Sorting, Searching, and Simulation in the MapReduce Framework

Michael T. Goodrich[1], Nodari Sitchinava[2], and Qin Zhang[2]

[1] Department of Computer Science, University of California, Irvine, USA
goodrich@ics.uci.edu
[2] MADALGO*, Department of Computer Science, University of Aarhus, Denmark
{nodari, qinzhang}@madalgo.au.dk

Abstract. In this paper, we study the MapReduce framework from an algorithmic standpoint and demonstrate the usefulness of our approach by designing and analyzing efficient MapReduce algorithms for fundamental sorting, searching, and simulation problems. This study is motivated by a goal of ultimately putting the MapReduce framework on an equal theoretical footing with the well-known PRAM and BSP parallel models, which would benefit both the theory and practice of MapReduce algorithms. Our PRAM and BSP simulation results imply efficient MapReduce solutions for many applications, such as sorting, 2- and 3-dimensional convex hulls, fixed-dimensional linear programming. All our algorithms take a constant number of rounds under the commonly made assumptions for the hardware running MapReduce.

1 Introduction

The *MapReduce framework* [5, 6] is a programming paradigm for designing parallel and distributed algorithms. It provides a simple programming interface that is specifically designed to make it easy for a programmer to design a parallel program that can efficiently perform a data-intensive computation. Moreover, it is a framework that allows for parallel programs to be directly translated into computations for cloud computing environments and server clusters (e.g., see [16]). This framework is gaining wide-spread interest in systems domains, in that this framework is being used in Google data centers and as a part of the open-source Hadoop system [19] for server clusters, which have been deployed by a wide variety of enterprises[3], including Yahoo!, IBM, The New York Times, eHarmony, Facebook, and Twitter.

Building on pioneering work by Feldman *et al.* [9] and Karloff *et al.* [14], our interest in this paper is in studying the MapReduce framework from an algorithmic standpoint, by designing and analyzing MapReduce algorithms for fundamental sorting, searching, and simulation problems. Such a study could be a step toward ultimately putting the MapReduce framework on an equal theoretical footing with the well-known PRAM and BSP parallel models.

* MADALGO is the Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

[3] See http://en.wikipedia.org/wiki/Hadoop.

Still, we would be remiss if we did not mention that this framework is not without its detractors. DeWitt and Stonebraker [7] mention several issues they feel are shortcomings of the MapReduce framework, including that it seems to require brute-force enumeration instead of indexing for performing searches. Naturally, we feel that this criticism is a bit harsh, as the theoretical limits of the MapReduce framework have yet to be fully explored; hence, we feel that further theoretical study is warranted. Indeed, this paper can be viewed as at least a partial refutation of the claim that the MapReduce framework disallows indexed searching, in that we show how to perform fast and efficient multi-search in the MapReduce framework.

The MapReduce Framework. In the MapReduce framework, a computation is specified as a sequence of map, shuffle, and reduce steps that operate on a set $X = \{x_1, x_2, \ldots, x_n\}$ of values:

- A *map step* applies a function, μ, to each value, x_i, to produce a finite set of key-value pairs (k, v). To allow for parallel execution, the computation of the function $\mu(x_i)$ must depend only on x_i.
- A *shuffle step* collects all the key-value pairs produced in the previous map step, and produces a set of lists, $L_k = (k; v_1, v_2, \ldots)$, where each such list consists of all the values, v_j, such that $k_j = k$ for a key k assigned in the map step.
- A *reduce step* applies a function, ρ, to each list $L_k = (k; v_1, v_2, \ldots)$, formed in the shuffle step, to produce a set of values, $y_1, y_2, \ldots$. The reduction function, ρ, is allowed to be defined sequentially on L_k, but should be independent of other lists $L_{k'}$ where $k' \neq k$.

The parallelism of the MapReduce framework comes from the fact that each map or reduce operation can be executed on a separate processor independently of others. Thus, the user simply defines the functions μ and ρ, and the system automatically schedules map-shuffle-reduce steps and routes data to available processors, including provisions for fault tolerance.

The outputs from a reduce step can, in general, be used as inputs to another round of map-shuffle-reduce steps. Thus, a typical MapReduce computation is described as a sequence of map-shuffle-reduce steps that perform a desired action in a series of *rounds* that produce the algorithm's output after the last reduce step.

Evaluating MapReduce Algorithms. Ideally, we desire the number of rounds in a MapReduce algorithm to be a constant. For example, consider an often-cited MapReduce algorithm to count all the instances of words in a document. Given a document, D, we define the set of input values X to be all the words in the document and we then proceed as follows: the Map step, for each word, w, in the document, maps w to $(w, 1)$. Then in the Shuffle step, collects all the $(w, 1)$ pairs for each word, producing a list $(w; 1, 1, \ldots, 1)$, with the number of 1's in each such list equal to the number of times w appears in the document. Finally, the Reduce step, scans each list $(w; 1, 1, \ldots, 1)$, summing up the number of 1's in each such list, and outputs pairs (w, n_w) as a final value, where n_w is the number of 1's in the list for each w. This single-round computation clearly computes the number of times each word appears in D.

The number of rounds in a MapReduce algorithm is not always equal to 1, however, and there are, in fact, several metrics that one can use to measure the efficiency of a MapReduce algorithm over the course of its execution, including the following:

- We can consider R, the *number of rounds* of map-shuffle-reduce that the algorithm uses.
- If we let $n_{r,1}, n_{r,2}, \ldots$ denote the mapper and reducer I/O sizes for round r, so that $n_{r,i}$ is the size of the inputs and outputs for mapper/reducer i in round r, then we can define C_r, the *communication complexity of round* r, to be the total size of the inputs and outputs for all the mappers and reducers in round r, that is, $C_r = \sum_i n_{r,i}$. We can also define $C = \sum_{r=0}^{R-1} C_r$ – the *communication complexity* for the entire algorithm.
- We can let t_r denote the *internal running time* for round r, which is the maximum internal running time taken by a mapper or reducer in round r, where we assume $t_r \geq \max_i\{n_{r,i}\}$, since a mapper or reducer must have a running time that is at least the size of its inputs and outputs. We can also define *total internal running time*, $t = \sum_{r=0}^{R-1} t_r$, for the entire algorithm, as well.

We can make a crude calibration of a MapReduce algorithm using the following additional parameters:

- L: the latency L of the shuffle network, which is the number of steps that a mapper or reducer has to wait until it receives its first input in a given round.
- B: the bandwidth of the shuffle network, which is the number of elements in a MapReduce computation that can be delivered by the shuffle network in any time unit.

Given these parameters, a lower bound for the total running time, T, of an implementation of a MapReduce algorithm can be characterized as follows:

$$T = \Omega\left(\sum_{r=0}^{R-1}(t_r + L + C_r/B)\right) = \Omega(t + RL + C/B).$$

For example, given a document D of n words, the simple word-counting MapReduce algorithm given above has a worst-case performance of $R = 1$, $C = \Theta(n)$, and $t = \Theta(n)$; hence, its worst-case time performance $T = \Theta(n)$, which is no faster than sequential computation. Unfortunately, such performance could be quite common in the natural-language documents. For instance, in the Brown Corpus [15], the word "the" accounts for 7% of all word occurrences.

Note, therefore, that focusing exclusively on R, the number of rounds in a MapReduce algorithm, can actually lead to an inefficient algorithm. For example, if we focus only on the number of rounds, R, then the most efficient algorithm would always be the *trivial one-round algorithm*, which maps all the inputs to a single key and then has the reducer for this key perform a standard sequential algorithm to solve the problem. This approach would run in one round, but it would not use any parallelism; hence, it would be relatively slow compared to an algorithm that was more "parallel."

Memory-Bound and I/O-Bound MapReduce Algorithms. So as to steer algorithm designers away from the trivial one-round algorithm, recent algorithmic formalizations of the MapReduce paradigm have focused primarily on optimizing the round complexity bound, R, while restricting the memory size or input/output size for reducers. Karloff *et al.* [14] define their MapReduce model, MRC, so that each reducer's I/O size is restricted to be $\mathcal{O}(N^{1-\epsilon})$ for some small constant $\epsilon > 0$, and Feldman *et al.* [9] define their model, MUD, so that reducer memory size is restricted to be $\mathcal{O}(\log^c N)$, for some constant $c \geq 0$, and reducers are further required to process their inputs in a single pass. These restrictions limit the feasibility of the trivial one-round algorithm for solving a problem in the MapReduce framework and instead compel algorithm designers to make better utilization of parallelism.

In this paper, we follow the I/O-bound approach, as it seems to correspond better to the way reducer computations are specified, but we take a somewhat more general characterization than Karloff *et al.* [14], in that we do not bound the I/O size for reducers explicitly to be $\mathcal{O}(N^{1-\epsilon})$, but instead allow it to be an arbitrary parameter:

– We define M to be an upper bound on the *I/O-buffer memory size* for all reducers used in a given MapReduce algorithm. That is, we predefine M to be a parameter and require that $\forall r, i : \ n_{r,i} \leq M$.

We then can use M in the design and/or analysis of each of our MapReduce algorithms. For instance, if each round of an algorithm has a reducer with an I/O size of at most M, then we say that this algorithm is an *I/O-memory-bound MapReduce algorithm* with parameter M. In addition, if each round has a reducer with an I/O size proportional to M (whose processing probably dominates the reducer's internal running time), then we can give a simplified lower bound on the time, T, for such an algorithm as

$$T = \Omega(R(M + L) + C/B).$$

This approach therefore can characterize the limits of parallelism that are possible in a MapReduce algorithm and it also shows that we should concentrate on the round complexity and communication complexity of a MapReduce algorithm in characterizing its performance.[4] Of course, such bounds for R and C may depend on M, but that is fine, for similar characterizations are common in the literature on external-memory algorithms (e.g., see [1, 3, 4, 18]). In the rest of the paper, when we talk about the MapReduce model, we always mean the I/O-memory-bound MapReduce model.

Our Contributions. In Section 2 we present a BSP-like computational framework which we prove to be equivalent to the I/O-memory-bound MapReduce model. This formulation is more familiar in the distributed algorithms community, making the design and analysis of algorithms more intuitive. The new formulation allows a simple simulation result of the BSP algorithms in the MapReduce model with no slowdown in the number of rounds, resulting in straightforward MapReduce implementations of a large number of existing algorithms for BSP model and its variants.

[4] These measures correspond naturally with the *time* and *work* bounds used to characterize PRAM algorithms (e.g., see [12]).

In Section 3 we present simulation of CRCW PRAM algorithms in our generalized MapReduce model, extending the EREW PRAM simulation results of Karloff et al. [14][5] (which also holds in our generalized model). The only prior known simulation of CRCW PRAM algorithm on MapReduce was via the standard CRCW-to-EREW simulation (which incurs $\mathcal{O}(\log_2 P)$ factor slowdown for a P-processor PRAM algorithm) and then applying the EREW simulation of Karloff et al. [14]. In contrast, our simulation achieves only $\Theta(\log_M P)$ slowdown in the round complexity, which is asymptotically optimal for a generic simulation.

Our CRCW PRAM simulation results achieve their efficiency through the use of an implicit data structure we call *invisible funnel trees*. It can be viewed as placing virtual multi-way trees rooted at the input items, which funnel concurrent read and write requests to the data items, but are never explicitly constructed.

Our simulation results immediately imply solutions with $\mathcal{O}(\log_M N)$ round and $\mathcal{O}(N \log_M N)$ communication complexities to problems of finding *convex hull* and solving *fixed-dimensional linear programming*.

For problems with no known constant time CRCW PRAM solutions we show that we can design efficient algorithms directly in our generic MapReduce framework. Specifically, in Section 4 using the idea of *invisible funnel trees* we develop solutions to the fundamental problems of *prefix sums* and randomized *indexing* of the input.

Finally, what is perhaps most unusual about the MapReduce framework is that there is no explicit notion of "place" for where data is stored nor for where computations are performed. This property is perhaps what led DeWitt and Stonebraker [7] to say that it does not support indexed searches. Nevertheless, in Section 5 we show that the MapReduce framework does in fact support efficient *multi-searching* – the problem of searching for a number of keys in a search tree. Our solution builds a low congestion search structure similar to [10]. However, to keep the communication complexity low, our structure is smaller, forcing us to process the queries in smaller batches, which we pipeline to maintain the optimal round complexity.

For ease of exposition let $\lambda = \log_M N$. All our algorithms exhibit $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities. Note, that in practice it is reasonable to assume that $M = \Omega(N^\epsilon)$ for some constant $\epsilon > 0$, resulting in $\lambda = \mathcal{O}(1)$, i.e. constant round and linear communication complexities for all our algorithms.

2 Generic MapReduce Computations

In this section we define a BSP-like computational model that captures the MapReduce framework.

Consider a set of computing nodes V. Let $A_v(r)$ be a set of items associated with each node $v \in V$ in round r. $A_v(r)$ defines the state of v. Let f be a sequential function defined for all nodes. Function f takes as input the state $A_v(r)$ of a node v and returns a new set $B_v(r)$, in the process destroying $A_v(r)$. Each item of $B_v(r)$ is of the form

[5] Their original proof was identified for the CREW PRAM model, but there was a flaw in that version, which could violate the I/O-buffer-memory size constraint during a CREW PRAM simulation. Based on a personal communication, we have learned that the subsequent version of their paper will identify their proof as being for the EREW PRAM.

(w, a), where $w \in V$ and a is a new item. We define the following computation which proceeds in R rounds.

At the beginning of the computation only the *input nodes* v have non-empty states $A_v(0)$. The state of an input node consists of a single input item.

In round r, each node v with non-empty state $A_v(r) \neq \emptyset$ performs the following. First, v applies function f on $A_v(r)$. This results in the new set $B_v(r)$ and deletion of $A_v(r)$. Then, for each element $b = (w, a) \in B_v(r)$, node v sends item a to node w. Note that if $w = v$, then v sends a back to itself. As a result of this process, each node may receive a set of items from others. Finally, the set of received items at each node v defines the new state $A_v(r+1)$ for the next round. The items comprising the non-empty states $A_v(r)$ after R rounds define the outputs of the entire computation at which point the computation halts.

The number of rounds R denotes the *round complexity* of the computation. The total number of all the items sent (or, equivalently, received) by the nodes in each round r defines the *communication complexity* C_r of round r, that is, $C_r = \sum_v |B_v(r)|$. Finally, the communication complexity C of the entire computation is defined as $C = \sum_{r=0}^{R-1} C_r = \sum_{r=0}^{R-1} \sum_v |B_v(r)|$. Note that this definition implies that nodes v whose states $A_v(r)$ are empty at the beginning of round r do not contribute to the communication complexity. Thus, the set V of nodes can be infinite. But, as long as only a finite number of nodes have non-empty $A_v(r)$ at the beginning of each round, the communication complexity of the computation is bounded.

Observe that during the computation, in order for node v to send items to node w in round r, v should know the label of the destination w, which can be obtained by v in the following possible ways (or any combination thereof): 1) the link (v, w) can be encoded in f as a function of the label of v and round r, 2) some node might send the label of w to v in the previous round, or 3) node v might keep the label of w as part of its state by constantly sending it to itself.

Thus, the above computation can be viewed as a computation on a *dynamic* directed graph $G = (V, E)$, where an edge $(v, w) \in E$ in round r represents a possible communication link between v and w during that round. The encoding of edges (v, w) as part of function f is equivalent to defining an *implicit* graph [13]; keeping all edges within a node throughout the computation is equivalent to defining a *static* graph. For ease of exposition, we define the following primitive operations that can be used within f at each node v:

- create an item; delete an item; modify an item; keep item x (that is, the item x will be sent to v itself by creating an item $(v, x) \in B_v(r)$); send an item x to node w (create an item $(w, x) \in B_v(r)$).
- create an edge; delete an edge. This is essentially the same as create an item and delete an item, since explicit edges are just maintained as items at nodes. This operations will simplify exposition when dealing with explicitly defined graphs G on which computation is performed.

The following theorem shows that the above framework captures the essence of computation in the MapReduce framework.[6]

[6] Due to space constraints, all omitted proofs can be found in Appendix A.

Theorem 1. *Let $G = (V, E)$ and f be defined as above such that in each round each node $v \in V$ sends, keeps and receives at most M items. Then computation on G with round complexity R and communication complexity C can be simulated in the I/O-memory-bound MapReduce model with the same round and communication complexities.*

The above theorem gives an abstract way of designing MapReduce algorithms. More precisely, to design a MapReduce algorithm, we define graph G and a sequential function f to be performed at each node $v \in V$. This is akin to designing BSP algorithms and is a more intuitive way than defining Map and Reduce functions.

Note that in the above framework we can easily implement a global loop primitive spanning over multiple rounds: each item maintains a counter that is updated at each round. We can also implement *parallel tail recursion* by defining the labels of nodes to include the recursive call stack identifiers.

3 Simulation Results

BSP simulation. The reader may observe that the generic MapReduce model of the previous section is very similar to the BSP model of Valiant [17], leading to the following conclusion.

Theorem 2. *Given a BSP algorithm $\mathcal{A}$ that runs in R super-steps with a total memory size N using $P \leq N$ processors, we can simulate $\mathcal{A}$ using R rounds and $C = \mathcal{O}(RN)$ communication in the I/O-memory-bound MapReduce framework with reducer memory size bounded by $M = \lceil N/P \rceil$.*

CRCW PRAM simulation. In this section we present a simulation of f-CRCW PRAM model, the strongest variant of the PRAM model, where concurrent writes to the same memory location are resolved by applying a commutative semigroup operator f on all values being written to the same memory address, such as *Sum*, *Min*, *Max*, etc.

The input to the simulation of a PRAM algorithm $\mathcal{A}$ is specified by an indexed set of P processor items, $p_1, \ldots, p_P$, and an indexed set of initialized PRAM memory cells, $m_1, \ldots, m_N$, where N is the total memory size used by $\mathcal{A}$. For ease of exposition we assume that that $P = N^{\mathcal{O}(1)}$, i.e. $\log_M P = \mathcal{O}(\log_M N) = \mathcal{O}(\lambda)$.

The main challenge in simulating the algorithm $\mathcal{A}$ in the MapReduce model is that there may be as many as P reads and writes to the same memory cell in any given step and P can be significantly larger than M, the memory size of reducers. Thus, we need to have a way to "fan in" these reads and writes. We accomplish this by using *invisible funnel trees*, where we imagine that there is a different implicit $\mathcal{O}(M)$-ary tree rooted at each memory cell that has the set of processors as its leaves. Intuitively, our simulation algorithm involves routing reads and writes up and down these N trees. We view them as "invisible", because we do not actually maintain them explicitly, since that would require $\Theta(PN)$ additional memory cells.

Each invisible funnel tree is an undirected[7] rooted tree $\mathcal{T}$ with branching factor $d = M/2$ and height $L = \lceil \log_d P \rceil = \mathcal{O}(\lambda)$. The root of the tree is defined to be

[7] Each undirected edge is represented by two directed edges.

at level 0 and leaves at level $L-1$. We label the nodes in $\mathcal{T}$ such that the k-th node (counting from the left) on level l is defined as $v = (l, k)$. Then, we can identify the parent of a non-root node $v = (l, k)$ as $p(v) = (l-1, \lfloor k/d \rfloor)$ and the q-th child of v as $w_q = (l+1, k \cdot d + q)$. Thus, given a node $v = (j, (l, k))$, i.e., the k-th node on level l of the j-th tree, we can uniquely identify the label of its parent $p(v)$ and each of its d children and without maintaining the edges explicitly.

At the initialization step, we send m_j to the root node of the j-th tree, i.e., m_j is sent to node $(j, root) = (j, (0, 0))$. For each processor p_i $(1 \leq i \leq P)$, we send π_i – the state of processor p_i to node u_i. Again, throughout the algorithm, each node keeps the items that it has received in previous rounds until they are explicitly deleted.

Each step of the PRAM algorithm $\mathcal{A}$ is specified as a read sub-step, followed by a constant-time internal computation, followed by a write sub-step performed by each of P processors. We show how to simulate each of these sub-steps.

1a. **Bottom-up read phase.** For each processor p_i that attempts to read memory location m_j, node u_i sends an item encoding a read request (in the following we simply say a read request) to the i-th leaf node of the j-th tree, i.e. to node $(j, L-1, i)$, indicating that it would like to read the contents of the j-th memory cell.
For $l = L-1$ downto 1 do:
   – For each node v at level l, if it received read request(s) in the previous round, then it sends a read request to its parent $p(v)$.

1b. **Top-down read phase.** The root node in the j-th tree sends the value m_j to child (j, w_k) if child w_k has sent a read request at the end of the bottom-up read phase.
For $l = 1$ to $L-2$ do:
   – For each node v at level l, if it received m_j from its parent in the previous round, then it sends m_j to all those children who have sent v read requests during the bottom-up read phase. After that v deletes all of its items.

   Each leaf v sends m_j to the node u_i $(1 \leq i \leq P)$ if u_i has sent v a read request at the beginning of the bottom-up read phase. After that v deletes all of its items.

2. **Internal computation phase.** At the end of the top-down phase, each node u_i receives its requested memory item m_j, performs the internal computation, updates the state π_i, and sends an item z encoding a write request to the node $(j, L-1, i)$ if processor p_i wants to write z to the memory cell m_j.

3. **Bottom-up write phase.** For $l = L-1$ downto 0 do:
   – For each node v at level l, if it received write request(s) in the previous round, let $z_1, \ldots, z_k$ $(k \leq d)$ be the items encoding those write requests. If v is not a root, it applies the semigroup function on input $z_1, \ldots, z_k$, sends the result z' to its parent, and then deletes all of its items. Otherwise, if v is a root, it modifies its current memory item to z'.

When we have completed the bottom-up write phase, we are inductively ready for simulating the next step in the PRAM algorithm. We have the following.

Theorem 3. *Given a CRCW PRAM algorithm $\mathcal{A}$ with write conflicts resolved according to a commutative semigroup operator such that $\mathcal{A}$ runs in T steps using P processors and N memory cells, we can simulate $\mathcal{A}$ in the I/O-memory-bound MapReduce framework in the optimal $R = \Theta(\lambda T)$ rounds and with $C = \mathcal{O}(\lambda T(N+P))$ communication complexity.*

Applications. Theorem 2 immediately implies $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexity MapReduce solutions for problems of sorting and computing convex hull via simulation of the corresponding BSP solutions [11, 10]. In Appendix B we present an alternative randomized algorithm for sorting with the same complexity but which might be simpler to implement in practice than the simulation of the complicated BSP algorithm in [11].

By Theorem 3, we can simulate any CRCW (thus, also CREW) PRAM algorithm. For example, simulation of the PRAM algorithm of Alon and Megiddo [2] for linear programming in fixed dimensions produces a MapReduce algorithm with $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities.

4 Prefix Sums and Random Indexing

The best known PRAM algorithm for prefix sums runs in $\mathcal{O}(\log^* N)$ time on Sum-CRCW model [8], resulting in a $\mathcal{O}(\lambda \log^* N)$ MapReduce algorithm (by Theorem 3). In this section, we show how we can improve this result to $\mathcal{O}(\lambda)$ rounds. We use the all-prefix-sum solution to design a random indexing of the input, which will be used in the multi-search algorithm in Section 5.

The all-prefix-sum problem is usually defined on an array of integers. Since there is no notion of arrays in the MapReduce framework, but rather a collection of items, we define the all-prefix-sum problem as follows: given a collection of items x_i, where x_i holds an integer a_i and an index value $0 \leq i \leq N-1$, compute for each item x_i a new value $b_i = \sum_{j=0}^{i} a_j$.

The classic PRAM algorithm for computing prefix sums [12] can be viewed as a computation along a virtual binary tree on top of the inputs. To compute the prefix sums in MapReduce we replace the binary tree with the invisible funnel tree and perform similar 2-pass computation with the details as follows.

In the initialization step, each input node simply sends its input item a_i with index i to the leaf node $v = (L-1, i)$ of the funnel tree. The rest of the algorithm proceeds in two phases, processing the nodes in $\mathcal{T}$ one level at a time. The nodes at other levels simply keep the items they have received during previous rounds.

1. **Bottom-up phase.** For $l = L-1$ downto 1 do: For each node v on level l do: If v is a leaf node, it received a single value a_i from an input node. The function f at v creates a copy $s_v = a_i$, keeps a_i it had received and sends s_v to the parent $p(v)$ of v. If v is a non-leaf node, let $w_0, w_1, \ldots, w_{d-1}$ denote v's child nodes in the left-to-right order. Node v received a set of d items $A_v(r) = \{s_{w_0}, s_{w_1}, \ldots, s_{w_{d-1}}\}$ from its children at the end of the previous round. $f(A_v(r))$ computes the sum $s_v = \sum_{j=0}^{d-1} s_{w_j}$, sends s_v to $p(v)$ and keeps all the items received from the children.
2. **Top-down phase.** For $l = 0$ to $L-1$ do: For each node v on level l do: If v is the root, it had received items $A_v(r) = \{s_{w_0}, s_{w_1}, \ldots, s_{w_{d-1}}\}$ at the end of the bottom-up phase. It creates for each child w_i $(0 \leq i \leq d-1)$ a new item $s'_i = \sum_{j=0}^{i-1} s_{w_j}$ and sends it to w_i. If v is a non-root node, let $s_{p(v)}$ be the item received from its parent in the previous round. Inductively, the value $s_{p(v)}$ is the sum of all items "to the left" of v. If v is a leaf having a unique item a_k, then it simply outputs

$a_k + s_{p(v)}$ as a final value, which is the prefix sum $\sum_{j=0}^{k} a_j$. Otherwise, it creates for each child w_i $(0 \leq i \leq d-1)$ a new item $s_{p(v)} + \sum_{j=0}^{i-1} s_{w_j}$ and sends it to w_i. In all cases, all items of v are deleted.

Lemma 1. *Given an indexed collection of N numbers, we can compute all prefix sums in the I/O-memory-bound MapReduce framework in $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities.*

Quite often, the input to the MapReduce computation is a collection of items with no particular ordering or indexing. If each input element is annotated with an estimate $N \leq \hat{N} \leq N^c$ of the size of the input, for some constants $c \geq 1$, then we can modify the all-prefix-sum algorithm to generate a random indexing for the input with high probability as follows.

We define the invisible funnel tree $\mathcal{T}$ on $\hat{N}^3$ leaves, thus, the height of the tree is $L = \lceil 3 \log_d \hat{N} \rceil = \mathcal{O}(\lambda)$. In the initialization step, each input node picks a random index i in the range $[0, \hat{N}^3 - 1]$ and sends $a_i = 1$ to the leaf node $v = (L-1, i)$ of $\mathcal{T}$. Let n_v be the number of items that leaf v receives. Note it is possible that $n_v > 1$, thus, we perform the all-prefix-sums computation with the following differences at the leaf nodes. During the bottom-up phase, we define $s_v = n_v$ at the leaf node v. At the end of the top-down phase, each leaf v assigns each of the item that it received from the input nodes the indices $s_{p(v)} + 1, s_{p(v)} + 2, \ldots, s_{p(v)} + n_v$ in a random order, which is the final output of the computation.

Lemma 2. *A random indexing of the input can be performed on a collection of data in the I/O-memory-bound MapReduce framework in $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities with high probability.*

5 Multi-searching and Sorting

Let $\mathcal{T}$ be a balanced binary search tree and Q be a set of queries. Let $N = |\mathcal{T}| + |Q|$. The problem of multi-search asks to annotate each query $q \in Q$ with a leaf $v \in \mathcal{T}$, such that the root-to-leaf search path for q in $\mathcal{T}$ terminates at v.

Goodrich [10] provides a solution to the multi-search problem in the BSP model. His solution first converts the binary search tree into a B-tree with the branching parameter $M = \lceil N/P \rceil$, i.e. each node of the B-tree contains $\Theta(M)$ routing entries and is of depth $\Theta(\lambda) = \Theta(\log_M N)$. Then it replicates each node to relieve congestion during query routing by estimating the query load of each node by routing a small sample of the queries down the B-tree. The replicated nodes are connected to others in such a way that the set of nodes reachable from each replicated root node, comprise the skeleton of the original B-tree. Finally, all the queries are distributed randomly across all the copies of the root nodes and propagated down this search structure G to the leaf nodes (and their copies).

The depth of G is $\Theta(\lambda)$ with each level consisting of $\mathcal{O}(|Q|/M)$ B-tree nodes each containing $\Theta(M)$ routing elements. Thus, the size of G is $\mathcal{O}(|\mathcal{T}| + \lambda|Q|)$. And by Theorem 2, we obtain a MapReduce solution to multi-search with $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda|\mathcal{T}| + \lambda^2|Q|) = \mathcal{O}(\lambda^2 N)$ communication complexities.

In this section we present a solution that improves the communication complexity to optimal $\mathcal{O}(\lambda N)$, while still achieving $\mathcal{O}(\lambda)$ round complexity with high probability. Note, that if $|Q| \leq N/\lambda$, then the size of the BSP search structure is only linear with N and we can perform the simulation of the algorithm with $\mathcal{O}(\lambda N)$ communication complexity. Thus, for the remainder of this section we assume that $|Q| > N/\lambda$.

Multi-searching. To solve the multi-search problem in MapReduce with optimal $\mathcal{O}(\lambda N)$ communication complexity, consider a random partition of Q into λ subsets $Q_1, Q_2, \ldots, Q_\lambda$ each containing $\mathcal{O}(N/\lambda)$ queries. By the above discussion, we clearly can construct a search structure G based on the query set Q_1, consisting of $\Theta(\lambda)$ levels each containing $\mathcal{O}(N/\lambda)$ routing elements, i.e. $|G| = \mathcal{O}(N)$. We can also implement a MapReduce algorithm $\mathcal{A}$ which propagates any query set Q' of size $|Q'| = \mathcal{O}(N/\lambda)$ down this search structure G.

To answer the multi-search queries for all queries Q, we proceed in $\Theta(\lambda)$ rounds. In the first λ rounds, in round i, $1 \leq i \leq \lambda$, we feed new subset Q_i of queries to the $\mathcal{O}(N/\lambda)$ root nodes of G and propagate the queries down to the leaves using algorithm $\mathcal{A}$. This approach can be viewed as a pipelined execution of λ multi-searches on G.

Finally, to implement the random partitioning of Q into λ subsets, we perform a random indexing for Q (Lemma 2) and assign query with index j to subset $Q_{\lceil j/\lambda \rceil}$. A node v containing a query $q \in Q_i$ keeps q (by sending it to itself) until round i, at which point it sends q to the appropriate source node of G.

Theorem 4. *Given a binary search tree $\mathcal{T}$ of size N, we can perform a multi-search of N queries over $\mathcal{T}$ in the I/O-memory-bound MapReduce model in $\mathcal{O}(\lambda)$ rounds with $\mathcal{O}(\lambda N)$ communication with high probability.*

Proof (Sketch). Let $L_1, \ldots, L_\lambda$ be the λ levels of nodes of G. First, all query items in the first query batch Q_1 pass (i.e., be routed down) L_j $(1 \leq j \leq \lambda)$ in one round with high probability. This is because for each node v in L_j, at most M query items of Q_1 will be routed to v with probability at least $1 - N^{-c}$ for any constant c. By taking the union of all the nodes in L_j, we have that with probability at least $1 - \mathcal{O}(N/\lambda) \cdot N^{-c}$, Q_1 pass L_j in one round. Similarly, we can prove that any Q_i $(1 \leq i \leq \lambda)$ can pass L_j $(1 \leq j \leq \lambda)$ in one round with the same probability since all sets Q_i have equal distributions. Since there are λ batches of queries and they are fed into G in a pipeline fashion, by union bound we have that with probability at least $1 - \lambda^2 \cdot \mathcal{O}(N/\lambda) \cdot N^{-c} \geq 1 - 1/N$ (by choosing a sufficient large constant c) the whole process completes within $\mathcal{O}(\lambda)$ rounds. The communication complexity follows directly because we only send $\mathcal{O}(|G| + |Q|) = \mathcal{O}(N)$ items in each round.

Applications and discussion. Using the solution to multi-search problem, in Appendix B we present a simple sorting algorithm which might be easier to implement in practice than the simulation of the BSP algorithm from Section 3.

The version of the BSP model used in [10] allows a processor to keep an *unlimited* number of items between rounds while still requiring each processor to send and receive at most $\lceil N/P \rceil = M$ items. A closer inspection of [10] reveals that the probability that some processor will contain more than M items in some round is at most N^{-c} for

any constant $c \geq 1$. Therefore, with high probability it can still be simulated in our MapReduce framework. With some additional work, we can reduce this probability of failure to N^{-cM} with a queuing strategy that we describe in Appendix C. The queuing algorithm might be of independent interest because it removes some of the requirements of the framework of Section 2.

Acknowledgments

We would like to thank Riko Jakob for pointing out the lower bound for our CRCW PRAM simulation.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31, 1116–1127 (1988)
2. Alon, N., Megiddo, N.: Parallel linear programming in fixed dimension almost surely in constant time. J. ACM 41(2), 422–434 (1994)
3. Arge, L.: External-memory algorithms with applications in GIS. In: Algorithmic Foundations of Geographic Information Systems. pp. 213–254. Springer-Verlag (1997)
4. Arge, L.: External memory data structures. In: Handbook of massive data sets, pp. 313–357. Kluwer Academic Publishers, Norwell, MA, USA (2002)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
6. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM 53(1), 72–77 (2010)
7. DeWitt, D.J., Stonebraker, M.: MapReduce: A major step backwards. Database Column (2008), http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/
8. Eisenstat, S.C.: $O(\log^* n)$ algorithms on a Sum-CRCW PRAM. Computing 79(1), 93–97 (2007)
9. Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., Svitkina, Z.: On distributing symmetric streaming computations. In: Teng, S.H. (ed.) SODA. pp. 710–719. SIAM (2008)
10. Goodrich, M.T.: Randomized fully-scalable BSP techniques for multi-searching and convex hull construction. In: SODA. pp. 767–776 (1997)
11. Goodrich, M.T.: Communication-efficient parallel sorting. SIAM Journal on Computing 29(2), 416 – 432 (1999)
12. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading, Mass. (1992)
13. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. In: 20th Annual ACM Symposium on Theory of Computing (STOC). pp. 334–343 (1988)
14. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proc. ACM-SIAM Sympos. Discrete Algorithms (SODA). pp. 938–948 (2010)
15. Kucera, H., Francis, W.N.: Computational Analysis of Present-Day American English. Brown University Press, Providence, RI (1967)
16. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: Supporting MapReduce on large-scale asymmetric multi-core clusters. SIGOPS Oper. Syst. Rev. 43(2), 25–34 (2009)
17. Valiant, L.G.: A bridging model for parallel computation. Comm. ACM 33, 103–111 (1990)
18. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. 33(2), 209–271 (2001)
19. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc. (2009)

A Omitted Proofs

A.1 Proof of Theorem 1

Proof. We implement round $r = 0$ of computation on G in the I/O-memory-bound MapReduce framework using only the Map and Shuffle steps and every round $r > 0$ using the Reduce step of round $r - 1$ and a Map and Shuffle step of round r.

1. Round $r = 0$: (a) Computing $B_v(r) = f(A_v(r))$: Initially, only the input nodes have non-empty sets $A_v(r)$, each of which contains only a single item. Thus, the output $B_v(r)$ only depends on a single item, fulfilling the requirement of Map. We define Map to be the same as f, i.e., it outputs a set of key-value tuples (w, x), each of which corresponds to an item (w, x) in $B_v(r)$. (b) Sending items to destinations: The Shuffle step on the output of the Map step ensures that all tuples with key w will be sent to the same reducer, which corresponds to the node w in G.
2. Round $r > 0$: First, each reducer v that receives a tuple $(v; x_1, x_2, \ldots, x_k)$ (as a result of the Shuffle step of the previous round) simulates the computation at node v in G. That is, it simulates the function f and outputs a set of tuples (w, x), each of which corresponds to an item in $B_v(r)$. We then define Map to be the identity map: On input (w, x), output key-value pair (w, x). Finally, the Shuffle step of round r completes the simulation of the round r of computation on graph G by sending all tuples with key w to the same reducer that will simulate node w in G in round $r+1$.

Keeping an item is equivalent to sending it to itself, thus, each node in G sends and receives at most M items. Therefore, no reducer receives or generates more than M items implying that the above is a correct I/O-memory-bound MapReduce algorithm.

A.2 Proof of Theorem 2: Simulation of BSP Algorithms

Proof. In the BSP model [17], the input of size N is distributed among P processors so that each processor contains at most $M = \lceil N/P \rceil$ input items. A computation is specified as a series of super-steps, each of which involves each processor performing an internal computation and then sending a set of up to M messages to other processors.

The initial state of the BSP algorithm is an indexed set of processors $\{p_1, p_2, \ldots, p_P\}$ and an indexed set of initialized memory cells $\{m_{1,1}, m_{1,2}, \ldots, m_{p,m}\}$, such that $m_{i,j}$ is the j-th memory cell assigned to processor i. Since our framework is almost equivalent to the BSP model, the simulation is straightforward:

– Each processor p_i $(1 \leq i \leq P)$ defines a node v_i in our generic MapReduce graph G, and the internal state π_i of p_i and its memory cells $\{m_{i,1}, \ldots, m_{i,m}\}$ define the items A_{v_i} of node v_i. In the BSP algorithm, in each super-step each processor p_i performs a series of computation, updates its internal state and memory cells to π'_i and $\{m'_{i,1}, \ldots, m'_{i,m}\}$, and sends a set of messages $\mu_{j_1}, \ldots, \mu_{j_k}$ to processors $p_{j_1}, \ldots, p_{j_k}$, where the total size of all messages sent or received by a processor is at most M. In our MapReduce simulation, function f at node v_i performs the same computation, modifies items $\{\pi_i, m_{i,1}, \ldots, m_{i,m}\}$ to $\{\pi'_i, m'_{i,1}, \ldots, m'_{i,m}\}$ and sends items $\mu_{j_1}, \ldots, \mu_{j_k}$ to nodes $v_{j_1}, \ldots, v_{j_k}$.

A.3 Proof for Theorem 3: Simulation of CRCW PRAM Algorithms

Proof. Each parallel step of the CRCW PRAM algorithm is simulated by $\mathcal{O}(\lambda)$ rounds in the I/O-memory-bound MapReduce algorithm, and the total number of items sent in each round is $\mathcal{O}(N + P)$.

Now we show that the round complexity of our simulation result is tight. Consider the problem of summing up N integers. In the Sum-CRCW PRAM model this problem can be solved in a single parallel step with $P = N$ processors. However, we show that it takes at least $\Omega(\log_M N) = \Omega(\lambda)$ rounds to solve this problem in the MapReduce model.

Notice that in order for some node v in our generic MapReduce model to compute the correct answer, it must collect information from each of the N input nodes. The proof is by induction on the size of the input. Assume that for any $n < N$ it takes at least $\log_M n$ rounds to compute the sum of n elements in the MapReduce model (the inductive hypothesis). Clearly, for $n \leq M$ this is true because each input node holds only a single item and needs to send it to v. In the last round, v can collect information only from at most M nodes due to our constraint on the buffer size. By the pigeonhole principle, at least one of these M nodes has already computed the result based on the information from at least $n = N/M$ input nodes. Let u be this node. By our inductive hypothesis it takes at least $\log_M n = \log_M N - 1$ rounds for u to perform this computation. The lower bound follows.

A.4 Proof for Lemma 1

Proof. The fact that the algorithm correctly computes all prefix sums is by induction on the values $s_{p(v)}$. In each round, each node sends and receives at most M items, fulfilling the condition of Theorem 1. The total number of rounds is $2L = \mathcal{O}(\lambda)$ plus the initial round of sending input elements to the leaves of $\mathcal{T}$. The total number of items sent in each round is dominated by items sent by N leaves, which is $\mathcal{O}(N)$ per round. Applying Theorem 1 completes the proof.

A.5 Proof for Lemma 2

Proof. First, note that the probability that $n_v > M$ at some leaf vertex is at most $N^{-\Omega(M)}$. Thus, with probability at least $1 - N^{-\Omega(M)}$, no leaf and, consequently, no node of $\mathcal{T}$ receives more than $\mathcal{O}(M)$ elements. Second, note that at most N leaves of the funnel tree $\mathcal{T}$ have $A_v(r) \neq \emptyset$. Since we do not maintain the edges of the tree explicitly, the total number of items sent in each round is again dominated by the items sent by at most N leaves, which is $\mathcal{O}(N)$ per round. Finally, the round and communication complexity follows from Lemma 1.

B Sorting

In this section, we show how to obtain a simple sorting algorithm in the MapReduce model by using our multi-search algorithm from Section 5.

Again, since there is no notion of arrays in MapReduce, but rather collections of items, we define the problem of *sorting* as follows: given an indexed collection of comparable items X, compute for each item $x_i \in X$ the number of other items in X that are smaller than x_i. For two items x_i and x_j with equal value, we break the tie by the value of their indices i and j, i.e. we can assume that all values are distinct.

First consider the following simple brute-force sorting result:

Lemma 3. *Given a set X of N indexed comparable items, we can sort them in $\mathcal{O}(\lambda)$ rounds and $\mathcal{O}(\lambda N^2)$ communication complexity in the MapReduce model.*

Proof. Consider the following CRCW PRAM algorithm with $P = N^2$ processors.

1. Each processor $p_{i,j}$ $(i \neq j)$ reads item x_i and x_j, compares them and stores in the matrix entry $A[i, j]$ the value 0 if $x_i < x_j$ and value 1 otherwise.
2. For each row i of matrix A, in parallel, the processors $p_{i,1}, \ldots, p_{i,N}$ compute the sum $k_i = \sum_{j=1}^{N} A[i, j]$, which equals the number of items x_j that are smaller than x_i

We can implement the above algorithm in MapReduce in $\mathcal{O}(\lambda)$ rounds and $\mathcal{O}(\lambda N^2)$ communication complexity as follows: The first step is implemented by Theorem 3 (or using the idea of *invisible funnel trees* directly in MapReduce) and the second step by Lemma 1.

Now we are ready to describe a simple randomized sorting algorithm with optimal $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities.

1. Pick $\Theta(\sqrt{N})$ random pivots. Sort the pivots using brute-force sorting algorithm. This results in the pivots being assigned a unique index/label in the range $[1, \sqrt{N}]$.
2. Build a search tree on the set of sorted pivots as the leaves of the tree.
3. Perform a multi-search on the input items over the search tree. The result is the label associated with each item which is equal to the "bucket" within which the input is partitioned into.
4. In parallel, apply steps 1 through 3 to each bucket to obtain the total of $\Theta(N^{3/4})$ buckets.
5. Apply brute force sorting algorithm to each bucket in parallel.

Theorem 5. *A set of N items can be sorted in the MapReduce model in $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexities.*

Proof. The first four steps of the algorithm can be performed using Lemmas 2 and 3 and Theorem 4 in $\mathcal{O}(\lambda)$ rounds and $\mathcal{O}(\lambda N)$ communication complexity.

With high probability, the first application of the first three steps of the algorithm creates buckets of size at most $\tilde{\mathcal{O}}(\sqrt{N})$. The second application creates buckets of size at most $\tilde{\mathcal{O}}(N^{1/4}) = \mathcal{O}(\sqrt{N})$. Thus, the last step of the algorithm runs in $\mathcal{O}(\lambda)$ rounds and $\mathcal{O}(\lambda N)$ communication complexity.

C FIFO Queues in MapReduce Model

As mentioned in Section 5, with probability $1 - N^{-c}$ for any constant $c \geq 1$ no processor in the BSP algorithm for multi-searching contains more than M items. Thus, the algorithm for multi-search in Section 5 can be implemented in the I/O-memory-bound MapReduce framework with high probability.

However, the failure of the algorithm implies a crash of a reducer in the MapReduce framework, which is quite undesirable. In this section we present a queuing strategy which ensures that no reducer receives more than M items, which might be of independent interest.

Consider the following modified version of the generic MapReduce framework from Section 2. In this version we still require each node $v \in V$ to send at most M items. However, instead of limiting the number of items that a node keeps or receives to be M, we only require that in every round at most M different nodes send to any given node v, and function f takes as input a list of at most M items. To accommodate the latter requirement, if a node receives or contains more than M items, the excess items are kept within the node's input buffer and are fed into function f in batches of $\mathcal{O}(M)$ items per round in a first-in-first-out (FIFO) order.

In this section we show that any algorithm $\mathcal{A}$ with round complexity R and communication complexity C in the modified framework can be implemented using the framework in Section 2 with the same asymptotic round and communication complexities.

We simulate algorithm $\mathcal{A}$ by implementing the FIFO queue at each node v by a doubly-linked list L_v of nodes, such that $L_v \cap L_w = \emptyset$ for all $v \neq w$ and $L_v \cap V = \emptyset$ for all $v \in V$. Each node $v \in V$ keeps a pointer $head_{L_v}$ to the head of its list L_v. In addition, v also keeps n_{head}, the number of query items at $head_{L_v}$. If L_v is empty, $head_{L_v}$ points at v and $n_{head} = 0$. Throughout the algorithm we maintain an invariant that for each doubly-linked list L_v, each node in L_v contains $[M/4, M/2]$ query items except the head node, i.e., the one containing the last items to be processed in the queue, which contains at most $M/2$ query items. We simulate one round of $\mathcal{A}$ by the following three rounds. Let $\mathcal{IN}(v)$ and $\mathcal{OUT}(v)$ denote the set of in- and out-neighbors of node $v \in V$, respectively. That is, for each $u \in \mathcal{IN}(v)$, $(u, v) \in E$ and for each $w \in \mathcal{OUT}(v), (v, w) \in E$.

R1. Each node $u \in V$ that wants to send $n_{u,v}$ query items to $v \in \mathcal{OUT}(u)$, instead of sending the actual query items, sends $n_{u,v}$ to v.

R2. Each node $v \in V$ receives a set of different values $n_{u_1,v}, n_{u_2,v}, \ldots, n_{u_k,v}$ from its in-neighbors $u_1, u_2, \ldots, u_k$ $(k \leq M)$. For convenience we define $n_{u_0,v} \triangleq n_{head}$. Next, v partitions the set $\{0, 1, \ldots, k\}$ into sets $S_1, \ldots, S_m$, $m \leq k$, such that $M/4 \leq \sum_{j \in S_i} n_{u_j,v} \leq M/2$ for all $1 \leq i \leq m-1$ and $\sum_{j \in S_m} n_{u_j,v} \leq M/2$. W.l.o.g., assume that $0 \in S_1$. For each S_i, we will have a corresponding node w_i in the list L_v: We let $w_1 = head_{L_v}$ and for each S_i, $2 < i \leq m$ we pick a new node w_i, create edges (w_i, w_{i-1}) and (w_{i-1}, w_i), and send it to nodes w_i and w_{i-1}, respectively. For each $j \in S_i$, we also notify u_j that it should send all its queries to w_i by sending the label of w_i to u_j. The only exception to this rule is that if $w_1 \neq v$ and w_1 contains the edge (w_1, v), i.e. it is the first node in L_v. In this case,

for each $j \in S_1$ each u_j should send queries directly to v. Finally, we update the pointer $head_{L_v}$ to point to w_m and update $n_{head} = \sum_{j \in S_m} n_{u_j,v}$, unless $w_m = v$, in which case $n_{head} = 0$.

R3. Each node $u_j \in \mathcal{IN}(v)$ receives the label of a node w_i from v in the previous rounds. It sends all its query items to w_i. Note that if $w_i = v$, all items will be sent to v directly. At the same time, each node $w \notin V$, i.e. $w \in L_v$, that has an edge (w, v) for some $v \in V$ sends all its items to v and extracts itself from the list. The node w accomplishes this by deleting all edges incident to w and by sending to its predecessor pred(w) in the queue L_v a new edge $(\text{pred}(w), v)$, thus, linking the rest of the queue to v.

Theorem 6. *Let $\mathcal{A}$ be an algorithm in the modified MapReduce framework, where in every round each node is required to send at most M items, but is allowed to keep and receive an unlimited number of items as long as they arrive from at most M different nodes, with excess items stored in FIFO input buffer and fed into function f in blocks of size at most M. If $\mathcal{A}$ runs in R round complexity and C communication complexity in the modified framework, then we can implement $\mathcal{A}$ in the original I/O-memory-bound MapReduce framework in $\mathcal{O}(R)$ rounds and $\mathcal{O}(C)$ communication complexity.*

Proof. First, it is easy to see that our simulation ensures that each node keeps as well as sends and receives at most M items. Next, note that in every three rounds (round $3t, 3t+1, 3t+2$), each node $v \in V$ routes $\min\{\Theta(M), k_v^t\}$ items, where k_v^t is the combined number of items in the queue L_v and the number of items that v's in-neighbors send to v during the three rounds. This is within a constant factor of the number of items that v routes in round t in algorithm $\mathcal{A}$. Finally, the only additional items we send in each round are the edges of the queues $\{L_v \mid v \in V\}$. Note that we only need to maintain $\mathcal{O}(1)$ additional edges for each node of each L_v. And since these nodes are non-empty, the additional edges do not contribute more than a constant factor to the communication complexity.

Applications. The DAG G of the multi-search BSP algorithm [10] satisfies the requirement that at most M nodes attempt to send items to any other node. In addition, if some processor of the BSP algorithm happens to keep more than M items, the processing of these items is delayed and can be processed in any order, including FIFO. Thus, the requirements of Theorem 6 are satisfied.

We do not know how to modify our random indexing algorithm in Section 4 to fit the modified framework. Thus, we cannot provide a Las Vegas algorithm. However, the above framework reduces the probability of failure from $N^{-\Omega(1)}$ to the probability of failure of the random indexing step, i.e., $N^{-\Omega(M)}$, which is much smaller for large values of M.

The modified framework might be of independent interest because it allows for an alternative way of designing algorithms for MapReduce. In particular, it removes the burden of keeping track of the number of items kept or sent by a node.