Sorting, Searching, and Simulation in the MapReduce Framework

Michael T. Goodrich¹, Nodari Sitchinava², and Qin Zhang²

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 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, \( \mu \), 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, \( \rho \), 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, \( \rho \), 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.
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. define their MapReduce model, MRC, so that each reducer’s I/O size is restricted to be $O(N^{1-\epsilon})$ for some constant $0 < \epsilon < 1$, and Feldman et al. define their model, MUD, so that reducer memory size is restricted to be $O(\log^c N)$, 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. in that we do not bound the I/O size for reducers explicitly to be $O(N^{1-\epsilon})$, 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 $\forall r, i : n_{r,i} \leq M$. 