Strictly speaking, there is no canonical model of “tape computations”. Variants allow the reading and writing of a tape in the same pass, or reading and writing a tape forwards as well as backwards.

But a common denominator is a model where a computation consists of a sequence of passes, and in each pass upto $d$ tapes are read concurrently, and one or more output tapes are written. The number $d$ naturally corresponds to the number of available tape readers. But even in the case $d = 1$, tapes can be concatenated to each other.

If the number of working tapes remains constant, when we have a simple correspondence between these tape computations, and the classes $\text{StrC}(d)$.

**Summary.** Let us restrict our attention to “efficient” algorithms, i.e. the case of $m, p = O(\text{polylog} n)$. In this setting, the above results imply the following equivalences (omitting the $m$ and $p$ parameters for clarity):

- $\text{StrCS}(1) \approx \text{Stream}$
- $\text{StrC}(1) \approx \text{StrSort}$
- $\text{StrC}(d) \approx \text{Tape}(d) \approx \text{LEMA}(d)$, for all $d \geq 1$

At this resolution, there are therefore only a small set of distinct models for massive data set computations.

## 4 Separations

While the previous section was concerned with showing equivalences between models, we are now interested in showing separation results. Our main result can be summarized as follows.

**Theorem 9** $\text{StrCS}(1) < \text{StrC}(1) < \text{StrC}(2)$. □

By $M_1 < M_2$ for two models $M_1$ and $M_2$ we mean that

(i) each problem solvable in $M_1$ can be solved in $M_2$ with the same memory and pass requirements, and

(ii) there is a decision problem that can be solved with $m, p = O(1)$ in $M_2$, but requires $mp = n^2(1)$ to be solved in $M_1$. 

7
Thus, the weaker model needs a polynomial number of passes whereas the stronger model needs only a constant number. By the results of the previous section, this immediately implies the following corollary.

**Corollary 10** Stream < StrSort < LEMA(2). □

**Proof (Theorem 9):** The first inequality is immediate from Lemmas 3 and 6 and the fact that e.g. computing high-order frequency moments exactly in the streaming model requires a polynomial memory (or a polynomial number of passes) (cf. [AMS99]). Naturally, this problem can be made into a decision problem by asking whether the frequency moment is at least some number.

For the second separation also, we first prove the separation for a function problem, and later reduce it to a decision problem. The function problem we consider is called ALTERNATING SEQUENCE. It is defined as follows.

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

**Output:** The sequence \(a_{i_1}b_{j_1}a_{i_2}b_{j_2}a_{i_3}b_{j_3}\ldots\), 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 sequences, 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 \(1,7,4,17,2,3,1,11,\ldots\)

The ALTERNATING SEQUENCE problem is conceptually easy to solve. The above figure suggests how the problem can be solved efficiently with concurrent access to two streams (i.e. in StrC(2)). First, one divides the input into two streams, one containing the \(a\)-pairs, and the other the \(b\)-pairs. Then, we begin by reading \(a'_1\), and scanning the second stream until a matching \(b_j\) is found. Then we scan the first stream to find an \(a_i\) matching \(b'_j\), and so on, alternating between the two streams. In this sense, the solution makes maximal use of the fact that it can read the two streams independently. We have just shown the following proposition.

**Proposition 11** The problem ALTERNATING SEQUENCE can be solved in StrC(2, O(1), O(1)). □

However, solving this problem requires a polynomial number of passes in the StrC(1) model.

**Theorem 12** The problem ALTERNATING SEQUENCE, where only comparisons of elements are allowed, can be solved in StrC(1, m, O((n/m)^{1/2}log n)). However, it cannot be solved in StrC(1, m, p) unless pm = \(\Omega(n^{1/3})\). □
Proposition 11 and Theorem 12 immediately imply Theorem 9, if we can give a decision problem to which we can reduce Alternating Sequence. This decision problem is the following: 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. For space reasons, we defer the reduction of the function problem to the decision problem to appendix B.1. This concludes the proof of Theorem 9. □

**Proof (Theorem 12):** We defer the algorithm for \(\text{StrC}(1, m, O((n/m)^{1/2} \log n))\) to appendix B.2, and concentrate on the (harder) lower bound of \(p = \Omega(n^{1/3}/m)\) passes for the \(\text{StrC}(1)\) 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 Alternating 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 13** Let \(S\) be the set of alternating increasing sequences of the alphabet \(\Sigma = \{A_1, \ldots, A_n, B_1, \ldots, B_n\}\), i.e. strings of the form \([A_{i_1}]B_{j_1}A_{i_2}B_{j_2} \cdots A_{i_k}B_{j_k}\) 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 \([\ldots]\) we mean that the symbol is optional. Let \(s\) be a \(\Sigma\)-string that contains all strings in \(S\) as subsequences (i.e. the elements of each \(s \in S\) occur in order in \(s\), but not necessarily consecutively). Then \(s\) has length at least \(n^2 + 1\). □

For space reasons, we defer the proof to appendix B.3. This lemma shows that it is hard to interleave the two streams \(A_1A_2 \ldots A_n\) and \(B_1B_2 \ldots B_n\) 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, whose proof is in appendix B.4.

**Corollary 14** Let \(s\) be a string, such that all alternating sequences of the form stated in Lemma 13 can be output by a linear scan of \(s\) by a machine of memory \(m\). Then the length of \(s\) is 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 \((\frac{t_2}{m + 1})^2 + 1\) 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 14 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 made \(O((i - 1)nm)\) comparisons in the first \(i - 1\) passes, and our choice of \(s_i\) must be consistent with them. This requires us to choose \(t_{i+1}\) such that \(t_{i+1} - t_i\) is somewhat larger than \(O(m\sqrt{n})\). For space reasons, we leave the technical details for appendix B.5. There, we show that we can choose an \(s_i\) as long as \(t_{i+1} - t_i = \Omega(\sqrt{nm^3(i+1)})\). This gives the following upper bound on \(k\):

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

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. \(\square\)

5 Conclusion

In this paper, we have given the first systematic comparison of several models for computations on massive data sets. We showed equivalences like the “streaming and sorting” model of computation being equivalent to linear access external memory algorithms that can read at most one disk at a time. On the other hand, increasing the number of disks to read from in the LEMA model increases the computational power. This result is somewhat surprising since the streaming and sorting model is already very powerful.

Our results are summarized by the hierarchy defined by the streaming circuits model. It shows that both the available memory, and the number of concurrently accessible streams have a considerable impact on computational power. Other features of the models, such as having many additional disks to store data on, are not of crucial importance for the power of a model.

Many open problems remain in the classification of models for massive data set computations, notably regarding separation of models. First, it would be interesting to find a more natural problem than Alternating Sequence to separate \(\text{StrC}(1)\) and \(\text{StrC}(2)\). We conjecture that Subsequence is such a problem. In this problem, one is given two strings, and is asked whether one is a subsequence of the other (i.e. the characters of the shorter string occur in order in the longer string, although not necessarily consecutively). A greedy algorithm solves this problem in \(\text{StrC}(2, O(1), O(1))\), but it is not known to be solvable efficiently in \(\text{StrC}(1)\).

A second open problem is to show separations \(\text{StrC}(d) < \text{StrC}(d + 1)\) for constant \(d \geq 2\). It is natural to conjecture that variations of Alternating Sequence can be used for these separations as well, but given the technically difficult proof of Theorem 12, this will probably be a challenging task.
References


A More on Equivalences

A.1 Simulating circuits in StrSort

Proof Sketch (Lemma 5): The proof is similar to the PRAM simulations in the external memory model [CGG+95, ABW02]. To evaluate a circuit, we inductively generate for each level \( \ell \) of the circuit a stream \( S_\ell \) that contains a list of the inputs taken by the circuit nodes on that level, ordered by node. (Note that \( S_1 \) can easily be computed from the input.) One streaming pass on \( S_\ell \) can compute the outputs of all nodes on level \( \ell \). To go from these outputs to \( S_{\ell+1} \), 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 read by multiple gates). Sorting on these labels yields the desired order. \( \square \)
A.2 Sorting in StrC(1)

In this section, we show that sorting can be done efficiently in StrC(1), proving that this model is as powerful as the streaming and sorting model.

Lemma 15 The sorting of an input tape can be done in StrC(1, m, p) using a poly-logarithmic number of passes and logarithmic memory. More precisely,

(i) If the input consists of b-bit integers, it can be sorted with \( p = 3b + 1 \) passes, and \( m = O(b) \) memory.

(ii) If only comparisons of elements are allowed, then the input can be sorted

(a) with \( p = O(\log n) \) passes and \( m = O(\log n) \) memory using randomization, and

(b) with \( p = O(\log^2 n) \) passes and \( m = O(\log n) \) memory deterministically.

Proof: For part (i), we briefly describe how radix sort can be implemented in our model. For \( i = 0, 1, 2, \ldots, b \) we produce a stream \( S_i \) where all elements are ordered according to the \( i \)-th least significant bit, so that \( S_b \) is the input in totally sorted order. We take \( S_0 \) to be the input stream.

To produce \( S_i \) from \( S_{i-1} \), we require three nodes in the circuit. The first node reads \( S_i \), and outputs all elements whose \( i \)-th least significant bit is zero, in the order they appear in \( S_i \). The second node similarly writes out all elements whose \( i \)-th least significant bit is one. A third node concatenates the two produced streams, which yields a \( S_i \) of the desired form, as can easily be verified inductively. This yields the circuit layout shown in figure 2 on page 18, containing \( b \) layers of node triples and one additional node to duplicate the input stream.

Randomized Sorting. For part (ii)(a), we employ a variant of randomized Quicksort. First, we generate a random permutation of the input: in a single pass on the input \( a_1a_2\ldots a_n \), we write out a stream of pairs \( (a_i, r_i) \), where \( r_i \) is a random \( 3 \log n \) bit number. With high probability, all \( r_i \) are distinct, so that sorting the stream on the \( r_i \) (using radix sort from part (i)), yields the input in a random order.

Now we perform a series of Quicksort type pivoting steps. Initially, we produce a stream with entries \( (a_i, \ell_i) \), where the \( a_i \) are the elements to be sorted (in the random order just established), and the \( \ell_i \) are labels, initially the empty string for all \( i \). In the following, we maintain the property that the current stream is a permutation of the input, where the elements having the same label appear in a consecutive stretch of the stream.

For each set of elements with the same label \( \ell \), we use the first element of the group as the pivoting element. A pivoting step itself is done by three nodes. The first node outputs all elements of a set that are less than the pivoting element, indexed with the new label \( \ell 0 \). The second node outputs all elements greater or equal to the pivoting element, with a new label \( \ell 1 \). The concatenation of the two streams is the input for the next set of nodes. We repeat the procedure until all labels are unique (or only equal elements have the same label), which by the usual Quicksort analysis takes only \( O(\log n) \) passes with high probability, since our pivoting elements were chosen uniformly at random.

After this, the elements are not yet in sorted order, but the lexicographic ordering on the labels corresponds to the ordering of the elements themselves. Thus, by applying radix sort on the labels, we obtain the input in sorted order. The total number of passes needed is \( O(\log n) \).

Deterministic Sorting. Our solution becomes more inefficient when a deterministic algorithm is required. Since we cannot interleave streams, implementing Mergesort is not easily possible.
For part (ii)(b), we therefore derandomize the above algorithm by choosing approximate medians as pivoting elements in each set of elements with the same label. This can be accomplished by a single pass and \(O(\log n)\) memory using a result by Greenwald and Khanna [GK01]. In this pass, we append an approximate median to each set of elements with the same label. Since we need the pivoting elements to precede their corresponding sets in order to perform the divide operation, we reverse the order of the stream after inserting the approximate medians; this can be done using a sorting pass with \(O(\log n)\) nodes. Since \(O(\log n)\) Quicksort passes are necessary, this leads to a total number of \(O(\log^2 n)\) required nodes for the circuit. \(\square\)

A.3 Proof of LEMA \((d) \approx \text{StrC}(d)\)

Proof Sketch (Lemma 8): The evaluation of an LEMA in \(\text{StrC}\) proceeds in a straightforward manner. The circuit consists of \(p(n)\) phases. The \(D\) streams entering a phase represent the content of the \(D\) disks at that point (note that we need \(D\) constant such that the circuit has constant width). These \(D\) streams get replicated and fed to \(D\) nodes that each read the streams concurrently to produce the content of one of the \(D\) disks at the point of the next out-of-sequence read/write. These streams then go to the next phase. This shows part (a).

For part (b), let \(S_i\) be the set of streams (including the input stream) created before the \(i\)-th pass, and consumed during or after the \(i\)-th pass of the circuit (also called the “active set” in the proof of Lemma 6). Note that since this set has constant size, the streams contained in it take up only \(O(n)\) space.

The LEMA operates in \(p(n)\) phases and uses \(D = d + 1\) disks. In the \(i\)-th phase, it starts out storing the streams of \(S_i\) on one of its disks, and then performs the operation of the \(i\)-th node in the circuit to compute the content of the streams in \(S_{i+1}\). Creating \(S_1\) is simple, since it contains just the input stream. And \(S_{p(n)+1}\) contains the output stream.

To perform the action of the \(i\)-th node, the LEMA copies the (up to \(d\)) input streams of the node onto different disks; this takes a single pass (reading from one disk, and writing to \(d\) disks). Then the output stream of that node of the streaming circuit is computed by concurrently scanning these \(d\) disks. Another pass is sufficient to rearrange data to produce one disk containing all active streams, i.e. \(S_{i+1}\).

Note that the efficiency of this algorithm can be improved by not rearranging all streams of \(S_{i+1}\) onto a single disk, but this would lower the number of passes only by a constant factor. \(\square\)

B More on Separations

B.1 The decision version of Alternating Sequence

In this section, we briefly sketch how the computation of the Alternating Sequence function can be reduced in \(\text{StrC}(1)\) to the decision version of the 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 problem will imply the hardness of the decision problem.

Suppose we have an algorithm for the decision problem. First, we show how this algorithm allows us to find the last element of the Alternating Sequence output (and not just its color) in \(O(\log^2 n)\) streaming passes. By coloring all \((a_i, a'_i)\) pairs red, and all \((b_i, b'_i)\) pairs blue, and invoking our decision algorithm, we know what type the last element is, suppose it is among the \(a\)-pairs.

If we color all \((a_i, a'_i)\) with \(i < j\) red, and all \((a_i, a'_i)\) with \(i \geq j\) blue then our decision algorithm answers the question “Is the index of the last element less than \(j\)?”. Using binary search, it therefore takes \(O(\log n)\) invocations of our algorithm to determine the index of the last element.
This algorithm computing the last element of the Alternating Sequence output can be used to give a divide-and-conquer solution to Alternating Sequence itself. The idea is to find a consecutive pair $(a_i, a'_i)(b_j, b'_j)$ that appears in “the middle” of the Alternating Sequence output. Then we can divide the input into two halves to be solved independently, because we know where to begin with the second half.

To discover these elements in the middle, we consider the first half of the input streams, i.e.

$$(a_1, a'_1)(a_2, a'_2) \ldots (a_{n/2}, a'_{n/2}) \quad \text{and} \quad (b_1, b'_1)(b_2, b'_2) \ldots (b_{n/2}, b'_{n/2}).$$

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.)

This yields our desired divide-and-conquer algorithm for Alternating Sequence. Note that in each divide step, at least one of the streams gets split into two pieces of at most half its size, showing that only $O(\log n)$ recursions are necessary. Also, all subproblems during a recursion stage can be concatenated into one stream, keeping the width of the circuit constant. Thus, Alternating Sequence can be reduced to the red/blue decision problem, while multiplying the number of passes by $O(\log^2 n)$.

### B.2 Solving Alternating Sequence in StrC(1)

We now give an algorithm in $\text{StrC}(1, m, O((n/m)^{1/2} \log n))$ that solves Alternating Sequence. 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

$$A_1 A_2 \ldots A_{\sqrt{nm}} B_1 B_2 \ldots B_{\sqrt{nm}}, \quad (1)$$

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 sorting the stream, which uses $O(\log n)$ passes in $\text{StrC}(1)$ by Lemma 15(i).

We can use this stream to output the beginning of the answer up to 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 $A_1, A_2, \ldots, A_m$ in memory. This enables us to construct the answer up to $i_k \leq m, j_k \leq \sqrt{nm}$ 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 $A_{m+1}, \ldots, A_{2m}$ 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 stream (1), after reading the whole stream, the A-elements have been processed up to 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.

### B.3 Interleaving two streams into one

In this section, we will prove Lemma 13. 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 \( a_i \) to be the string \( A_1A_{i+1}\ldots A_n \) and \( b_i \) be the string \( B_iB_{i+1}\ldots B_n \). Then the following string of length \( n^2+1 \) has the desired properties:

\[
b_1a_1b_1a_2b_2a_3b_3a_4b_4\ldots a_{n-1}b_{n-1}A_nB_n
\]

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

**Proof (Lemma 13):** Fix \( s \in \Sigma^* \), and let \( s_i \) be the suffix of \( s \) that begins at position \( i \) (e.g. \( s_1 = s \)). We will now define sets of strings \( S_i \) such that all strings in \( S_i \) have to appear as subsequences in \( s_i \). We set \( S_1 := s \), and define the other \( S_i \) inductively. If the \( i \)-th symbol of \( s \) is \( A_k \), then we set

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

Here “\( A_k s \)” 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 \( s \) 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 \( s_i \). So to show the desired lower bound on the length of \( s \), 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 \( A_kB_\ell \)” 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 \( A_kB_{\ell} \) 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 16** Let \( k,k',\ell,\ell' \) 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 \( s \) is an \( A \)-element, say \( A_j \) (the “\( B \)-case” is similar). Then the \( \beta \)-predicates will be unchanged from \( i \) to \( i+1 \). And the \( \alpha \)-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. \( \square \)

Thus, going from \( i \) to \( i+1 \) there will be at most one \( \alpha(...,k,\ell) \), \( \beta(...,k,\ell) \) pair for which one of the predicates becomes false, after both having been true so far. Since at the beginning, all \( n^2 \) such \( \alpha, \beta \)-pairs are true, it takes at least \( n^2 \) elements of \( s \) 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 \( s \) has to contain at least \( n^2 + 1 \) characters. \( \square \)

**B.4 Interleaving streams with small memory**

Note that Corollary 14 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 \( s \), giving a trivial lower bound of \( 2n \). But the corollary is still strong enough for our purposes.

**Proof (Corollary 14):** Group the elements of \( A_1,A_2,\ldots,A_n \) into \( \frac{n}{m+1} \) groups of \( m+1 \) consecutive elements each (i.e. \( A_1,\ldots,A_{m+1} \) form the first group, \( A_{m+2},\ldots,A_{2m+2} \) 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_1 B_1 A_2 B_2 \ldots 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 s must for each interleaved group pair read at least one A-element and one B-element from 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 s must actually be an interleaved string in the sense of Lemma 13 on the group representatives. Since there are \frac{n}{m+1} A- and B-groups each, Lemma 13 therefore implies that s has to have length at least \left( \frac{n}{m+1} \right)^2 + 1. □

### B.5 Adversarial selection of input

For the inductive step, we have to modify our argument slightly. 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 \( a_i \neq a_j, d'_i \neq d'_j, b_i \neq b_j \) 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 \( \ell \) (A,B)-pairs from appearing consecutively in the strings of S, then a simple modification of the proof of Lemma 13 yields a lower bound on \( |s| \) 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 \( t_{i+1} - t_i = \Omega(\sqrt{nm^3(i+1)}) \), 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 \( i \).
Figure 1: The two kinds of gates in a streaming circuit: concatenation of two streams, and a streaming pass on upto $d$ streams.

Figure 2: Streaming circuit that sorts a stream of 4-bit integers.