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. We study the MapReduce framework from an algorithmic standpoint, providing a generalization of the previous algorithmic models for MapReduce. We present optimal solutions for the fundamental problems of all-prefix-sums, sorting and multi-searching. Additionally, we design optimal simulations of the the well-established PRAM and BSP models in MapReduce, immediately resulting in optimal solutions to the problems of computing fixed-dimensional linear programming and 2-D and 3-D convex hulls.

1 Introduction

MapReduce [2,3] is a programming paradigm for designing parallel and distributed algorithms. Building on pioneering work by Feldman *et al.* [6] and Karloff *et al.* [12], our interest in this paper is in studying the MapReduce framework from an algorithmic standpoint. 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 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*.

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

T. Asano et al. (Eds.): ISAAC 2011, LNCS 7074, pp. 374–383, 2011.

Evaluating MapReduce Algorithms. Since the shuffle step requires communication among computers over the network, we desire to minimize the number of rounds in a MapReduce algorithm (ideally, keeping it to a constant). However, there are 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* 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 define the *communication complexity of round* r as $C_r = \sum_i n_{r,i}$. 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 $t_r \geq \max_i\{n_{r,i}\}$.We can also define *total internal running time*, $t = \sum_{r=0}^{R-1} t_r$, for the entire algorithm.

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

- L: the latency L of the shuffle network: 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: 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:

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

Memory-Bound and I/O-Bound MapReduce Algorithms. Note, that MapReduce allows design of trivial one-round algorithms that are actually sequential by mapping inputs to a single key and then implementing a sequential algorithm on the reducer with that key. To steer the programmers away from such sequential implementations, recent algorithmic formalizations of the MapReduce paradigm have focused primarily on optimizing the round complexity, R, while restricting the memory size or input/output size for reducers. Karloff *et al.* [12] define their MapReduce model, MRC, so that each reducer's I/O size is restricted to be $\mathcal{O}(N^{1-\epsilon})$ for some constant $0 < \epsilon < 1$, and Feldman *et al.* [6] 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.

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.* [12], 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 can use M in the design and/or analysis of 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 and we can give a simplified lower bound on the time, T, for such an algorithm as

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

Our Contributions. In Section 2 we present a BSP-like [13] 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 the BSP model and its variants.

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. [12][1] (which also holds in our generalized model). 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.

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 [4] 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 of roughly equal size.

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 Algorithmic Framework for I/O-memory-bound MapReduce

In this section we define a graph-based framework that can be implemented in the I/O-memory-bound MapReduce model with the same round and communication complexities and makes algorithm design simpler and more intuitive.

[1] 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.

Consider a set V of computing nodes. Let $A_v(r)$ be a set of items, each of constant size (in words), 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 the same 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 $A_v(r)$.[2] Each item of $B_v(r)$ is of the form (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, defines 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 (in words), 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 much larger than the size of the input. But, as long as only a small number of them has non-empty $A_v(r)$ at the beginning of each round, the communication complexity of the computation is bounded. Let K_{max} be the upper bound on such nodes in any round.

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 [11]; 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 $(w, x) \in B_v(r)$).

[2] Note that while f is defined the same for all nodes, since it takes $A_v(r)$ and, consequently, v as input, the output of f may vary at different nodes.

– create an edge; delete an edge. This is essentially the same as creating an item and deleting an item, since explicit edges are just maintained as items at nodes. These 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.

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 using K_{max} mappers/reducers.*

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 implying that the above is a correct I/O-memory-bound MapReduce algorithm. □

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 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 [13], leading to the following conclusion. [3]

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 $M = \lceil N/P \rceil$ using $\mathcal{O}(P)$ mappers/reducers.*

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, such as *Sum*, *Min*, *Max*, etc., to all values being written to the same memory address.

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 $P = N^{\mathcal{O}(1)}$, 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 that has the set of processors as its leaves and is rooted at each memory cell. 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[4] 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 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 and starting with index 0) 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.

[3] Due to space constraints, all omitted proofs can be found in the full version of the paper [9].

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

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 $(1 \le i \le P)$ 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 $z_1, \ldots, z_k$ $(k \le d)$ 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. In each round at most $\mathcal{O}(N+P)$ nodes are non-empty, thus, 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 using $\mathcal{O}(N+P)$ mappers/reducers.*

Applications. Theorem 2 immediately implies $\mathcal{O}(\lambda)$ round and $\mathcal{O}(\lambda N)$ communication complexity MapReduce solutions for problems of sorting and computing 2-dimensional convex hull via simulation of the BSP solutions [8,7]. In the full version of the paper [9] 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 [8].

By Theorem 3, we can simulate any CRCW (thus, also CREW) PRAM algorithm. For example, simulation of the PRAM algorithm of Alon and Megiddo [1] 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 [5], 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.

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 $b_i = \sum_{j=0}^{i} a_j$.

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.*

Proof (Sketch). The classic PRAM algorithm for computing prefix sums [10] 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. The details of the algorithm are straightforward and are presented in the full version of the paper [9]. The depth of the invisible funnel tree is λ, resulting in the stated round and 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 constant $c \geq 1$, then we can modify the all-prefix-sum algorithm to generate a random indexing for the input with high probability as follows.[5]

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 $\mathcal{T}$, and performing prefix sums computation on these values. Note that some leaf nodes might receive more than one item, so we make straightforward modifications to the prefix sums algorithm to address this complication (see [9] for details).

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.

[5] Throughout the paper, when we say an event holds with high probability we mean the probability is at least $1 - 1/N$.

Goodrich [7] 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 $M = \lceil N/P \rceil$ (P is the number of BSP processors) in a natural way, i.e. each node of the B-tree corresponds to a subtree of the original binary tree of height $\Theta(\log_2 M)$. The height of the B-tree is $\Theta(\lambda) = \Theta(\log_M N)$. 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 comprises the "skeleton" of the original B-tree (see Goodrich [7] for details). Call the resulting graph G. Finally, all the queries are distributed randomly across all the copies of the root nodes and propagated down in 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 $\mathcal{O}(\lambda N)$, while still achieving $\mathcal{O}(\lambda)$ round complexity; with high probability.

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 round i, $1 \le i \le \lambda$, 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 passes (i.e., routed down) L_j $(1 \le j \le \lambda)$ in one round with probability at least $1 - \mathcal{O}(N/\lambda) \cdot N^{-c}$. 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 similar analysis as in [7]). Thus v can send all those queries to the next level in the next round without violating the output constraint. The statement follows by taking the union of all the nodes in L_j. Similarly, we can prove that any Q_i $(1 \le i \le \lambda)$ can pass L_j $(1 \le j \le \lambda)$ 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 pipelined fashion, by union bound it follows that with probability at least $1 - \lambda^2 \cdot \mathcal{O}(N/\lambda) \cdot N^{-c} \geq 1 - 1/N$ (by choosing a sufficiently 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. The solution to the multi-search problem combined with Theorem 2 immediately implies a solution to the problem of 3-dimensional convex hull via simulation of the BSP solution [7]. In the full version of the paper [9] we also present a simple sorting algorithm which uses the multi-searching solution and is easier to implement in practice than the direct simulation of the BSP sorting algorithm [8] using Theorem 2.

In [9] we also describe a queuing strategy that reduces the failure probability of Theorem 4 from $1/N$ to $N^{-\Omega(M)}$. The queuing algorithm may be of independent interest because it removes some of the constraints 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. Alon, N., Megiddo, N.: Parallel linear programming in fixed dimension almost surely in constant time. J. ACM 41(2), 422–434 (1994)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
3. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM 53(1), 72–77 (2010)
4. DeWitt, D.J., Stonebraker, M.: MapReduce: A major step backwards. Database Column (2008), `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`
5. Eisenstat, S.C.: $O(\log^* n)$ algorithms on a Sum-CRCW PRAM. Computing 79(1), 93–97 (2007)
6. 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)
7. Goodrich, M.T.: Randomized fully-scalable BSP techniques for multi-searching and convex hull construction. In: SODA, pp. 767–776 (1997)
8. Goodrich, M.T.: Communication-efficient parallel sorting. SIAM Journal on Computing 29(2), 416–432 (1999)
9. Goodrich, M.T., Sitchinava, N., Zhang, Q.: Sorting, searching, and simulation in the mapreduce framework (2011), `http://arxiv.org/abs/1101.1902`
10. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
11. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. In: 20th Annual ACM Symposium on Theory of Computing (STOC), pp. 334–343 (1988)
12. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proc. ACM-SIAM Sympos. Discrete Algorithms (SODA), pp. 938–948 (2010)
13. Valiant, L.G.: A bridging model for parallel computation. Comm. ACM 33, 103–111 (1990)