A Model of Computation for MapReduce

Howard Karloff* Siddharth Suri† Sergei Vassilvitskii‡

Abstract

In recent years the MapReduce framework has emerged as one of the most widely used parallel computing platforms for processing data on terabyte and petabyte scales. Used daily at companies such as Yahoo!, Google, Amazon, and Facebook, and adopted more recently by several universities, it allows for easy parallelization of data intensive computations over many machines. One key feature of MapReduce that differentiates it from previous models of parallel computation is that it interleaves sequential and parallel computation. We propose a model of efficient computation using the MapReduce paradigm. Since MapReduce is designed for computations over massive data sets, our model limits the number of machines and the memory per machine to be substantially sublinear in the size of the input. On the other hand, we place very loose restrictions on the computational power of of any individual machine—our model allows each machine to perform sequential computations in time polynomial in the size of the original input.

We compare MapReduce to the PRAM model of computation. We prove a simulation lemma showing that a large class of PRAM algorithms can be efficiently simulated via MapReduce. The strength of MapReduce, however, lies in the fact that it uses both sequential and parallel computation. We demonstrate how algorithms can take advantage of this fact to compute an MST of a dense graph in only two rounds, as opposed to $\Omega(\log(n))$ rounds needed in the standard PRAM model. We show how to evaluate a wide class of functions using the MapReduce framework. We conclude by applying this result to show how to compute some basic algorithmic problems such as undirected s-t connectivity in the MapReduce framework.

1 Introduction

In a world in which large data sets are measured in tera- and petabytes, a new form of parallel computing has emerged as an easy-to-program, reliable, and distributed paradigm to process these massive quantities of available data. The MapReduce framework was originally developed at Google [4], but has recently seen wide adoption and has become the *de facto* standard for large scale data analysis. Publicly available statistics indicate that MapReduce is used to process more than 10 petabytes of information per day at Google alone [5]. An open source version, called Hadoop, has recently been developed, and is seeing increased adoption both in industry and academia [14]. Over 70 companies use Hadoop including Yahoo!, Facebook, Adobe, and IBM [8]. Moreover, Amazon's Elastic Compute Cloud (EC2) is a Hadoop cluster where users can upload large data sets and rent processor time. In addition, at least seven universities (including CMU, Cornell, and the University of Maryland) are using Hadoop clusters for research [8].

MapReduce is substantially different from previously analyzed models of parallel computation because it interleaves parallel and sequential computation. In recent years several nontrivial MapReduce algorithms have emerged, from computing the diameter of a graph [9] to implementing the EM algorithm to cluster massive data sets [3]. Each of these algorithms gives some insights into what can be done in a MapReduce framework, however, there is a lack of rigorous algorithmic analyses of the issues involved. In this work we begin by presenting a formal model of computation for MapReduce and compare it to the popular PRAM model. We show that a large subclass of PRAM algorithms, namely those using $O(n^{2-\epsilon})$ processors and $O(n^{2-\epsilon})$ total memory, for a fixed $\epsilon > 0$, can be efficiently simulated in MapReduce. We conclude by demonstrating two basic techniques for parallelizing using MapReduce and show their applications by presenting algorithms for MST in dense graphs and undirected s-t connectivity.

1.1 MapReduce Basics

In the MapReduce programming paradigm, the basic unit of information is a $\langle key; value\rangle$ pair where each *key* and each *value* are binary strings. The input to any MapReduce algorithm is a set of $\langle key; value\rangle$ pairs. Operations on a set of pairs occur in three stages: the map stage, the shuffle stage and the reduce stage, which we discuss in turn.

In the map stage, the mapper μ takes as input a *single* $\langle key; value\rangle$ pair, and produces as output any

*AT&T Labs—Research, howard@research.att.com
†Yahoo! Research, suri@yahoo-inc.com
‡Yahoo! Research, sergei@yahoo-inc.com

number of new $\langle key; value\rangle$ pairs. It is crucial that the map operation is stateless—that is, it operates on one pair at a time. This allows for easy parallelization as different inputs for the map can be processed by different machines.

During the shuffle stage, the underlying system that implements MapReduce sends all of the values that are associated with an individual key to the same machine. This occurs automatically, and is seamless to the programmer.

In the reduce stage, the reducer ρ takes all of the values associated with a single key k, and outputs a multiset of $\langle key; value\rangle$ pairs with the same key, k. This highlights one of the sequential aspects of MapReduce computation: *all* of the maps need to finish before the reduce stage can begin.

Sine the reducer has access to all the values with the same key, it can perform sequential computations on these values. In the reduce step, the parallelism is exploited by observing that reducers operating on different keys can be executed simultaneously.

Overall, a program in the MapReduce paradigm can consist of many rounds of different map and reduce functions, performed one after another.

1.2 MapReduce Example To better understand the power of the model, consider the following simple example of computing the k-th frequency moment of a large data (multi)-set. Let x be the input string of length n, and denote by x_i the i^{th} symbol in x. To represent the input as a sequence of $\langle key; value\rangle$ pairs, we look at x as a sequence of n pairs, $\langle i; x_i\rangle$. Let \$ be a special symbol. The program is as follows:

1. Begin by mapping every tuple to a pair with the symbol as the key, and the position as the value. Thus the first mapper, μ_1 is defined as: $\mu_1(\langle i; x_i\rangle) = \langle x_i; i\rangle$.
2. After the aggregation by the key, the input to each reducer will be a unique string symbol, and the list of positions in which this symbol appears. We proceed to collapse that list into a single number, defining the first reducer ρ_1 as $\rho_1(\langle x_i; \{v_1, \ldots, v_m\}\rangle) = \langle x_i; m^k\rangle$.
3. At this point we just want to sum the number of remaining pairs. First, map each pair to have the same key: $\mu_2(\langle x_i; v\rangle) = \langle \$; v\rangle$.
4. Since all of the pairs now have the same key, they will all be mapped to the same reducer. We can simply sum them: $\rho_2(\langle \$; \{v_1, \ldots, v_l\}\rangle) = \langle \$; \sum_i v_i\rangle$.

Another major attraction of MapReduce, besides its ease of parallelization, is its ease of use. As the example shows, the framework shields the programmer from the low-level details of parallel programming such as fault tolerance, data distribution, scheduling, etc. Also, all of the data shuffling and aggregation are handled by the underlying system itself. The programmer only needs to specify the *map* and *reduce* functions; the system-level issues are handled by the underlying implementation.

The drawback of this model is that in order to achieve this parallelizability, programmers are restricted to using only map and reduce functions in their programs [4]. Thus, this model trades off programmer flexibility for ease of parallelization. This is a complex tradeoff, and it is not *a priori* clear which problems can be efficiently solved in the MapReduce paradigm. The main contribution of this work is a model for what is efficiently computable in the MapReduce paradigm.

2 The MapReduce Programming Paradigm

In this section we give a more formal definition of the MapReduce programming paradigm. We begin by defining mappers and reducers. We then describe how the system executes these two functions along with the shuffle step. As mentioned above, the fundamental unit of data in map reduce computations is the $\langle key; value\rangle$ pair, where keys and values are always just binary strings.

DEFINITION 2.1. *A* mapper *is a (possibly randomized) function that takes as input* one *ordered* $\langle key; value\rangle$ *pair of binary strings. As output the mapper produces a finite multiset of new* $\langle key; value\rangle$ *pairs.*

It is important that the mapper operates on one $\langle key; value\rangle$ pair at a time.

DEFINITION 2.2. *A* reducer *is a (possibly randomized) function that takes as input a binary string* k *which is the key, and a sequence of values* $v_1, v_2, \ldots$ *which are also binary strings. As output, the reducer produces a multiset of pairs of binary strings* $\langle k; v_{k,1}\rangle, \langle k; v_{k,2}\rangle, \langle k; v_{k,3}\rangle, \ldots$. *The key in the output tuples is identical to the key in the input tuple.*

One simple consequence of these two definitions is that that mappers can manipulate keys arbitrarily, but reducers cannot change the keys at all.

Next we describe how the system executes MapReduce computations. A map reduce program consists of a sequence $\langle \mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R\rangle$ of mappers and reducers. The input is a multiset of $\langle key; value\rangle$ pairs denoted by U_0. To execute the program on input U_0:

For $r = 1, 2, \ldots, R$, do:

1. EXECUTE MAP: Feed each pair $\langle k; v\rangle$ in U_{r-1} to mapper μ_r, and run it. The mapper will generate

a sequence of tuples, $\langle k_1; v_1\rangle, \langle k_2; v_2\rangle, \ldots$. Let U'_r be the multiset of $\langle key; value\rangle$ pairs output by μ_r, that is, $U'_r = \bigcup_{\langle k;v\rangle \in U_{r-1}} \mu_r(\langle k; v\rangle)$.

2. SHUFFLE: For each k, let $V_{k,r}$ be the multiset of values v_i such that $\langle k; v_i\rangle \in U'_r$. The underlying MapReduce implementation constructs the multisets $V_{k,r}$ from U'_r.

3. EXECUTE REDUCE: For each k, feed k and some *arbitrary* permutation of $V_{k,r}$ to a separate instance of reducer ρ_r, and run it. The reducer will generate a sequence of tuples $\langle k; v'_1\rangle, \langle k; v'_2\rangle, \ldots$. Let U_r be the multiset of $\langle key; value\rangle$ pairs output by ρ_r, that is, $U_r = \bigcup_k \rho_r(\langle k; V_{k,r}\rangle)$.

The computation halts after the last reducer, ρ_R, halts.

As stated before, the main benefit of this programming paradigm is the ease of parallelization. Since each mapper μ_r only operates on one tuple at a time, the system can have many instances of μ_r operating on different tuples in U_{r-1} in parallel. After the map step, the system partitions the set of tuples output by various instances of μ_r based on their key. That is, part i of the partition has all $\langle key; value\rangle$ pairs that have key k_i. Since reducer ρ_r only operates on one part of this partition, the system can have many instances of ρ_r running on different parts in parallel.

3 The MapReduce Class ($\mathcal{MRC}$)

In this section we formally define the MapReduce Class ($\mathcal{MRC}$). There are three guiding principles that we wish to enforce in our definitions:

Memory Since MapReduce allows for computation to be executed on parts of the input in parallel, the full power of the programming paradigm is realized when the input is too big to fit into memory on a single machine. Thus we require that the input to any mapper or reducer be substantially sublinear in the size of the data. Otherwise, every problem in $\mathcal{P}$ could be solved in the MapReduce formulation by first mapping the whole input to a single reducer, and then having the reducer solve the problem itself.

Machines In order for the model to have practical relevance, we also limit the total number of available machines. For example, an algorithm requiring n^3 machines, where n is the size of the Web, will not be practical in the near future. We limit the total number of machines available to be substantially sublinear in the data size.

Time Finally, there is a question of the total running time available. In a major difference from previous work [6], we do not restrict the power of the individual reducer, except that we require that both the map and the reduce functions run in time polynomial in the original input length in order to ensure efficiency. Furthermore, we will only consider programs that require a small number of Map Reduce rounds, because shuffling is a time consuming operation.

We are now ready to formally define the class $\mathcal{MRC}$. The *input* is a finite sequence of pairs $\langle k_j; v_j\rangle$, for $j = 1, 2, 3, ...$, where k_j and v_j are binary strings. The length of the input is $n = \sum_j (|k_j| + |v_j|)$.

DEFINITION 3.1. *Fix an $\epsilon > 0$. An algorithm in $\mathcal{MRC}^i$ consists of a sequence $\langle \mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R\rangle$ of operations which outputs the correct answer with probability at least 3/4 where:*

- *Each μ_r is a randomized mapper implemented by a RAM with $O(\log n)$-length words, that uses $O(n^{1-\epsilon})$ space and time polynomial in n.*
- *Each ρ_r is a randomized reducer implemented by a RAM with $O(\log n)$-length words, that uses $O(n^{1-\epsilon})$ space and time polynomial in n.*
- *The total space $\sum_{\langle k;v\rangle \in U'_r} (|k| + |v|)$ used by $\langle key; value\rangle$ pairs output by μ_r is $O(n^{2-2\epsilon})$.*
- *The number of rounds $R = O(\log^i n)$.*

We note that while technically RAMs produce a sequence of $\langle key; value\rangle$ pairs as output, we interpret the sequence as a multiset of the corresponding pairs.

We allow the use of randomization in $\mathcal{MRC}$, and demand the final correct answer with probability at least 3/4, but there are obvious Las Vegas and deterministic variants, the latter of which we call $\mathcal{DMRC}$.

We emphasize that mappers process pairs one at a time, and remember nothing about the previous pairs. It also is important to remember that each reducer gets a sequence of values, in some arbitrary (not random) order. Nonetheless the output of the reducer must be correct, or must be correct with a certain probability if the algorithm is randomized, regardless of the order. Note that both mappers and reducers run in time polynomial in n, not polynomial in the length of the input they receive.

At this point a careful reader may complain that the example algorithm given in Section 1 does not fit into this model. Indeed, if the string consists of n copies of the same symbol, then the input to a single reducer will be at least n, in violation of the space constraints in the model. We give an $\mathcal{MRC}$ algorithm for the frequency moments problem in Section 6.1.1.

3.1 Discussion We now pause to justify some of the modeling decisions made above.

3.1.1 Machines As we argued before, it is unrealistic to assume that there are a linear number of machines available when n large. As such we assume that the total number of machines available is $\Theta(n^{1-\epsilon})$. We admit that algorithms with too small an ϵ will be impractical should n be large, but it seems unnatural to to tie one's hands by limiting the number of machines to an arbitrary bound of say, $O(n^{1/2})$.

Recall that each key gets mapped to a unique reducer instance. Since the total number of distinct keys may be as large as $O(n^{2-2\epsilon})$, the total number of reduce instances may be just as large. Therefore more than one instance of a reducer may be run on the same machine.

3.1.2 Memory Restrictions As a consequence of mappers and reducers running on physical machines, the total space available to any map or reduce computation is $O(n^{1-\epsilon})$. One important consequence of this memory restriction is that the size of every $\langle key; value\rangle$ pair must be $O(n^{1-\epsilon})$.

Another consequence of this memory restriction is that the overall amount of memory available across all machines in the system is $O(n^{2-2\epsilon})$. Because the reducers cannot begin executing until after the last mapper has finished, the key value pairs output by the mappers have to be stored temporarily. Thus, the total space taken by all of the $\langle key; value\rangle$ pairs in U'_r must be $O(n^{2-2\epsilon})$. That the total memory available, across all machines, is $O(n^{2-2\epsilon})$ allows one to duplicate the input somewhat, but not absurdly—one is not restricted to simply partitioning the input.

In contrast, mappers operate on one tuple at a time, and therefore they can execute on tuples immediately as they are emitted by the reducers. As such, there is no space restriction on the total size of the output of the reducers.

3.1.3 Shuffle Step In the shuffle step the system partitions the tuples across the $\Theta(n^{1-\epsilon})$ machines so that all of the pairs with the same key go to the same machine. This allows for the reducer to be executed on that machine. Observe that two pairs $(k, V_{k,r})$, $(k', V_{k',r})$, with $k' \neq k$, may be sent to the same machine to be executed sequentially by different reduce instances. The system must ensure that the memory of no machine is exceeded. Next we prove that the space restrictions in Definition 3.1 allow the shuffle step to place all of the values associated with a key on one machine without violating the memory constraints.

LEMMA 3.1. *Consider round r of the execution of an algorithm in $\mathcal{MRC}$. Let K_r be the set of keys in U'_r, let V_r be the multiset of values in U'_r, and let $V_{k,r}$ denote the multiset of values in U'_r that have key k.*

Then K_r and V_r can be be partitioned across $\Theta(n^{1-\epsilon})$ machines such that all machines get $O(n^{1-\epsilon})$ bits, and the pair $\langle k, V_{k,r}\rangle$ gets sent to the same machine.

Proof. For a set of binary strings B denote by $s(B) = \sum_{b\in B} |b|$ the total space used by the strings in B. Since the algorithm is in $\mathcal{MRC}$, by definition, $s(V_r)+s(K_r) \leq s(U'_r) = O(n^{2-2\epsilon})$. Furthermore, the space of the reducer is restricted to $O(n^{1-\epsilon})$; therefore for all k, $|k| + s(V_{k,r})$ is $O(n^{1-\epsilon})$.

Using Graham's greedy algorithm for the minimum makespan scheduling problem [7, 13], we can conclude that the maximum number of bits mapped to any one machine is no more than the average load per machine plus the maximum size of any $\langle k, V_{k,r}\rangle$ pair. Thus,

$$\begin{aligned}
&\leq \frac{s(V_r) + s(K_r)}{\text{number of machines}} + \max_{k\in K_r}(|k| + s(V_{k,r})) \\
&\leq \frac{O(n^{2-2\epsilon})}{\Theta(n^{1-\epsilon})} + O(n^{1-\epsilon}) \\
&\leq O(n^{1-\epsilon}).
\end{aligned}$$

□

We emphasize that every memory restriction in Definition 3.1 is necessary for execution of the shuffle step.

3.1.4 Time Restrictions Just as one can complain that ϵ may be too small, resulting in impractical algorithms, one can justifiably object that allowing arbitrary polynomial time per mapper and reducer is unreasonable. Our goal in defining MRC is to rigorously define the limitations imposed on the algorithm designer by the MapReduce paradigm. Just as before, we admit that algorithms with polynomial running times of too high a degree will be impractical should n be large. However, it seems unnatural to limit the running time to an arbitrary bound of, say, $O(n \log n)$.

Finally, the time per MapReduce round in practice can be large, so it is important to dramatically limit the number of rounds. In fact, we strive to find algorithms in $\mathcal{MRC}^0$, but will show that there are many nontrivial algorithms in $\mathcal{MRC}^1$.

4 Related Work

We begin by comparing the MapReduce framework with other models of parallel computation. After that we discuss other works that use MapReduce.

4.1 Comparing MapReduce and PRAMs Numerous models of parallel computation have been proposed in the literature; see [1] for a survey of them. While the most popular by far for theoretical study is the PRAM, probably the next two most popular are LogP, proposed by Culler et al. [2], and BSP, proposed by Valiant [12]. These three models are all architecture independent. Other researchers have studied architecture-dependent models, such as the fixed-connection network model described in [10].

Since the most prevalent model in theoretical computer science is the PRAM, it seems most appropriate to compare our MapReduce model to it. In a PRAM, an arbitrary number of processors, sharing an unboundedly large memory, operate synchronously on a shared input to produce some output. Different variants of PRAMs deal differently with issues of concurrent reading and concurrent writing, but the differences are insignificant from our perspective. One usually assumes that, to solve a problem of some size n, the number of processors should be bounded by a polynomial in n—a necessary, but hardly sufficient, condition to ensure efficiency.

There are two general strands of PRAM research. The first asks, what problems can be solved in polylog time on a PRAM with a polynomial number of processors? Polylog time serves as a gold-standard for parallel running time, and a polynomial number of processors provides a necessary condition for efficiency. The class $\mathcal{NC}$ is defined as the set of such problems. The second strand of research asks, what algorithms can be efficiently parallelized? That is, for which problems are there parallel algorithms which are much faster than the corresponding sequential ones, yet with processor-time product close to the sequential running time.

While theoretically appealing, the PRAM model suffers from the practical drawback that fully shared-memory machines with large numbers of processors do not exist to date (though they may in the future) and simulations are slow. Building a large computer with a large robust shared memory seems difficult. Moreover, allowing an arbitrary polynomial number of processors allows the creation of theoretically beautiful parallel algorithms which will never be run for any substantial n.

It seems natural to inquire about the relations between $\mathcal{MRC}$, $\mathcal{DMRC}$, and known complexity classes such as $\mathcal{NC}$ and $\mathcal{P}$. Since the comparisons are cleaner in the deterministic case, we focus on $\mathcal{DMRC}$ here, but there are analogous questions for $\mathcal{MRC}$.

Strictly speaking, before comparing $\mathcal{DMRC}$ to $\mathcal{NC}$ and $\mathcal{P}$, one has to convert the binary string input $\langle b_1, b_2, ..., b_n\rangle$ into the MapReduce input format, which we can do by replacing the bit string by the sequence $\langle\langle 1, b_1\rangle, \langle 2, b_2\rangle, ..., \langle n, b_n\rangle\rangle$. In what follows we abuse the terminology and compare these classes directly.

It is easy to see that $\mathcal{DMRC} \subseteq \mathcal{P}$, but is $\mathcal{P} \subseteq \mathcal{DMRC}$? Similarly, what is the relationship between $\mathcal{DMRC}$ and $\mathcal{NC}$?

We partially settle the answer to the latter question in Section 7, showing that a large class of languages $L \in \mathcal{NC}$ are in $\mathcal{DMRC}$ as well. The answer to the converse question—is $\mathcal{DMRC}$ a subset of $\mathcal{NC}$?—is NO, unless $\mathcal{P} = \mathcal{NC}$, but trivially so.

THEOREM 4.1. *If $\mathcal{P} \neq \mathcal{NC}$ then $\mathcal{DMRC} \not\subseteq \mathcal{NC}$.*

Proof. Assume $\mathcal{P} \neq \mathcal{NC}$. Then any $\mathcal{P}$-complete language, such as CIRCUIT VALUE is not in $\mathcal{NC}$. Recall the definition of CIRCUIT VALUE: given a Boolean circuit with one output gate, having AND, OR, and NOT gates, and Boolean values for the inputs, does the circuit evaluate to TRUE?

We now "pad" inputs to CIRCUIT VALUE, getting a new language PADDED CIRCUIT VALUE, which will be in $\mathcal{DMRC} - \mathcal{NC}$. Specifically, define a new language PADDED CIRCUIT VALUE as follows. To generate all strings in PADDED CIRCUIT VALUE, take each string in CIRCUIT VALUE (for which the output evaluates to TRUE) and append $n^2 - n$ zeroes, if the input length was n. Let N denote the size of the padded input, $N = n^2$.

The key-value language associated to PADDED CIRCUIT VALUE is clearly in $\mathcal{DMRC}$, for now one needs only memory roughly $\sqrt{N}$ to solve an instance of PADDED CIRCUIT VALUE of length N on *one* reducer, after stripping out the padding.

However, PADDED CIRCUIT VALUE is $\mathcal{P}$-Complete, as CIRCUIT VALUE can be reduced to PADDED CIRCUIT VALUE in log space, and hence does not lie in $\mathcal{NC}$ (by the assumption that $\mathcal{P} \neq \mathcal{NC}$). □

While we strongly suspect that the answer to the question as to whether $\mathcal{P} \subseteq \mathcal{DMRC}$ is NO, we cannot prove that there is a language in $\mathcal{P}$ whose associated key-value language lies outside $\mathcal{DMRC}$.

Any language, like all of those in $\mathcal{DMRC}$, solvable in polynomial time on a RAM in quadratic space is, by the generic simulation of RAM's by Turing machines, solvable on a Turing machine simultaneously in space $O(n^2 \log n)$ and polynomial time. (We are abusing the terminology a bit here, since, strictly speaking, languages are not in $\mathcal{DMRC}$.) It follows that an obvious candidate for a language in $\mathcal{P} - \mathcal{DMRC}$ would be a language which can be solved by a Turing machine in polynomial time but not simultaneously with $O(n^2 \log n)$ space. However, such languages are not known to exist. Specifically, if L were such a language, then $L \notin \mathcal{LOGSPACE}$, but $L \in \mathcal{P}$. So the

desired L would be in $\mathcal{P} - \mathcal{LOGSPACE}$, yet whether $\mathcal{P} = \mathcal{LOGSPACE}$ is a long standing open question.

4.2 MapReduce: Algorithms and Models MapReduce is very well suited for naive parallelization—for example, counting how many times a word appears in a data set. However, more recently algorithms have emerged for nontrivially parallelizable computations. Kang et al. [9] show how to use MapReduce to compute diameters of massive graphs, taking as an example a webgraph with 1.5 billion nodes and 5.5 billion arcs. Tsourakakis et al. [11] use MapReduce for counting the total number of triangles in a graph. Motivated by personalized news results, Das et al. [3] implement the EM clustering algorithm on MapReduce. Overall, each of these works gives practical MapReduce algorithms, but does not rigorously define the framework under which they should be analyzed.

Previously, Feldman et al. [6] introduced the notion of Massively Unordered Distributed (MUD) algorithms, a model based on the MapReduce framework. While modeling the same underlying system, their approach has two crucial differences from ours. First, in the MUD framework each reducer operates on a stream of data, whereas, in our model, each reducer has random access to *all* of the values associated with the given key. Second, in MUD, each reducer is restricted to only using polylogarithmic space. These distinctions gives our model more power and play an important role in our algorithms.

5 Finding an MST of a Dense Graph Using MapReduce

Now that we have formally defined the MapReduce model, we proceed to describe an algorithm in $\mathcal{MRC}$ for finding the Minimum Spanning Tree (MST) of a dense graph. As we shall exhibit, this algorithm will take advantage of the interleaving of sequential and parallel computation that MapReduce offers algorithm designers. Thus, given a graph $G = (V, E)$ on $|V| = N$ vertices and $|E| = m \geq N^{1+c}$ edges for some constant $c > 0$ (n still denoting the length of the input, not the number of vertices), our goal is to compute the minimum spanning tree of the graph.

We give a new algorithm for MST and then show how it can be easily parallelized. Fix a number k, and randomly partition the set of vertices into k equally sized subsets, $V = V_1 \cup V_2 \cup \cdots \cup V_k$, with $V_i \cap V_j = \emptyset$ for $i \neq j$ and $|V_i| = N/k$ for all i. For every pair $\{i, j\}$, let $E_{i,j} \subseteq E$ be the set of edges induced by the vertex set $V_i \cup V_j$. That is, $E_{i,j} = \{(u, v) \in E \mid u, v \in V_i \cup V_j\}$. Denote the resulting subgraph by $G_{i,j} = (V_i \cup V_j, E_{i,j})$.

Assume without loss of generality (say, by appending an index to each weight to break ties) that all of the edge weights are unique. Our algorithm proceeds as follows. First, for each of the $\binom{k}{2}$ subgraphs $G_{i,j}$, compute the unique minimum spanning forest $M_{i,j}$. Then let H be the graph consisting of all of the edges present in some $M_{i,j}$: $H = (V, \cup_{i,j} M_{i,j})$. Finally, compute M, the minimum spanning tree of H. The following theorem proves that this algorithm is correct.

THEOREM 5.1. *The tree M computed by the algorithm is the minimum spanning tree of G.*

The algorithm works by sparsifying the input graph and then taking the MST of the resulting subgraph H. We show that no relevant edge was thrown out, that is, the minimum spanning trees of G and H are identical.

Proof. Consider an edge $e = \{u, v\}$ that was discarded, that is, $e \in E(G)$ but $e \notin E(H)$; we show that e is not part of the minimum spanning tree of G. Observe that any edge $e = \{u, v\}$ is present in at least one subgraph $G_{i,j}$. If $e \notin M_{i,j}$ then by the cycle property of minimum spanning trees, there must be some cycle $C \subseteq E_{i,j}$ such that e is the heaviest edge on the cycle. However, since $E_{i,j} \subseteq E$, we have now exhibited a cycle in the original graph G on which e is the heaviest edge. Therefore e cannot be in the MST of G and can safely be discarded. □

The algorithm presented above is far from the optimal sequential algorithm; however, it allows for easy parallelization. Notice that the minimum spanning tree for the individual subgraphs, $G_{i,j}$, can be computed in parallel. Furthermore, by setting the parameter k appropriately, we can reduce the memory used by each MST computation. As we show below, with high probability the memory used is $\tilde{O}(m/k)$ when computing $M_{i,j}$ and $O(Nk)$ when computing the final minimum spanning tree of H. These two facts imply that the algorithm is in $\mathcal{MRC}$.

LEMMA 5.1. *Let $k = N^{c/2}$, then with high probability the size of* every *$E_{i,j}$ is $\tilde{O}(N^{1+c/2})$.*

Proof. We can bound the total number of edges in $E_{i,j}$ by bounding the total degrees of the vertices. $|E_{i,j}| \leq \sum_{v \in V_i} \deg(v) + \sum_{v \in V_j} \deg(v)$. For the purpose of the proof only, partition the vertices into groups by their degree: let W_1 be the set of vertices of degree at most 2, W_2, the set of vertices with degree 3 or 4, and generally $W_i = \{v \in V : 2^{i-1} < \deg(v) \leq 2^i\}$. There are $\log N$ total groups.

Consider the number of vertices from group W_i that are mapped to part V_j. If the group has a small

number of elements, that is, $|W_i| < 2N^{c/2} \log N$, then $\sum_{v \in W_i} \deg(v) \leq 2N^{1+c/2} \log N = \tilde{O}(N^{1+c/2})$. If the group is large, that is, $|W_i| \geq 2N^{c/2} \log N$, a simple application of Chernoff bounds says that the number of elements of W_i mapped into the partition j, $|W_i \cap V_j|$ is $O(\log N)$ with probability at least $1 - \frac{1}{N}$. Therefore with probability at least $1 - \frac{\log N}{N}$:

$$\begin{aligned}\sum_{v \in V_j} \deg(v) &\leq \sum_i \sum_{v \in V_j \cap W_i} \deg(v) \\ &\leq \sum_i 2N^{1+c/2} \log^2 N \\ &\leq \tilde{O}(N^{1+c/2}).\end{aligned}$$

□

Lemma 5.1 tells us that with high probability each part has $\tilde{O}(N^{1+c/2})$ edges. Therefore the total input size to any reducer is $O(n^{1-\epsilon})$.

The algorithm uses the sequential computation available to reducers to compute the minimum spanning tree of the subgraph given to that reducer. There are N^c total parts, each producing a spanning tree with $2N/k - 1 = O(N^{1-c/2})$ edges. Thus the size of H is bounded by $\tilde{O}(N^{1+c/2}) = O(n^{1-\epsilon})$, again being small enough to fit into the memory of a single machine.

6 An Algorithmic Design Technique For $\mathcal{MRC}$

We begin by describing a basic building block of many algorithms in $\mathcal{MRC}$ called "$\mathcal{MRC}$-parallelizable functions." We then show how a family of such functions can used as subroutines of $\mathcal{MRC}$ computations. After that we show how this can be used to compute frequency moments on large inputs, and s-t connectivity on undirected graphs.

DEFINITION 6.1. *Let S be a set. Call a function f on S* $\mathcal{MRC}$-parallelizable *if there are functions g and h so that:*

1. *For any partition $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of S, where $\cup_i T_i = S$ and $T_i \cap T_j = \emptyset$ for $i \neq j$ (of course), f can be expressed as: $f(S) = h(g(T_1), g(T_2), \ldots, g(T_k))$.*
2. *g and h can be expressed in $O(\log n)$ bits.*
3. *g and h can be computed in time polynomial in $|S|$ and every output of g can be expressed in $O(\log n)$ bits.*

Intuitively, this definition says that if one wants to evaluate f on a set S, one could do so by partitioning S arbitrarily, applying g to each part of the partition, and then applying h to the results. Next we show how a family of such functions can be computed under the memory restrictions imposed by $\mathcal{MRC}$.

LEMMA 6.1. *Consider a universe $\mathcal{U}$ of size n and a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subsets of $\mathcal{U}$, where $S_i \subseteq \mathcal{U}$, $\sum_{i=1}^k |S_i| \leq n^{2-2\epsilon}$, and $k \leq n^{2-3\epsilon}$. Let $\mathcal{F} = \{f_1, \ldots, f_k\}$ be a collection of $\mathcal{MRC}$-parallelizable functions. Then the output $f_1(S_1), \ldots, f_k(S_k)$ can be computed using $O(n^{1-\epsilon})$ reducers each with $O(n^{1-\epsilon})$ space.*

This lemma says that a family of $\mathcal{MRC}$-parallelizable functions defined over subsets of the same universe can be computed as a subroutine of a MapReduce computation where the original input size is n. Since $O(n^{2-2\epsilon})$ is be the global amount of memory available to any MapReduce program, the lemma requires that the input has few enough subsets that they fit in memory, and that the sum of the sizes of the subsets also fits into memory. The power of the $\mathcal{MRC}$-parallelizable functions lemma is that it allows an algorithm designer to focus on the structure of the problem and the input; the lemma will handle how to distribute the input across the reducers in such a way as to not overflow the memory of any one reducer.

At a high level, we would like to assign a reducer for each set S_i, map both the elements of S_i and the function f_i itself to the same reducer and compute the output. There is two technical challenge which we need to be wary of. The S_i may be too large to fit on one reducer. In particular if $|S_i| > n^{1-\epsilon}$ then the computation of $f_i(S_i)$ needs to be spread across several reducers. To deal with these issues we use the fact that functions f_i are $\mathcal{MRC}$-parallelizable to our advantage. In the first round we partition the set of reducers into t different blocks. Each set S_i is then partitioned across the reducers in its assigned block, which computes the intermediate values $g_i(S_i)$. This partitioning ensures that the input to any individual reducer is not too large. In the second round, we map all of the intermediate results of block S_i to the same reducer, and compute the final output using the function h_i. The mapping in this step will again ensure that no reducer is inundated with an input that is larger than its memory.

Input The input to the subroutine consists of pairs of the form $\langle i; u \rangle$ indicating that $u \in S_i$, and the individual functions g_i and h_i for all $i \in [k]$.

Initialize Let $M = n^{1-\epsilon}$ denote the number of reducers the subroutine will use. Partition them into blocks of size $B = \Theta(n^{\epsilon})$. Let $t = \lceil M/B \rceil$ be the total number of blocks. Construct universal hash functions, $hash_1$ and $hash_2 : [k] \to [t]$.

Map 1: For each $\langle i; u \rangle$, output $\langle r; (u, i) \rangle$ where r is chosen uniformly at random among the reducers in block $B_{hash_1(i)}$. Map each function g_i and h_i to

$\langle b; (g_i, i)\rangle$ and $\langle b; (h_i, i)\rangle$, for every b in the block $B_{hash_1(i)}$.

Reduce 1: The input to this reducer is of the form $\langle r; ((u_1, i), \ldots, (u_k, i), (g_i, i), (h_i, i))\rangle$, where $\{u_1, u_2, \ldots, u_k\} = T_j \subseteq S_i$ is one of the parts in the partition of S_i induced by Map 1. The reducer computes $g_i(T_j)$ and outputs $\langle r; (g_i(T_j), i, h_i)\rangle$.

Map 2: The input to the mapper is of the form $\langle r; (g_i(T_j), i, h_i)\rangle$. The mapper outputs $\langle hash_2(i); (g_i(T_j), h_i))\rangle$.

Reduce 2: The input to the final reducer is $\langle hash_2(i); ((g_i(T_1), h_i), (g_i(T_2), h_i), \ldots, (g_i(T_B), h_i))\rangle$, where the set $\{T_1, T_2, \ldots, T_B\}$ forms a partition of S_i. The reducer computes h_i and outputs $\langle hash_2(i); h_i(g_i(T_1), g_i(T_2), \ldots, g_i(T_B))\rangle = \langle hash_2(i); f_i(S_i)\rangle$.

The next lemma shows that $hash_1$ prevents any reducer in the Reduce 1 phase from overflowing its memory.

LEMMA 6.2. *Each reducer in step Reduce 1 will have $\tilde{O}(n^{1-\epsilon})$ elements mapped to it with high probability.*

We prove the Lemma by showing that each n^ϵ-sized block of reducers gets $\tilde{O}(n)$ elements mapped to it with high probability. Since the individual reducer for each element is selected uniformly at random from those in a block, an easy application of Chernoff bounds completes the Lemma.

Proof. Partition the sets S_i into groups, such that group $G_j = \{S_i \in \mathcal{S} : 2^{j-1} < |S_i| \leq 2^j\}$. Since $|S_i|$ is bounded by n, there are at most $\log n$ such groups. Define the *volume* of group j as $V_j = |G_j| \cdot 2^j$. Groups having volume less than $O(n \log n)$ could all be mapped to one block without violating the space restrictions of the reducers. We now focus on groups with $V_j > n \log n$. Let G_j be such a group; then G_j contains between $\frac{n \log n}{2^j}$ and $\frac{2n \log n}{2^j}$ elements. Fix a particular block of reducers. Since the size of the block is n^ϵ, there are $n^{1-2\epsilon}$ such blocks. Since $hash_1$ is universal, the probability that any set $S \in G_j$ maps to a particular block is exactly $n^{2\epsilon-1}$. Therefore, in expectation, the number of elements of G_j mapping to this block is $\nu = \frac{2n^{2\epsilon} \log n}{2^j}$. A bad event happens if more than $\delta = n^{1-2\epsilon}$ elements map to this block, as that would result in a total volume of $\Omega(n \log n)$. However Chernoff bounds tell us that the probability of such an event happening is less than $2^{-(1+\delta)\nu} = O(1/n^2)$. Taking a union bound over all $n^{1-\epsilon}$ blocks and $\log n$ groups, we can conclude that the probability of any block, and therefore any reducer, being overloaded is bounded below by $1 - 1/n$. □

LEMMA 6.3. *With high probability, each reducer in step Reduce 2 will have at most $n^{1-\epsilon}$ values of g_i mapped to it.*

Proof. Since $hash_2$ is universal, in expectation the number of sets mapped to a block in Reduce 2 is $\frac{k}{t}$. If $k < t$ then each set can be mapped to its own block. If $k \geq t$ then $\frac{k}{t} \leq \frac{n^{2-3\epsilon}}{n^{1-2\epsilon}} = n^{1-\epsilon}$. Denote by N_b the number of sets mapped to block b. By the Chernoff bound,

$$\Pr\left[N_b > (1 + \log n)\frac{k}{t}\right] < 2^{-(1+\log n)\frac{k}{t}} \leq 1/n.$$

Since there are $t = \Theta(n^{1-2\epsilon})$ blocks, applying the union bound shows that the probability any reducer gets overloaded is $O(\frac{1}{n^{2\epsilon}})$. □

A similar argument to Lemma 6.3 shows that the reducers have enough memory to store the g_i and h_i functions. This combined with Lemmas 6.2 and 6.3 and the fact that g_i and h_i are polynomial-time computable prove Lemma 6.1.

6.1 Applications of the Functions Lemma As mentioned above the power of the functions lemma is that it allows the algorithm designer to think of parallel algorithms without the worry of overloading a particular reducer. The memory restrictions of Lemma 6.1 allow it to be used as a subroutine when the size of input of the calling $\mathcal{MRC}$ algorithm is n. Because the subroutine uses $O(n^{1-\epsilon})$ reducers each with $O(n^{1-\epsilon})$ memory, it does not violate any constraints specified by the MRC class when original input size is n. Next we show two explicit examples of the use of this subroutine. The first uses Lemma 6.1 twice to compute frequency moments where the f_i are identical. The second uses Lemma 6.1 as a subroutine where the f_i are different for each i.

6.1.1 Frequency Moments Suppose we would like to compute the k^{th} frequency moment of a string. Let $\mathcal{L}$ be the string alphabet, and represent a length-n string as a set of pairs $\langle i, \ell_i\rangle$ where $i \in [N]$ represents the position and $\ell_i \in \mathcal{L}$ is the symbol at position i. This set of pairs is also the universe U. For every element $\ell \in \mathcal{L}$, denote by $S_\ell \subseteq U$ the set of pairs in $\mathcal{U}$ containing letter ℓ. To compute frequency moments we need to compute $f_\ell = |S_\ell|^k$. Summing the values of the f_ℓ returns the frequency moment. It is easy to see that f_ℓ is an $\mathcal{MRC}$-parallelizable function. Define g as the size function, $g(\{t_1, t_2, \ldots, t_k\}) = k$, and h as $h(i_1, i_2, \ldots, i_m) = (i_1 + i_2 + \cdots + i_m)^k$. For any partition $\mathcal{T} = (T_1, \ldots, T_m)$ of S_ℓ, $h(g(T_1), g(T_2), \ldots, g(T_m)) = |S_\ell|^k$. Thus, one application of the functions lemma yields the values of

the $f_\ell(S_\ell)$. We can then use another simple application of the functions lemma to compute the overall frequency moment: $\sum_{\ell \in \mathcal{L}} f_\ell(S_\ell)$.

6.1.2 Undirected s-t connectivity Suppose we are given an N-node graph $G = (V, E)$ and two nodes $s, t \in V$ and we are asked whether there exists a path from s to t. Note that this problem can be efficiently computed by PRAMs, thus we can use the Simulation Theorem (Theorem 7.1) to achieve such an algorithm. In this section, however, we give a more direct approach.

In the case that the graph is relatively dense, with $|E| = N^{1+\Omega(1)}$, we can use matrix multiplication to compute the N^{th} power of the adjacency matrix in $O(\log N)$ rounds[1]. If, however, the graph is sparse, the full adjacency matrix will not fit into memory across all of the machines (recall that the total memory available is $N^{2-2\epsilon}$, whereas the full matrix will be of size N^2) and we need to resort to other methods.

In what follows we give a simple labeling algorithm that computes s-t connectivity on sparse graphs in $O(\log N)$ rounds[2]. We first give the high level details and then describe how to implement it in MapReduce.

Throughout the algorithm, each node $v \in V$ maintains a label $\ell(v)$, describing the connected component it is in. Denote by $L_v \subseteq V$ as the set of vertices with label v. L_v represents the connected component containing v. Following standard notation, we define $\Gamma(v)$ to be the set of neighbors of v. For a set S, denote by $\Gamma(S)$ the set of neighbors of all nodes in S themselves not in S. Finally, denote by $\Gamma'(v) = \Gamma(L_v)$. Let π denote an arbitrary total order on the vertices.

1. Begin with every node $v \in V$ being active with label $\ell(v) = v$.
2. For $i = 1, 2, 3, ..., O(\log N)$ do:
   (a) Call each active node a leader with probability $1/2$.
   (b) For every active non-leader node w, find the smallest (according to π) node $w^* \in \Gamma'(w)$.
   (c) If w^* is not empty, mark w passive and relabel each node with label w by w^*.
3. Output *true* if s and t have the same labels, false otherwise.

LEMMA 6.4. *At any point of the algorithm, if any two nodes s and t have the same label, then there is a path from s to t in G.*

Proof. The proof proceeds by induction. At the beginning of the algorithm every node has its own label and the statement is vacuously true.

Suppose the statement is true at the beginning of round i. The only interesting case is when $\ell(s) \neq \ell(t)$ before the iteration and $\ell(s) = \ell(t)$ after the iteration. Consider a non-leader node w, and a node w^* as described in the algorithm. Assume without loss of generality that $s \in L_w$ and $t \in L_{w^*}$. By induction, there exist paths from s to w and from w^* to t. The definition of $\Gamma'(v)$ ensures that there exists a node u, with $\ell(u) = \ell(w)$, and the edge $(u, w^*) \in E$. Thus the path $s \to w \to u \to w^* \to t$ is in G (the $w \to u$ path existing because $l(u) = l(w)$). □

LEMMA 6.5. *Every connected component of G has a unique label after $O(\log N)$ rounds with high probability.*

Proof. To prove the running time we show that the number of labels in any connected component decreases by a constant factor (in expectation) in every round, until, of course, every vertex in the connected component has the same label. Fix an active node u (note that the total number of distinct labels is equal to the number of active nodes.). If the component containing u has more than one label, then there must exist a node $v' \in \Gamma'(u)$ with a different label from u. Let $\ell(v') = v$. With probability $1/4$ the active node v is selected as a leader and u is a non-leader. Then $v' \in \Gamma'(u)$, and u will be relabeled as v and marked passive. Therefore, the probability of any node's surviving a round while there is more than one label in a connected component is at most $3/4$. An application of Chernoff bounds concludes the proof. □

So far we have proven that the above algorithm is correct; we now show how to implement it in MapReduce. The key to the parallelization is that leader selection, follower selection and the relabeling can all be done in parallel. To make this more precise we turn again to the Functions Lemma.

Selecting the set of leaders in parallel is trivial. To select the followers, let $hash_1 : V \to \{0, 1\}$ be a universal hash function; the set of leaders is precisely those active $v \in V$ with $hash_1(v) = 1$. The next task is for every non-leader node w to compute the node w^* that it will be following. Observe that w^* depends on $\Gamma'(w) \subseteq V$, in fact the algorithm requires the minimum label from nodes in $\Gamma'(w)$. Since min is an $\mathcal{MRC}$-parallelizable function, it fits the conditions of the Theorem. The only thing that remains is computing the individual sets $\Gamma'(w)$. We achieve this by scanning through all of the edges. For an edge $\{u, v\}$ we can check if the labels of the endpoints agree. If not, then $\ell(v) \in \Gamma(\ell(u))$ and $\ell(u) \in \Gamma(\ell(v))$, where abusing

[1] Dense matrix multiplication is trivial in $\mathcal{MRC}$—partition each matrix into blocks and multiply the blocks before aggregating the results.

[2] We suspect that this is a standard connectivity algorithm in the PRAM literature.

notation we use $\ell(v)$ to refer to the node that v is labeled with.

Finally, we describe the relabeling step. Let w and w^* be as in the description of the algorithm. We need to relabel all of the nodes with the label ℓ_w to have the label ℓ_{w^*}. For the subset L_w, let f_w be such a relabel function. It is easy to check that the family of sets $\{L_w\}$ and the family of functions $\{f_w\}$ satisfies the conditions of the Lemma 6.1.

7 Simulating PRAMs via MapReduce

THEOREM 7.1. *Any CREW PRAM algorithm using $O(n^{2-2\epsilon})$ total memory, $O(n^{2-2\epsilon})$ processors and $t = t(n)$ time can be run in $O(t)$ rounds in $\mathcal{DMRC}$.*

In this proof we will show that such a PRAM algorithm can be simulated by an algorithm in $\mathcal{DMRC}$. At a high level we will use $O(n^{2-2\epsilon})$ reducers where one reducer simulates each processor used in the PRAM algorithm and another reducer simulates each memory location used by the PRAM algorithm. Conceptually, we will use the mappers to route memory requests and ship the relevant memory bits to the reducer responsible for the particular processor. Each reducer will then perform one step of computation for each of the PRAM processors assigned to it, write out memory updates, and request new memory positions. The process then repeats. The authors of [6] give a similar simulation algorithm in their work.

Proof. We reduce the simulation problem to only keeping track of updated memory locations. Therefore we ensure that every memory location is updated every round by modifying the PRAM algorithm to have an extra $O(n^{2-2\epsilon})$ processors (one for each location in memory). At every time step each of these "dummy" processors requests a unique memory address and attempts to write the same value back to it. If at any point in time there are two writes to the same memory location, the dummy value gets overwritten.

We now describe the simulation. At time t of the PRAM algorithm let b_i^t denote the $\langle$address, value$\rangle$ pairs that processor i reads from. Let $b_i^t = \emptyset$ if processor i does not read from a memory location at time t. Let w_i^t be the $\langle$address, value$\rangle$ pair that processor i writes to at time t. Let $w_i^t = \emptyset$ if processor i does not write to a memory location at time t.

We will show how the computation at time t is executed by a constant number of MapReduce steps. Assume inductively that reducer ρ_1^t has as input $\langle i; b_i^t\rangle$. Then ρ_1^t will simulate one step of the computation for the processor and output $\langle i; r_i^{t+1}, w_i^t\rangle$, where r_i^{t+1} is the memory address that processor i will need during the next time step; and w_i^t is the $\langle$address; value$\rangle$ pairs that were written to during time t.

The next mapper μ_1^t will take as input $\langle i; r^{t+1}, w_i^t\rangle$. The mapper outputs $\langle r^{t+1}; i\rangle$ signifying the memory location requested by processor i. Moreover, let $w_i^t = (a, v)$ where a is an address and v the value written to it, then the mapper also outputs $\langle a; w_i^t, i\rangle$ signifying the update to the state of the memory.

The next reducer ρ_2^t takes as input tuples of two types. The first type has form $\langle a_j; (a_j, v_j), i\rangle$ which represents that the new value for address a_j is v_j. It will get such values for all writes that were done to address a_j. Since the PRAM algorithm is CREW, this tuple will only occur once per memory address a_j. The second type of input it will take has form $\langle a_j; i\rangle$. This represents that processor i would like the value in address a_j. Thus, ρ_2^t fulfills this request by outputting $\langle a_j; (a_j, v_j), i\rangle$. Finally map μ_2^t makes sure that the processor i gets the new value for a_j by taking as input $\langle a_j; (a_j, v_j), i)\rangle$ and outputting $\langle i; a_j, v_j\rangle$. □

8 Conclusion

We have presented a rigorous computational model for the MapReduce paradigm. By restricting both the total memory per machine and the total number of machines to $O(n^{1-\epsilon})$ we ensure that the programmer must parallelize the computation and that the number of machines used must remain relatively small. The combination of these two characteristics were not previously captured in the PRAM model. We strived to be parsimonious in our definitions, and therefore specifically did not restrict the time available for a reducer to be, for example, linear. Rather we simply require that mappers, as well as reduces run in polynomial time.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments.

References

[1] D. K. G. Campbell. A survey of models of parallel computation. Technical report, University of York, March 1997.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 4:1–12, May 1993.

[3] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of WWW*, pages 271–280, 2007.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.
[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
[6] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributing symmetric streaming computations. In S.-H. Teng, editor, *SODA*, pages 710–719. SIAM, 2008.
[7] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference*, pages 205–217, New York, NY, USA, 1971. ACM.
[8] Hadoop wiki - powered by. `http://wiki.apache.org/hadoop/PoweredBy`.
[9] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with hadoop. Technical Report CMU-ML-08-117, CMU, December 2008.
[10] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
[11] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Knowledge Discovery and Data Mining (KDD)*, 2009.
[12] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, August 1990.
[13] V. V. Vazirani. *Approximation Algorithms*. Springer, March 2004.
[14] Yahoo! partners with four top universities to advance cloud computing systems and applications research. Yahoo! Press Release, 2009. `http://research.yahoo.com/news/2743`.