Adapting Parallel Algorithms to the W-Stream Model, with Applications to Graph Problems*

Camil Demetrescu[1], Bruno Escoffier[2], Gabriel Moruz[3], and Andrea Ribichini[1]

[1] Dipartimento di Informatica e Sistemistica,
Università di Roma "La Sapienza", Rome, Italy
{demetres,ribichini}@dis.uniroma1.it
[2] Lamsade, Université Paris Dauphine, France
escoffier@lamsade.dauphine.fr
[3] MADALGO, BRICS, Department of Computer Science,
University of Aarhus, Denmark
gabi@daimi.au.dk

Abstract. In this paper we show how parallel 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 give new insights on developing streaming algorithms and yield 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, 23]. Streaming is an effective paradigm also in scenarios where the input data is not necessarily represented as a data stream. Due to high sequential access rates of modern disks, streaming algorithms can be effectively deployed for processing massive files on secondary storage [14], providing new insights into the solution of 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 input size [14, 18, 19]. 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

* Supported in part 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 "MAINSTREAM: Algorithms for massive information structures and data streams".

L. Kučera and A. Kučera (Eds.): MFCS 2007, LNCS 4708, pp. 194–205, 2007.

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]), most of which require a working space linear in the vertex set size.

Motivated by practical factors, such as availability of large amounts of secondary storage at low cost, a number of authors have recently proposed less restrictive streaming models, where algorithms can both read and write data streams. Among them, we mention the W-Stream model and the StrSort model [1, 21]. In the W-Stream model, at each pass we operate with an input stream and an output stream. The 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 achieving effective space-passes tradeoffs for fundamental graph problems, most classical lower bounds in read-only streaming hold also in this model [8]. The StrSort model is just W-Stream augmented with a sorting primitive that can be used at each pass to reorder the output stream for free. Sorting provides a lot of computational power, making it possible to solve several graph problems using polylog passes and working space [1]. For a comprehensive survey of algorithmic techniques for processing data streams, we refer the interested reader to the extensive bibliographies in [4, 19].

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 can be derived from PRAM algorithms using a general simulation. Aggarwal *et al.* [1] discussed how circuits with uniform linear width and polylog depth (NC) can be simulated efficiently in StrSort, 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 W-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 [25] for the parallel transitive closure problem. However, to the best of our knowledge, no general techniques for simulating parallel algorithms in the W-Stream model have been addressed so far in the literature.

Our Contributions. In this paper, we show how classical parallel algorithms designed in the PRAM model can be turned into near-optimal algorithms in W-Stream for several classical combinatorial problems. We first 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, where n is the input size. Relevant examples include sorting, list ranking, and Euler tour. For other problems, however, this simulation does not provide good upper bounds. One prominent example concerns graph problems, for which efficient PRAM algorithms typically require $O(m+n)$ processors on graphs with n vertices and m edges. For those problems,

this simulation method yields $p = O((m \cdot \text{polylog}\, n)/s)$ bounds, 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. This way, we get the PRAM algorithms closer to streaming algorithms, since a memory cell in the working memory can be processed against an arbitrary number of cells in the stream. For some problems, this enhancement allows us to substantially reduce the number of processors while maintaining the same number of rounds. 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. Since algorithms obtained in this way are not always optimal – although very close to being so –, for some of the problems above we give better *ad hoc* 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 does not help in reducing the number of processors required by a PRAM algorithm while maintaining the same time bounds, and thus cannot lead to better W-Stream algorithms. An 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 W-Stream. We show in the next sections that our techniques yield near-optimal algorithms for many classical combinatorial problems in the W-Stream model. In Theorem 1 we discuss how to simulate general CRCW PRAM algorithms. Throughout this paper, we assume that each memory address, cell value, and processor state can be stored using $O(\log M)$ bits, where M is the memory size of the parallel machine.

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 may read $O(1)$ memory cells, perform $O(1)$ instructions to update its internal state, and write $O(1)$ memory cells. A round of A can be simulated in W-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. We simulate the task of each processor in a constant number of passes as follows. We first read from the input stream its state and the content of the $O(1)$ memory cells used by A and then we execute the $O(1)$ instructions performed. Finally, we write 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 W-Stream using Theorem 1. For instance, solving list ranking in PRAM takes $O(\log n)$ rounds and $O(n/\log n)$ processors [3], where n is the length of the list. By Theorem 1, we obtain a W-Stream algorithm that runs in $O((n \log n)/s)$ passes. An Euler tour of a tree with n vertices is computed in parallel in $O(1)$ rounds using $O(n)$ processors [15], which by Theorem 1 yields again a $p = O((n \log n)/s)$ bound in W-Stream. However, for other problems, the bounds obtained this way are far from being optimal. For instance, efficient PRAM algorithms for graph problems typically require $O(m + n)$ 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 $p = O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds are known for many of them.

In Definition 1 we introduce RPRAM as an extension of the PRAM model. It allows every processor to handle in a parallel round not only $O(1)$ memory cells, but an arbitrary number of cells. Since in W-Stream a value in the working memory might be processed against all the data in the stream, we view RPRAM as a natural link between PRAM and W-Stream, even though it may be unrealistic in a practical setting. We first introduce a generic simulation that turns RPRAM algorithms into W-Stream algorithms. We then give RPRAM implementations that lead to efficient algorithms in W-Stream for a number of problems where the PRAM simulation in Theorem 1 does not yield good results.

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

- *can read an arbitrary number of memory cells. Each cell can only be read a constant number of times during the round, 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 by different processors 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.*

The jump in computational power provided by RPRAM allows substantial improvements for many classical PRAM algorithms such as decreasing the number of parallel rounds while preserving the number of processors or reducing the number of processors used while maintaining the same number of parallel rounds. We show in Theorem 2 that parallel algorithms implemented in this more powerful model can be simulated in W-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 $M = \text{poly}(N)$. 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). We follow the proof of Theorem 1. The main difference is that a processor in the RPRAM model can read and write an arbitrary number of 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. Replacing cell values read from the input stream with the new values written on the output stream can be performed in one additional pass.

3 Sorting

As a first simple application of the simulation techniques introduced in Section 2, we show how to derive efficient sorting algorithms in W-Stream. We first recall that n items can be sorted on a PRAM with $O(n)$ processors in $O(\log n)$ parallel rounds and $O(n \log n)$ comparisons [15]. By Theorem 1, this yields a W-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 numbers i and j of items smaller than and equal to the item the processor is assigned to respectively. Then each processor writes its own item into all the cells with indices between $i + 1$ and $i + 1 + j$, and thus we obtain a sorted sequence.

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

Using the simulation in Theorem 2, we obtain the result stated below.

Corollary 1. *Sorting n items in* W-Stream *can be performed in $O(n \log n/s)$ passes.*

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 [18]. Since sorting requires $p = \Omega(n/s)$ passes in W-Stream, this bound is essentially optimal. However, both our algorithm and the algorithm in [18] perform $O(n^2)$ comparisons. We reduce the number of comparisons to the optimal $O(n \log n)$ at the expense of increasing the number of passes to $O((n \log^2 n)/s)$ by simulating an optimal PRAM algorithm via Theorem 1, as stated before.

4 Graph Problems

In this section we discuss how to derive efficient W-Stream algorithms for several graph problems using the RPRAM simulation in Theorem 2. Since efficient PRAM graph algorithms typically require $O(m + n)$ processors on graphs with n vertices and m edges [6], simulating such algorithms in W-Stream using Theorem 1 yields 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. 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 us 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 $p = O((n \cdot \text{polylog } n)/s)$.

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 uses $O(m+n)$ processors and runs in $O(\log n)$ time with high probability [6, 20]. We first describe the algorithm and then we give an RPRAM implementation that uses only $O(n)$ processors which, by Theorem 2, leads to a nearly optimal algorithm in W-Stream.

PRAM Algorithm. The algorithm is based on building a set of star subgraphs and contracting the stars. It each parallel round it performs 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 whether 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 $(f(v), u)$;
3. For each vertex having parent u, set the parent to $f(u)$.

The algorithm performs $O(\log n)$ parallel rounds with high probability [6].

RPRAM Implementation. We show how to implement each parallel round in RPRAM in $O(1)$ rounds using only $O(n)$ processors. We attach a processor to each vertex. We first assign each vertex the status of parent or child, and then for each vertex we scan its neighborhood to find a parent, if there exists one (in case of several parents, we break ties arbitrarily). Updating the parents according to the third step also takes one round in RPRAM. We obtain the result in Theorem 4.

Theorem 4. *Solving CC in* RPRAM *takes* $O(n)$ *processors and* $O(\log n)$ *rounds with high probability.*

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

Corollary 2. *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 notice that the same bound can be achieved deteministically by starting from the PRAM algorithm for CC in [22]. This bound can be further improved to $O((n \log n)/s)$ passes as shown in [8].

4.2 Minimum Spanning Tree (MST)

In this section, we first describe the PRAM algorithm in [6] for computing the MST of an undirected graph. We then give an RPRAM implementation that leads to an optimal algorithm (up to a polylog factor) in W-Stream by using the simulation in Theorem 2. Finally, we give an algorithm designed in W-Stream that outperforms the algorithm obtained via simulation.

PRAM Algorithm. The randomized CC algorithm previously introduced can be extended to find a minimum spanning tree in a (connected) graph [6]. It also takes $O(\log n)$ rounds with high probability and uses $O(m+n)$ processors. 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 some MST. We modify the second step of the CC algorithm as follows. Each child vertex u determines the minimum weight incident edge (u, v). If v is a parent vertex, then we set $f(u) = v$ and flag the edge (u, v) as belonging to the spanning tree. This algorithm computes a MST and performs $O(\log n)$ rounds with high probability.

RPRAM Implementation. The updated second step runs in $O(1)$ rounds in RPRAM and uses $O(n)$ processors. Since the implementations of the other steps of the CC algorithm are unchanged and take $O(1)$ rounds and $O(n)$ processors, we obtain the result stated in Theorem 5.

Theorem 5. *MST is solvable in* RPRAM *using* $O(n)$ *processors and* $O(\log n)$ *rounds with high probability.*

Assuming edge weights can be encoded using $O(\log n)$ bits, we obtain the following bound in W-Stream by Theorem 2.

Corollary 3. *MST can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes.*

We now give a deterministic algorithm designed directly in W-Stream that improves the bounds achieved by using the simulation.

A Faster ad hoc *W-Stream Algorithm.* We again assume edge weights can be encoded using $O(\log n)$ bits. We build 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. We then compute the connected components induced by E' and contract the graph by considering each connected component a single vertex. We repeat these steps until the graph contains a single vertex or there are no more edges to add. More precisely, we consider at each iteration a contracted graph where the vertices are the connected components of the partial MST so far computed. Denoting $G_i = (V_i, E_i)$ the graph before the i^{th} iteration, the $(i+1)^{th}$ iteration consists of the following steps.

1. for each vertex $u \in V_i$, we compute a minimum weight edge (u, v) incident to u, and flag (u, v) as belonging to the MST (cycles that might occur due to weight ties are avoided by using a tie-breaking rule). Denote $E'_i = \{(u, v), u \in V_i\}$ the set of flagged edges.

2. we run a CC algorithm on the graph (V_i, E'_i). The resulted connected components are the vertices of V_{i+1}.
3. we replace each edge (u, v) by $(c(u), c(v))$, where $c(u)$ and $c(v)$ denote 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. Computing the connected components also takes $O((n_i \log n)/s)$ passes, and therefore 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 and thus $|V_{i+1}| \leq |V_i|/2$, i.e., the number of connected components is divided by at least two. We obtain that the total number of passes performed in the worst case 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 polylog 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 [24] gave 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 uses only $O(n)$ processors while preserving the time bounds and thus can be turned using Theorem 2 in a W-Stream algorithm that runs in $O((n \log^2 n)/s)$ passes. We also give a direct implementation that uses only $O((n \log n)/s)$ passes.

PRAM Algorithm. Given a graph G, the algorithm considers a graph G' such that vertices in G' correspond to edges in G and connected components in G' correspond to biconnected components in G. The algorithm first computes a rooted spanning tree T of G and then builds a subgraph G'' of G' having as vertices all the edges of T. The edges of G'' are chosen such 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 components. 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, label the vertices by their preorder numbers.
2. for each vertex u, compute two values, $low(u)$ and $high(u)$, as follows.

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

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

3. add edges to G'' according to the following two rules. For all edges $(w, v) \in G \setminus T$ with $v + desc(v) \leq w$, add $((p(v), v), (p(w), w))$ to G'', and for all $(v, w) \in T$ with $p(w) = v$, $v \neq 1$, 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 of 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 are performed using list ranking and Euler tour [24], which take $O(\log n)$ time and $O(n)$ processors in PRAM, and thus in RPRAM. Since the second step takes $O(\log n)$ time using $O(n)$ processors in PRAM [24], the same bounds hold for RPRAM. We implement the third step in RPRAM in constant time and $O(n)$ processors, since it suffices a scan of the neighborhood for each vertex. 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 scanning the neighborhood for all vertices v and assigning the edges to the proper biconnected components. Since we implement all the steps of the algorithm in RPRAM in $O(\log n)$ rounds and $O(n)$ processors, we obtain the following result.

Theorem 7. *BCC 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. *BCC can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes with high probability.*

We now show that we can achieve better bounds with an implementation designed directly in W-Stream.

A Faster ad hoc *W-Stream Algorithm.* We describe how to implement directly in W-Stream the steps of the parallel algorithm of Tarjan and Vishkin [24]. 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 W-Stream algorithms that run in $O((n \log n)/s)$ passes. For computing the connected components in the fourth step, we 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, it suffices to 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 pos-

torder numbers together with the number of descendants can be implemented in $O((n \log n)/s)$ passes using list ranking, Euler tour and sorting. Concerning the second step, we compute the *low* and *high* values by processing $\Theta(s/\log n)$ vertices at each pass, according to the postorder numbers.

Theorem 8. *BCC can be solved in* W-Stream *in* $O((n \log n)/s)$ *passes in the worst case.*

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [8], this upper bound is optimal up to a polylog factor. To the best of our knowledge, no previous algorithm was known for BCC in W-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 [17]. Using the simulation in Theorem 2, this leads to an efficient W-Stream implementation.

PRAM Algorithm. A maximal independent set S of a graph G is incrementally built through a series of iterations, where each iteration consists of a sequence of three steps, as follows. In the first step, we compute a random subset I of the vertices in G, by including each vertex v with probability $1/(2 \cdot deg(v))$. Then, for each edge (u, v) in G, with $u, v \in I$, we remove from I the vertex with the smallest degree. Finally, in the third step, we add to S the vertices in I, and then we remove from G the vertices in I together with their neighbors. The above steps are iterated until G gets empty. The algorithm uses $O(m + n)$ processors and $O(\log n)$ parallel rounds.

RPRAM Implementation. We implement the first step of each iteration in constant time and $O(n)$ processors in RPRAM, since it requires each vertex to compute its own degree. The second step can also be implemented in constant time, by having each vertex in I scan its neighborhood, and remove itself upon encountering a neighbor also in I with a larger degree. Finally, we implement the third step in constant time as well by scanning the neighborhood of each vertex that is not in I, and removing it from G if at least one of its neighbors is in I. Since the algorithm performs $O(\log n)$ iterations with high probability [17], we obtain the bound in Theorem 9.

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

We now show that the bound in Corollary 5 is optimal up to a polylog 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 has an n-bit vector A and Bob has an

n-bit vector B; they wish to know whether A and B are disjoint, i.e., $A \cdot B = 0$. 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.

5 Limits of the RPRAM Approach

In this section we prove that the increased power that RPRAM provides does not always help in reducing the number of processors to $O(n)$ and thus in obtaining W-Stream algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes. As an example, in Theorem 11 we prove that detecting cycles of length two in a graph takes $\Omega(m/s)$ passes.

Theorem 11. *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., $A \cdot B = 0$. Alice creates a stream containing an edge $e(i) = (x_i, y_i)$ for each i such that $A[i] = 1$ and Bob creates a stream containing an edge $e^r(i) = (y_i, x_i)$ for each i such that $B[i] = 1$, where $x_i = i \text{ div } \lceil \sqrt{m} \rceil$ and $y_i = i \bmod \lceil \sqrt{m} \rceil$. Let G be the directed graph induced by the union of the 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 [16], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from Alice to Bob at each pass, we obtain $s = \Omega(m)$, which leads to $p = \Omega(m/s)$.

Testing whether a digraph has a cycle of length two can be easily done in one round in RPRAM using $O(m)$ processors, by just checking in parallel whether there is any edge (x, y) that also appears as (y, x) in the graph. This leads to an algorithm in W-Stream that runs in $O((m \log n)/s)$ passes by Theorem 2.

References

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

[2] Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Computer and System Sciences 58(1), 137–147 (1999)

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

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

[5] Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proc. 13th annual ACM-SIAM symposium on Discrete algorithms (SODA'02), pp. 623–632. ACM Press, New York (2002)
[6] Blelloch, G., Maggs, B.: Parallel algorithms. In: The Computer Science and Engineering Handbook, pp. 277–315 (1997)
[7] Chiang, Y., Goodrich, M., Grove, E., Tamassia, R., Vemgroff, D., Vitter, J.: External-memory graph algorithms. In: Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'95), pp. 139–149. ACM Press, New York (1995)
[8] Demetrescu, C., Finocchi, I., Ribichini, A.: Trading off space for passes in graph streaming problems. In: Proc. 17th Annual ACM-SIAM Symposium of Discrete Algorithms (SODA'06), pp. 714–723. ACM Press, New York (2006)
[9] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 207–216. Springer, Heidelberg (2004)
[10] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the streaming model: the value of space. In: Proceedings of the 16th ACM/SIAM Symposium on Discrete Algorithms (SODA'05), pp. 745–754 (2005)
[11] Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: An approximate L^1 difference algorithm for massive data streams. SIAM Journal on Computing 32(1), 131–151 (2002)
[12] Gilbert, A., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Fast, small-space algorithms for approximate histogram maintenance. In: Proc. 34th ACM Symposium on Theory of Computing (STOC'02), pp. 389–398. ACM Press, New York (2002)
[13] Golab, L., Ozsu, M.: Data stream management issues: a survey. Technical report, School of Computer Science, University of Waterloo, TR CS-2003-08 (2003)
[14] Henzinger, M., Raghavan, P., Rajagopalan, S.: Computing on data streams. In: "External Memory algorithms". DIMACS series in Discrete Mathematics and Theoretical Computer Science 50, 107–118 (1999)
[15] Jájá, J.: An introduction to parallel algorithms. Addison-Wesley, Reading (1992)
[16] Kushilevitz, E., Nisan, N.: Communication Complexity. Cambr. U. Press (1997)
[17] Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal of Computing 15(4), 1036–1053 (1986)
[18] Munro, I., Paterson, M.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)
[19] Muthukrishnan, S.: Data streams: algorithms and applications. Technical report (2003), Available at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`
[20] Reif, J.: Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR 08-85, Aiken Comp. Lab, Harvard U., Cambridge (1985)
[21] Ruhl, M.: Efficient Algorithms for New Computational Models. PhD thesis, Massauchussets Institute of Technology (September 2003)
[22] Shiloach, Y., Vishkin, U.: An $o(\log n)$ Parallel Connectivity Algorithm. J. Algorithms 3(1), 57–67 (1982)
[23] Sullivan, M., Heybey, A.: Tribeca: A system for managing large databases of network traffic. In: Proceedings USENIX Annual Technical Conference (1998)
[24] Tarjan, R., Vishkin, U.: Finding biconnected components and computing tree functions in logarithmic parallel time. In: Proc. 25th Annual IEEE Symposium on Foundations of Computer Science (FOCS'84), pp. 12–20. IEEE Computer Society Press, Los Alamitos (1984)
[25] Ullman, J., Yannakakis, M.: High-probability parallel transitive-closure algorithms. SIAM Journal on Computing 20(1), 100–125 (1991)