Parallel Algorithms are Good for Streaming

Camil Demetrescu, Bruno Escoffier, Gabriel Moruz, Andrea Ribichini

UNIVERSITE PARIS DAUPHINE

Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la Décision

CNRS UMR 7024

CAHIER DU LAMSADE

234

Mars 2006

Parallel Algorithms are Good for Streaming

Camil Demetrescu, Bruno Escoffier,
Gabriel Moruz, Andrea Ribichini

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Parallel Algorithms are Good for Streaming[1]

Camil Demetrescu[†], Bruno Escoffier[‡], Gabriel Moruz[§], Andrea Ribichini[†]

Abstract

In this paper we show how PRAM algorithms can be turned into efficient streaming algorithms for several classical combinatorial problems in the W-Stream model. In this model, at each pass one input stream is read and one output stream is written; streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i + 1$. Our techniques yield near-optimal algorithms (up to polylog factors) for several classical problems in this model including sorting, connectivity, minimum spanning tree, biconnected components, and maximal independent set.

1 Introduction

Data stream processing has gained increasing popularity in the last few years as an effective paradigm for processing massive data sets. Huge data streams arise in several modern applications, including database systems, IP traffic analysis, sensor networks, and transaction logs [13, 14, 24]. Streaming can be an effective paradigm also in scenarios where the input data is not necessarily represented in the form of a data stream. Due to the high sequential access rates of modern disks, streaming algorithms can be effectively deployed

[1]Work supported in part the by the Danish National Research Foundation (BRICS, Basic Research in Computer Science, www.brics.dk), by the Sixth Framework Programme of the EU under contract number 001907 ("DELIS: Dynamically Evolving, Large Scale Information Systems"), and by the Italian MIUR Project ALGO-NEXT "Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments".

[†]Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy. {demetres,ribichini}@dis.uniroma1.it

[‡]LAMSADE, CNRS and Université Paris-Dauphine, Paris, France. escoffier@lamsade.dauphine.fr

[§]BRICS, Department of Computer Science, University of Aarhus, Denmark. gabi@daimi.au.dk

for processing massive files on secondary storage [15], providing new insights into the solution of several computational problems in external memory. In the classical read-only streaming model, algorithms are constrained to access the input data sequentially in one (or few) passes, using only a small amount of working memory, typically much smaller than the size of the input [15, 20, 21]. Usual parameters of the model are the working memory size s and the number of passes p that are performed over the data, which are usually functions of the input size. Among the problems that have been studied in this model under the restriction that $p = O(1)$, we recall statistics and data sketching problems (see, e.g., [2, 11, 12]), which can be typically approximated using polylogarithmic working space, and graph problems (see, e.g., [5, 9, 10, 19]), most of which require a working space linear in the vertex set size.

Motivated by practical factors, such as the availability of large amounts of temporary storage at low cost, some authors have recently proposed less restrictive streaming models where algorithms can both read and write data streams. Among them, we mention the $\mathrm{W\text{-}Stream}$ model and the $\mathrm{StrSort}$ model [1, 23]. In the $\mathrm{W\text{-}Stream}$ model, at each pass one input stream is read and one output stream is written; streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i+1$. Despite the use of intermediate streams, which allows it to achieve effective space-passes tradeoffs for fundamental graph problems, most classical lower bounds in read-only streaming hold also in this model [8]. The $\mathrm{StrSort}$ model is just $\mathrm{W\text{-}Stream}$ augmented with a sorting primitive that can be used to reorder each intermediate stream for free. The use of sorting provides a lot of power, making it possible to solve several graph problems with polylogarithmic space and passes [1]. For a comprehensive survey of algorithmic techniques for processing data streams, we refer the interested reader to the extensive bibliographies in [4, 21].

It is well known that algorithmic ideas developed in the context of parallel computational models have inspired the design of efficient algorithms in other models. For instance, Chiang *et al.* [7] showed that efficient external memory algorithms for several problems can be derived from PRAM algorithms using a general simulation. Recently, Aggarwal *et al.* [1] discussed how uniform linear width, poly-logarithmic depth circuits can be simulated efficiently in the $\mathrm{StrSort}$ model, providing a systematic way of constructing algorithms in this model for problems in NC that use a linear number of processors. Examples of problems in this class include undirected connectivity and maximal independent set.

Parallel techniques seem to play a crucial role in the design of efficient algorithms in the $\mathrm{W\text{-}Stream}$ model as well. For instance, the single-source shortest paths algorithm described in [8] is inspired by a framework introduced by Ullman and Yannakakis [26] for the parallel transitive closure problem. However, to the best of our knowledge no general techniques for simulating parallel algorithms in the $\mathrm{W\text{-}Stream}$ model have been addressed so far in the literature.

Our contributions. In this paper, we show how PRAM algorithms can be turned into near-optimal streaming algorithms for several classical combinatorial problems in the W-Stream model. As a first step, we show that any PRAM algorithm that runs in time T using N processors and memory M can be simulated in W-Stream using $p = O((T \cdot N \cdot \log M)/s)$ passes. This yields near-optimal trade-off upper bounds of the form $p = O((n \cdot \text{polylog } n)/s)$ in W-Stream for several problems on sequences and forests of size n. Relevant examples include sorting, list ranking, and Euler tour. For other problems, however, this simulation does not provide good upper bounds. One prominent example is given by graph problems, for which efficient PRAM algorithms typically require $O(m + n)$ processors on graphs with n vertices and m edges. For those problems, our first simulation method would yield bounds of the form $p = O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds in W-Stream are known for many of them.

To overcome this problem, we study an intermediate parallel model, which we call RPRAM, derived from the PRAM model by relaxing the assumption that a processor can only access a constant number of cells at each round. For some problems, this allows it to reduce substantially the number of processors while maintaining the same number of rounds. While this may be unrealistic in practice, we show that simulating RPRAM algorithms in W-Stream leads to near-optimal algorithms (up to polylogarithmic factors) for several fundamental problems, including sorting, minimum spanning tree, biconnected components, and maximal independent set. Algorithms obtained in this way may not always be optimal (although very close to being so): we prove this by showing that for some of the problems above there are better algorithms designed directly in W-Stream without using simulations.

Finally, we show that there exist problems for which the increased computational power of the RPRAM model cannot be exploited to reduce the number of processors required by a PRAM algorithm while maintaining the same time bounds, and thus cannot lead to better W-Stream algorithms: one example is deciding whether a directed graph contains a cycle of length two.

2 Simulating parallel algorithms in W-Stream

In this section we show general techniques for simulating parallel algorithms in the W-Stream model. As we will see in the next sections, our techniques yield near-optimal algorithms for many classical combinatorial problems in the W-Stream model. We first discuss how to simulate general CRCW PRAM algorithms.

Theorem 1 *Let A be a* PRAM *algorithm that uses N processors and runs in time T using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof (Sketch). In the PRAM model, at each parallel round every processor can read $O(1)$ memory cells, perform $O(1)$ instructions to update its internal state, and possibly write $O(1)$ memory cells. We assume that each memory address, cell value, and processor state can be represented using $O(\log M)$ bits. A round of A can be simulated in $\mathrm{W\text{-}Stream}$ by performing $O((N \log M)/s)$ passes, where at each pass we simulate the execution of $\Theta(s/\log M)$ processors using s bits of working memory. The content of the memory cells accessed by the algorithm and the state of each processor are maintained on the intermediate streams as items of the form $(address, value)$ and $(processor, state)$, respectively. The task of each processor can be simulated in a constant number of passes by first reading from the input stream its state and the content of $O(1)$ memory cells, executing an instruction of the algorithm, and then writing to the output stream the new state and possibly the values of the $O(1)$ output cells. Memory cells that remain unchanged are simply propagated through the intermediate streams by just copying them from the input stream to the output stream at each pass. □

There are many examples of problems that can be solved near-optimally in $\mathrm{W\text{-}Stream}$ using Theorem 1. For instance, list ranking can be solved in PRAM in $O(\log n)$ rounds and $O(n/\log n)$ processors [3], where n is the length of the list; by Theorem 1, this yields a $\mathrm{W\text{-}Stream}$ algorithm that runs in $O((n \log n)/s)$ passes. As another example, an Euler tour of a tree with n vertices can be found in parallel in $O(1)$ rounds using n processors [16], which by Theorem 1 yields again a $p = O((n \log n)/s)$ bound in $\mathrm{W\text{-}Stream}$. However, for other problems, the bounds that can be obtained in this way are far from optimal. For instance, efficient PRAM algorithms for graph problems typically require $(m+n)$ processors, where n is the number of vertices, and m is the number of edges. For these problems, Theorem 1 yields bounds of the form $p = O((m \cdot \mathrm{polylog}\ n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds are typically known for many of them.

We now introduce a variant of the PRAM model, which would be completely unrealistic in a practical setting, but will be useful to derive efficient algorithms in $\mathrm{W\text{-}Stream}$ for problems where Theorem 1 does not yield good results.

Definition 1 *An* RPRAM *(Relaxed* PRAM*) is an extended CRCW* PRAM *machine with* N *processors and memory size* M *where at each round a processor can execute* $O(M)$ *instructions, which:*

- *can read all the values stored in memory. Each value can only be read a constant number of times, and no assumptions can be made as to the order in which values are given to the processor;*
- *can write an arbitrary subset of the memory cells. The result of concurrent writes to the same cell in the same round is undefined. Writing can only be performed after all read operations have been done.*

Similarly to a PRAM*, each processor has a constant number of registers of size* $O(\log M)$ *bits.*

Notice that, while in a PRAM each processor can read/write only $O(1)$ memory cells at each round, in an RPRAM this restriction is relaxed so that any number of cells can be accessed in the same parallel round. This jump in computational power allows it to reduce substantially the number of processors used by many classical PRAM algorithms while maintaining the same number of parallel rounds, or even reducing it. As shown below, parallel algorithms implemented in this more powerful model can be simulated in $\mathrm{W\text{-}Stream}$ within the same bounds of Theorem 1.

Theorem 2 *Let* A *be an* RPRAM *algorithm that uses* N *processors and runs in time* T *using space* $M = \mathrm{poly}(N)$*. Then* A *can be simulated in* $\mathrm{W\text{-}Stream}$ *in* $p = O((T \cdot N \cdot \log M)/s)$ *passes using* s *bits of working memory and intermediate streams of size* $O(M + N)$*.*

Proof (Sketch). The claim can be proven similarly to Theorem 1. The main difference is that a processor in the RPRAM model can read and write several memory cells at each round, executing many instructions while still using $O(\log M)$ bits to maintain its internal state. Since the instructions of algorithm A performed by a processor during a round do not assume any particular order for reading the memory cells, reading memory values from the input stream can still be simulated in one pass. Since writing operations occur after all reads have been done, updating memory values can be performed in an additional pass by replacing cell values read from the input stream with the new values in the output stream. □

In the remainder of this paper, we show that parallel algorithms for several classical problems can be naturally implemented in the RPRAM model, yielding by Theorem 2 efficient algorithms in $\mathrm{W\text{-}Stream}$.

3 Sorting

As a first simple application of the simulation techniques introduced in Section 2, we show how to derive efficient sorting algorithms in $\mathrm{W\text{-}Stream}$. We first recall that n items can be sorted on a PRAM with $O(n)$ processors in $O(\log n)$ parallel rounds [16]. By Theorem 1, this yields a $\mathrm{W\text{-}Stream}$ sorting algorithm that runs in $p = O((n \log^2 n)/s)$ passes. In RPRAM, however, sorting can be solved by $O(n)$ processors in constant time as follows. Each processor is assigned to an input item; in one parallel round it scans the entire memory and counts the number of items smaller than and equal to the item the processor is assigned to. Let i and j be those numbers, respectively. Then each processor

writes its own item into all the cells with indices in the range $i+1$ through $i+1+j$, thus producing a sorted sequence. This yields the following theorem.

Theorem 3 *Sorting n items in* RPRAM *can be done in $O(1)$ parallel rounds using $O(n)$ processors.*

Using Theorem 2 and Theorem 3, we obtain a W-Stream sorting algorithm that takes $p = O((n \log n)/s)$ passes, thus matching the performance of the best known algorithm for sorting in a streaming setting [20]. Since we can prove that sorting requires $p = \Omega(n/s)$ passes in the W-Stream model, this bound is essentially optimal.

4 Graph algorithms

In this section we discuss how to derive efficient W-Stream algorithms for several graph problems using the RPRAM simulation of Theorem 2. We notice that efficient PRAM graph algorithms typically require $O(m+n)$ processors [6] on graphs with n vertices and m edges. Simulating such algorithms in W-Stream using Theorem 1 would yield bounds of the form $p = O((m \cdot \mathrm{polylog}\ n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds in W-Stream are known for many of them. Graph connectivity is one prominent example [8]. Notice that, assigning each vertex to a processor, RPRAM gives enough power for each vertex to scan its entire neighborhood in a single parallel round. Since many parallel graph algorithms can be implemented using repeated neighborhood scanning, in many cases this allows it to reduce the number of processors from $O(m+n)$ to $O(n)$ while maintaining the same running time. By Theorem 2, this yields improved bounds of the form $p = O((n \cdot \mathrm{polylog}\ n)/s)$. We now show however that there are graph problems for which this is impossible.

Lemma 1 *Testing whether a directed graph with m edges contains a cycle of length two requires $p = \Omega(m/s)$ passes in* W-Stream.

Proof (Sketch). We prove the lower bound by showing a reduction from the bit vector disjointness two-party communication complexity problem. Alice has an m-bit vector A and Bob has an m-bit vector B; they wish to know whether A and B are disjoint, i.e. $A \cdot B > 0$. Let $e(i) = (x_i, y_i)$ and let $e^r(i) = (y_i, x_i)$, where $x_i = i$ div $\lceil \sqrt{m} \rceil$ and $y_i = i \bmod \lceil \sqrt{m} \rceil$. To make the reduction, Alice creates a stream containing an edge $e(i)$ for each i such that $A[i] = 1$ and Bob creates a stream containing an edge $e^r(i)$ for each i such that $B[i] = 1$. Let G be the directed graph induced by the union of the $\leq 2m$ edges in the streams created by Alice and Bob. Clearly, there is a cycle of length two in G if and only if $A \cdot B > 0$. Since solving bit vector disjointness requires transmitting $\Omega(m)$ bits [17], and the distributed execution of any streaming algorithm would require

the working memory image to be sent back and forth from Alice to Bob at each pass, it must be $p \cdot s = \Omega(m)$. Hence, $p = \Omega(m/s)$. □

Lemma 1 implies that testing whether a directed graph has a cycle of length two requires $\Omega(m/(n \log n))$ rounds on an RPRAM with n processors. Notice that this problem can be easily solved in one round on a PRAM with $O(m+n)$ processors, by just checking in parallel whether there is any edge (x, y) that also appears as (y, x) in the graph.

4.1 Connected components (CC)

A classical PRAM random-mating algorithm for computing the connected components of a graph with n vertices and m edges requires $O(m + n)$ processors and $O(\log n)$ time with high probability [6, 22]. We first recall how the algorithm works, and then we show that it can be implemented in RPRAM with just $O(n)$ processors. By Theorem 2, this yields a near optimal algorithm in W-Stream.

Parallel algorithm. The algorithm is based on building a set of star subgraphs and contracting the stars. It iterates the following sequence of steps.

1. Each vertex is assigned the status of parent or child independently with probability $1/2$;
2. For each child vertex u, determine if it is adjacent to a parent vertex. If so, choose one such a vertex to be the parent $f(u)$ of u, and replace each edge (u, v) by $(f(u), v)$ and each edge (v, u) by $(u, f(v))$;
3. For each vertex having parent u, set the parent to $f(u)$.

In this contraction phase, the probability that a vertex is assigned to a connected component, i.e. has a neighbor parent, is at least $1/4$. It has been shown that with high probability the number of iterations is bounded by $O(\log n)$.

RPRAM implementation. Each iteration can be implemented in RPRAM in constant time using $O(n)$ processors as follows. We attach a processor to each vertex. In a round we assign to each vertex the status of parent of child. In a second round each vertex scans its neighborhood to find a parent, if exists (in case of several parents, we can break ties arbitrarily). Updating the parents as in step three can also be done in one round. This yields the following bound.

Theorem 4 *CC can be solved in* RPRAM *using* n *processors in* $O(\log n)$ *rounds with high probability.*

By Theorem 2, this yields the following bound in W-Stream.

Corollary 1 *CC can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes with high probability.*

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [8], this upper bound is optimal up to a polylogarithmic factor. We note however that CC can be solved deterministically in W-Stream in $O((n \log n)/s)$ passes [8].

4.2 Minimum spanning tree (MST)

As explained in [6], the randomized CC algorithm given above can be extended to find a minimum spanning tree in a (connected) graph.

Parallel algorithm. The algorithm is based on the property that given a subset V' of vertices, a minimum weight edge having one and only one endpoint in V' is in any MST. We modify the CC algorithm as follows. At the second step, each child vertex u determines the minimum weight incident edge (u, v). If v is a parent vertex, then $f(u) = v$ and flag the edge (u, v) as belonging to the spanning tree. If v is not a parent vertex, do nothing. This algorithm computes a MST and performs $O(\log n)$ iterations with high probability.

RPRAM implementation. Assuming that edge weights can be encoded using $O(\log n)$ bits, the modification does not affect significantly the number of rounds needed in RPRAM, since each iteration can still be performed in $O(1)$ rounds.

Theorem 5 *MST can be solved in* RPRAM *using* n *processors in* $O(\log n)$ *rounds with high probability.*

By Theorem 2, we obtain the following bound in W-Stream.

Corollary 2 *MST can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes.*

We now show that we can improve this bound by a $\log n$ factor with an algorithm designed directly in W-Stream.

W-Stream algorithm. We compute the MST by progressively adding edges as follows. We compute for each vertex the minimum weight edge incident to it. This set of edges E' is added to the MST. Then we compute the connected components induced by E' and contract the graph by considering each connected components as a single vertex. We repeat these steps until the graph contains a single vertex. More precisely, we consider at each iteration a contracted graph where the vertices are the connected components of the partial MST so far computed. Let $G_i = (V_i, E_i)$ be the graph before the i^{th} iteration:

1. for each vertex $u \in V_i$, we find a minimum weight edge (u, v) incident to u and add (u, v) to the MST. Let $E'_i = \{(u, v), u \in V_i\}$;

2. we run a CC algorithm on the graph (V_i, E'_i). The connected components obtained in this way are the vertices of V_{i+1};

3. we replace each edge (u, v) by $(c(u), c(v))$, where $c(u)$ and $c(v)$ represent the labels of the connected components previously computed.

We now analyze the number of passes required in W-Stream. Let $|V_i| = n_i$. The first and the third steps require $O((n_i \log n)/s)$ passes each, since we can process in one pass $O(s/\log n)$ vertices. As computing the connected components takes $O((n_i \log n)/s)$ passes, the i^{th} iteration requires $O((n_i \log n)/s)$ passes. We note that at each iteration, we add an edge for every vertex in V_i. Hence, the number of connected components is divided by at least two. Therefore the total number of passes performed is given by $T(n) = T(n/2) + O((n \log n)/s)$, which sums up to $O((n \log n)/s)$.

Theorem 6 *MST can be computed in* $O((n \log n)/s)$ *passes in* W-Stream.

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [8], this upper bound is optimal up to a logarithmic factor. To the best of our knowledge, no previous algorithm was known for MST in W-Stream.

4.3 Biconnected components (BCC)

Tarjan and Vishkin [25] give a PRAM algorithm that computes the biconnected components (BCC) of an undirected graph in $O(\log n)$ time using $O(m+n)$ processors. We give an RPRAM implementation of their algorithm that can be simulated in W-Stream using $O((n \log^2 n)/s)$ passes. We also give a direct implementation that uses only $O((n \log n)/s)$ passes.

Parallel algorithm. Given a graph G, the algorithm considers a graph G' such that the vertices in G' correspond to the edges in G and the connected components of G' correspond to the biconnected components of G. The algorithm starts from a rooted spanning tree T on G and builds a subgraph G'' of G' having as vertices all the edges of T. The edges of G'' are built in such a way that two vertices are in the same connected component of G'' if and only if the corresponding edges in G are in the same biconnected component. After computing the connected components of G'' the algorithm appends the remaining edges of G to their corresponding biconnected component. We now briefly sketch the five steps of the algorithm:

1. build a rooted spanning tree T of G and compute for each vertex its preorder and postorder numbers together with the number of descendants. Also, the vertices are labelled with their preorder numbers.

2. compute for each vertex u two values, $low(u)$ and $high(u)$, given as follows.

$$\begin{aligned} low(u) &= \min(\{u\} \cup \{low(w) | p(w) = u\} \cup \{w | (u,w) \in G \setminus T\}) \\ high(u) &= \max(\{u\} \cup \{high(w) | p(w) = u\} \cup \{w | (u,w) \in G \setminus T\}), \end{aligned}$$

   where $p(u)$ denotes the parent of vertex u.

3. add edges to G'' according to the following two rules. For all $(w,v) \in G \setminus T$ with $v + desc(v) \leq w$, add $((p(v),v),(p(w),w))$ to G'' and for all $(v,w) \in T$ with $p(w) = v$ and $v \neq 1$, add $((p(v),v),(v,w))$ to G'' if $low(w) < v$ or $high(w) \geq v + desc(v)$, where $desc(v)$ denotes the number of descendants of vertex v.

4. compute the connected components of G''.

5. add the remaining edges of G to their biconnected components. Each edge $(v,w) \in G \setminus T$, with $v < w$, is assigned to the biconnected component of $(p(w),w)$.

RPRAM implementation. We give RPRAM descriptions for all the five steps of the algorithm, each of them using $O(\log n)$ time and $O(n)$ processors. First, we compute a spanning tree for the graph using the RPRAM algorithm previously introduced. Rooting the tree and computing for each vertex the preorder and postorder numbers as well as the number of descendants can be done using list ranking and Euler tour [25], which take $O(\log n)$ time and $O(n)$ processors in PRAM, thus in RPRAM. Since the second step takes $O(\log n)$ time using $O(n)$ processors in PRAM [25], the same bounds hold for RPRAM. As for the third step, we implement it in RPRAM in constant time and $O(n)$ processors by allowing every vertex to read its neighborhood. For computing the connected components of G'' in the fourth step, we use the RPRAM algorithm previously introduced that takes $O(\log n)$ time and $O(n)$ processors. Finally, we implement the

last step of the algorithm in RPRAM in $O(1)$ time and $O(n)$ processors by reading for all vertices v the whole neighborhood and assign the edges to the proper biconnected components.

Since all the steps of the algorithm can be implemented in RPRAM in $O(\log n)$ rounds and $O(n)$ processors, we get the following result.

Theorem 7 *BCC can be solved in* RPRAM *using* $O(n)$ *processors in* $O(\log n)$ *rounds.*

By Theorem 2, this yields the following bound in $\mathrm{W\text{-}Stream}$.

Corollary 3 *BCC can be solved in* $\mathrm{W\text{-}Stream}$ *in* $O((n \log^2 n)/s)$ *passes.*

We now show that we can improve this bound by a $\log n$ factor with an implementation designed directly in $\mathrm{W\text{-}Stream}$.

W-Stream algorithm. We describe how to implement directly in $\mathrm{W\text{-}Stream}$ the steps of the parallel algorithm of Tarjan and Vishkin [25]. Notice that we have given constant time RPRAM descriptions for the third and the fifth step, thus by applying the simulation in Theorem 2 we obtain $\mathrm{W\text{-}Stream}$ algorithms that run in $O((n \log n)/s)$ passes. For computing the connected components in the fourth step, we can use the algorithm in [8] that requires $O((n \log n)/s)$ passes. Therefore, to achieve a global bound of $O((n \log n)/s)$ passes, we must give implementations that run in $O((n \log n)/s)$ passes for the first two steps. For the first step, we can compute a spanning tree within the bound of Theorem 6. Rooting the tree and computing the preorder and postorder numbers together with the number of descendants can be easily implemented in $O((n \log n)/s)$ passes using list ranking, Euler tour and sorting. As for the second step, we compute the low and $high$ values by processing $s/\log n$ vertices at each pass, according to the postorder numbers.

Theorem 8 *BCC can be solved in* $\mathrm{W\text{-}Stream}$ *in* $O((n \log n)/s)$ *passes.*

By the $p = \Omega(n/s)$ lower bound for CC in $\mathrm{W\text{-}Stream}$ [8], this upper bound is optimal up to a logarithmic factor. To the best of our knowledge, no previous algorithm was known for BCC in $\mathrm{W\text{-}Stream}$.

4.4 Maximal independent set (MIS)

We give an efficient RPRAM algorithm for the maximal independent set problem (MIS), based on the PRAM algorithm proposed by Luby [18]. Through the simulation in Theorem 2, this leads to an efficient $\mathrm{W\text{-}Stream}$ implementation.

Parallel algorithm. A maximal independent set S of a graph G can be built incrementally through a series of iterations, as follows. At each iteration, in the first step, a random subset I of the vertices in G is determined, by including each vertex with probability $1/(2 \cdot deg(v))$. Then, in the second step, for each edge (u, v) of G such that $u, v \in I$, the vertex with the smallest degree is removed from I. Finally, in the third step, the vertices in I are added to S, and then both the vertices in I and their neighbors are removed from G. The above steps are repeated until G gets empty.

RPRAM implementation. The first step of each iteration can clearly be implemented in constant time by an RPRAM with $O(n)$ processors, since each vertex can compute its own degree in one parallel round. The second step also can be implemented in constant time, by having each vertex in I scan its neighborhood, and remove itself if it encounters a neighbor also in I and with a larger degree. Finally, the third step can be implemented in constant time as well, since each vertex not in I, in just one parallel round, can scan its neighborhood and remove itself from G if one of its neighbors is in I.

Theorem 9 *MIS can be solved in* RPRAM *using* $O(n)$ *processors in* $O(\log n)$ *rounds with high probability.*

By Theorem 2, this yields the following bound in W-Stream.

Corollary 4 *MIS can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes with high probability.*

We now show that the bound of Corollary 4 is optimal up to a polylogarithmic factor.

Theorem 10 *MIS requires* $\Omega(n/s)$ *passes in* W-Stream.

Proof (Sketch). The proof is based on a reduction from the bit vector disjointness communication complexity problem. Alice and Bob have each a vector of size n. They build a graph on $4n$ vertices v_i^j, where $i = 1, \cdots, n$ and $j = 1, \cdots, 4$. If $A_i = 0$, then Alice adds edges (v_i^1, v_i^2) and (v_i^3, v_i^4), whereas if $B_i = 0$, then Bob adds edges (v_i^1, v_i^3) and (v_i^2, v_i^4). The size of any MIS is $2n$ if $A \cdot B = 0$ and strictly greater otherwise. □

To the best of our knowledge, no previous algorithm was known for MIS in W-Stream. It remains an open question whether one can solve this problem more efficiently in this model.

References

[1] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, 2004.

[2] 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.

[3] R. Anderson and M. G. L. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS'02)*, pages 1–16, 2002.

[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02), San Francisco, California, pp. 623–632*, 2002.

[6] G. Blelloch and B. Maggs. Parallel algorithms. In *The Computer Science and Engineering Handbook*, pages 277–315. 1997.

[7] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vemgroff, and J. Vitter. External-memory graph algorithms. In *Proc. 6th Annual ACM-SIAM Symposium on Dicrete Algorithms (SODA'95)*, pages 139–149, 1995.

[8] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proc. 17th Annual ACM-SIAM Symposium of Discrete Algorithms (SODA'06)*, pages 714–723, 2006.

[9] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proc. of the 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, 2004.

[10] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of the 16th ACM/SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 745–754, 2005.

[11] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L^1 difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002.

[12] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC'02)*, pages 389–398, 2002.

[13] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data. Technical report, DIMACS Technical Report 2001-43, 2001.

[14] L. Golab and M. Ozsu. Data stream management issues Ð a survey. Technical report, School of Computer Science, University of Waterloo, TR CS-2003-08, 2003.

[15] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *In "External Memory algorithms", DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 50:107–118, 1999.

[16] J. Jájá. *An introduction to parallel algorithms*. Addison-Wesley, 1992.

[17] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[18] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1053, 1986.

[19] A. McGregor. Finding matchings in the streaming model. In *Proceedings of the 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'05), to appear*, 2005.

[20] I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[21] S. Muthukrishnan. Data streams: algorithms and applications. Technical report, 2003. Available at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[22] J. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR 08-85, Aiken Computation Laboratory, Harvard University, Cambridge, 1985.

[23] J. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massauchussets Institute of Technology, September 2003.

[24] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings USENIX Annual Technical Conference*, 1998.

[25] R. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science (FOCS'84)*, pages 12–20, 1984.

[26] J. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.