

# A New Computation Model for Rack-Based Computing

Foto N. Afrati  
National Technical University of Athens  
afirati@softlab.ece.ntua.gr

Jeffrey D. Ullman  
Stanford University  
ullman@infolab.stanford.edu

## ABSTRACT

Implementations of map-reduce are being used to perform many operations on very large data. We explore alternative ways that a system could use the environment and capabilities of map-reduce implementations such as Hadoop, yet perform operations that are not identical to map-reduce. In particular, we look at strategies for taking the join of several relations and sorting large sets. The centerpiece of this exploration is a computational model that captures the essentials of the environment in which systems like Hadoop operate. Files are unordered sets of tuples that can be read and/or written in parallel; processes are limited in the amount of input/output they can perform, and processors are available in essentially unlimited supply. In our study, we focus on communication among processes and processing time costs, both total and elapsed. We show tradeoffs among them depending on the computational limits we invoke on the processes.

## 1. Introduction

Search engines and other data-intensive applications mainly process large amounts of data that need special-purpose computations. The canonical problem today is the sparse-matrix-vector calculation involved with PageRank [5], where the dimension of the matrix and vector can be in the 10's of billions. Most of these computations are conceptually simple but their size has led implementors to distribute them across hundreds or thousands of low-end machines. This problem, and others like it, led to a new software stack to take the place of file systems, operating systems, and database-management systems.

### 1.1 The New Software Stack and our Contribution

Central to this stack is a file system such as the Google File System (GFS) [13] or Hadoop File System (HFS) [2]. Such file systems are characterized by:

- Block sizes that are perhaps 1000 times larger than those in conventional file systems — multimegabyte instead of multi-kilobyte.
- Replication of blocks in relatively independent locations (e.g., on different racks) to increase availability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODS'09, Providence, RI, USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

A powerful tool for building applications on such a file system is map-reduce [10] or its open-source equivalent Hadoop [2]. Briefly, map-reduce allows a Map function to be applied to data stored in one or more files, resulting in key-value pairs. Many instantiations of the Map function can operate at once, and all their produced pairs are routed by a *master controller* to one or more Reduce processes, so that all pairs with the same key wind up at the same Reduce process. The Reduce processes apply another function to combine the values associated with one key to produce a single result for that key.

Map-reduce, inspired from functional programming, is a natural way to implement sparse-matrix-vector multiplication in parallel, and we shall soon see an example of how it can be used to compute parallel joins. Map-reduce offers resilience to hardware failures, which can be expected to occur during a massive calculation. The master controller manages Map and Reduce processes and is able to redo them if a process fails.

The new software stack includes higher-level, more database-like facilities, as well. Examples are Google's BigTable [7], or Yahoo!'s PNUTS [9]. At a still higher level, Yahoo!'s PIG/PigLatin [17] translates relational operations such as joins into map-reduce computations.

However, there are concerns that as effective the map-reduce framework might be with certain tasks, there are issues that are not effectively addressed by this framework. For example, [8] talks about adding to map-reduce a "merge" phase and demonstrates how this can express relational algebra operators. In [11] a discussion is held that the efficiency of a DBMS, as embodied in tools such as indexes, are missing from the map-reduce framework.

- This paper presents a model in which we can improve the efficiency of certain computations beyond what the map-reduce framework allows, and yet exploit the same assumptions and computing environment in which map-reduce has been effective.
- For example, as we shall see in Section 3.2, there are opportunities for efficiency improvements for joins with even small deviations from the standard map-reduce framework.

The sorting task has often been used for testing computational environments about data management applications. For example, the goal in [3, 6, 20] is to explore the viability of commercial technologies for utilizing cluster resources, racks of computers and disks; in these works, algorithms for external sorting are implemented with the focus on I/O efficiency. These algorithms are tested against well known benchmarks [16, 20]. The map-reduce framework does not lend well to the sorting task due to its high degree of sequentiality.

- We develop the first sorting algorithm for computer environments with the same assumptions as map-reduce, i.e., on our proposed computation model.

- In Section 5.1, we find a constant depth algorithm. E.g., this algorithm can be used for sorting  $10^{15}$  elements with today's technology at an elapsed processing time four times the processing time for sorting  $10^{10}$  elements which we assume can be done on one machine.
- Based on similar ideas, we develop a merging algorithm (Section 5.7) and a sorting algorithm for any number  $n$  of elements in Sections 5.6 through 5.9. The algorithm has been designed to minimize the communication cost. We show that the communication cost of the algorithm is  $O(n \log^{1.7} n)$  while we show the lower bound on the communication for sorting in our model to be  $O(n \log n)$  (in Section 5.5). Finally in Section 5.8 we give an asymptotically better algorithm (by a factor of  $\log \log n$  away from the lower bound) which, as we argue, may be less useful in practice, due to its small base of the logarithm.

## 2. The New Large-Scale-Data Model

Here, we introduce the data elements of the model and then discuss the different cost measures by which we evaluate algorithms.

### 2.1 Elements of the Model

Algorithms in the model we propose are expressed in terms of:

1. *Files*: A file is a set of tuples. It is stored in a file system such as GFS, that is, replicated and with a very large block size  $b$ . Unusual assumptions about files are:
  - (a) We assume the order of tuples in a file cannot be predicted. Thus, these files are really relations as in a relational DBMS.
  - (b) Many processes can read a file in parallel. That assumption is justified by the fact that all blocks are replicated and so several copies can be read at once.
  - (c) Many processes can write pieces of a file at the same time. The justification is that tuples of the file can appear in any order, so several processes can write into the same buffer, or into several buffers, and thence into the file.
2. *Processes*: A process is the conventional unit of computation. It may obtain input from one or more files and write output to one or more files. The unusual aspect of processes in our model is that we assume there are upper and lower limits on how much input and output a process may have. The lower limit is the block size  $b$ , since if it gets any input at all from a file, it will get at least one block of data (which could be mostly empty, but in our cost model, to be discussed, the process must “pay” for a full block). The upper limit, which we denote by  $s$ , can represent one of several things, such as:
  - (a) The amount of time we are willing to spend moving data from the file system to or from the local storage of the processor that executes the process. The typical computing environment in which our model makes sense is a rack or racks of processors, connected by fairly low-speed interconnect, e.g., gigabit Ethernet. In that environment, even loading main memory can take a minute.
  - (b) The size of main memory at a processor, if we want to execute the process there to avoid the cost of moving data between main memory and local disk.

We shall leave  $s$  as a parameter without choosing one specific interpretation. We could alternatively allow for  $s$  not to reflect a physical limit on processors but rather to force processes to be

of limited scope and thereby to constrain algorithms to obtain lots of parallelism. It is interesting to note that  $s$  and  $b$  may not differ by too much. However, as we shall see, it is the parameter  $s$ , limiting input/output size for a single process that has the most influence on the design of algorithms. In order to simplify arguments in what follows, we shall often treat  $s$  not as an absolute limit, but as an order-of-magnitude limit. That is, we shall allow processes that have input and/or output size  $O(s)$ , where some small constant is hidden inside the big-oh.

3. *Processors*: These are conventional nodes with a CPU, main memory, and secondary storage. We do not assume that the processors hold particular files or components of files. There is an essentially infinite supply of processors. Any process can be assigned to any but only one processor. We do not assume that the application can control which processor gets which process; thus, it is not possible for one process to pass data to another by leaving it at a processor<sup>1</sup>.

### 2.2 Cost Measures for Algorithms

An *algorithm* in our model is an acyclic graph of processes with an arc from  $P_1$  to  $P_2$  if  $P_1$  generates output that is (part of) the input to  $P_2$ . It is subject to the constraint that a process cannot begin until all of its input has been created. Note that we assume an infinite supply of processors, so any process can begin as soon as its input is ready.

Our goal is to study efficient computation, so we need to measure the *processing time* of algorithms. Since an algorithm consists of many processes, perhaps communicating by passing files among each other, we also need to consider *communication costs*. Thus, we define:

- The *total communication cost* (*total processing cost*, respectively) is the sum of the communication (processing, respectively) costs of all processes that constitute an algorithm.
- The *elapsed communication cost* (*elapsed processing cost*, respectively) is defined on the acyclic graph of processes. Consider a path through this graph, and sum the communication (processing, respectively) costs of the processes along that path. The maximum sum, over all paths, is the elapsed communication (processing, respectively) cost.

### 2.3 Comparison With Other Models

Models in which processors have limited resources and/or limited ability to communicate with other processors have been studied for decades. However, the constraints inherent in the new massive-data architectures are somewhat different from what has been looked at previously, and these differences naturally change what the best algorithms are for many problems.

#### The Kung-Hong Model.

A generation ago, the Kung-Hong model [14] examined the amount of I/O (transfer between main and secondary memory) that was needed on a processor that had a limited amount of main memory. They gave a lower bound for matrix-multiplication in this model. The same model was used to explore transitive closure algorithms ([1], [23]) later. One important difference between the Kung-Hong model and the model we present here is that we place a limit on communication, not local memory.

<sup>1</sup>However, in current map-reduce implementations, data is passed between Map and Reduce processes not through the file store, but through the local disks of processors executing the processes. This choice does not have a major effect on the time; it only affects the likelihood of a data failure.

EXAMPLE 2.1. *As a simple example of how this change affects algorithms, consider the simple problem of summing a very large file of integers. In the Kung-Hong model, it is permitted to stream the entire file into one process, and use main memory to hold the sum. As long as the sum itself is not so large that it cannot fit in main memory, there is no limit on how much data can be read by one process.*

*In our model, one process could only read a limited amount  $s$  of the file. To sum a file with more than  $s$  integers, we would have to use a tree of processes, and the elapsed communication would be greater than the length of the file by a logarithmic factor. On the other hand, because we permit parallel execution of processes, the elapsed time would be much less under our model than under Kung-Hong.* □

### The Valiant Model.

In 1990, Valiant [24] introduced a bridging model between software and hardware having in mind such applications as those where communication was enabled by packet switching networks or optical crossbars, although the model goes arguably beyond that. One of the concerns was to compare with sequential or PRAM algorithms and show competitiveness for several problems including sorting. In [12], a probabilistic algorithm for sorting is developed for this model. The goal in this algorithm is to optimize the competitive ratio of the total number of operations and the ratio between communication time and border parallel computation time (i.e., computation time when the competitive ratio is equal to one). These works differ from ours in the assumptions about the systems that are incorporated to the model and the measures by which algorithms are evaluated.

### Communication Complexity.

There have been several interesting models that address communication among processes. [18] is a central work in this area, although the first studies were based on VLSI complexity, e.g. [22] — the development of lower bounds on chip speed and area for chips that solve common problems such as sorting. Our model is quite different from VLSI models, since we place no constraint on where processes are located, and we do not assume that physical location affects computation or communication speed (although strictly speaking, the placement of processes on the same or different racks might have an effect on performance).

The more general communication-complexity models also differ from ours in assuming that the data is distributed among the processors and not shared in ways that are permitted by our model. In particular, our ability to share data through files changes the constraints radically, compared with models such as [18].

## 3. Algorithms for Multiway Join

The map-reduce framework is quite effective for many problems. In this section we first review how one uses map-reduce to compute the join  $R(A, B) \bowtie S(B, C)$ . Under our model, map-reduce uses as little communication and time as is possible. We then explore the three-way join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$  and compare using a cascade of two map-reduce operations — one for each join — with algorithms that join all three relations at once. We shall see that, depending on the size of intermediate joins, one or the other method is more efficient. This is the first evidence that our model extends map-reduce by providing for more efficient algorithms.

### 3.1 The Two-Way Join and Map-Reduce

Suppose that relations  $R$  and  $S$  are each stored in a file and have  $n_R$  and  $n_S$  tuples, respectively. Let their join have  $n_{RS}$  tuples. To join these relations, we must associate each tuple from either relation with the key that is the value of its  $B$ -component. A collection of Map processes will turn each tuple  $(a, b)$  from  $R$  into a key-value

pair with key  $b$  and value  $(a, R)$ . Note that we include the relation with the value, so we can, in the Reduce phase, match only tuples from  $R$  with tuples from  $S$ , and not a pair of tuples from  $R$  or a pair of tuples from  $S$ .

Similarly, we use a collection of Map processes to turn each tuple  $(b, c)$  from  $S$  into a key-value pair with key  $b$  and value  $(c, S)$ . If  $s$  is the limit on the communication for a single process, then we need  $O(n_R/s)$  Map processes for  $R$  and  $O(n_S/s)$  processes for  $S$ . However, their aggregate communication is  $O(n_R + n_S)$ , which is the minimum we need just to access the files of  $R$  and  $S$ .

The role of the Reduce processes is to combine tuples from  $R$  and  $S$  that have a common  $B$ -value. Typically, we shall need many Reduce processes, so we need to arrange that all tuples with a fixed  $B$ -value are sent to the same Reduce process. Suppose we need  $m$  Reduce processes. Then choose a hash function  $h$  that maps  $B$ -values into  $m$  buckets. The output of any Map process with key  $b$  is sent to a file that corresponds to the value  $h(b)$ . If we create one Reduce process for each of the  $m$  hash values, then these together can find all the joining pairs of tuples, one from  $R$  and one from  $S$ , and write them to one output file. Note that since files are sets, rather than lists, of elements, the Reduce processes are not constrained to execute in any order, and can operate in parallel.

### 3.2 Implementation Under Hadoop

If the above algorithm is implemented in Hadoop, then the partitioning of keys according to the hash function  $h$  can be done behind the scenes. That is, you tell Hadoop the value of  $m$  you desire, and it will create  $m$  Reduce processes and partition the keys among them using a hash function. Further, it passes the key-value pairs to a Reduce process with the keys in sorted order. Thus, it is possible to implement Reduce to take advantage of the fact that all tuples from  $R$  and  $S$  with a fixed value of  $B$  will appear consecutively on the input.

That feature is both good and bad. It allows a simpler implementation of Reduce, but the time spent by Hadoop in sorting the input to a Reduce process may be more than the time spent setting up the main-memory data structures described in Section 3.1. However, to within a constant factor, it does not matter whether the  $m$ -way partitioning of keys is done explicitly or implicitly.

Ragho Murthy has pointed out another option that takes us outside the strict map-reduce formulation, by in effect performing a semijoin first. We could Map only the  $R$  tuples, sending tuple  $(a, b)$  of  $R$  to the key-value pair  $(b, (a, R))$ , as before. Next, another set of Map processes takes the set of keys — i.e., the values of  $R.B$  — and maps only those tuples  $(b, c)$  from  $S$  such that  $b$  is a key from  $R$ , to the key-value pair  $(b, (c, S))$ . Other tuples of  $S$  are not mapped. Only then does a collection of Reduce processes join the tuples from  $R$  and  $S$  having a common  $B$ -value. In a sense, this network of processes is of the form map-map-reduce, rather than just map-reduce.

### 3.3 Communication Cost of the Two-Way Join

If we pick  $m \geq (n_R + n_S)/s$ , then each Reduce process has at most  $s$  input. We must also pick  $m \geq n_{RS}$ , so that each process has at most  $s$  output. The total communication of the Reduce processes is thus  $O(n_R + n_S + n_{RS})$ . That expression is at least as large as the communication for the Map processes, so it also serves as the total big-oh communication cost of the algorithm. Note that this expression is the minimum possible for any algorithm that reads its input and writes its output.

Technically, we need to worry about the effect of the block size  $b$ . However, as long as  $b < s$ , and  $n_R$  and  $n_S$  are at least  $b$ , then we can pick a small enough  $m$  such that each Reduce process gets at least one block of data, and we can ignore the effect of  $b$ .

The elapsed communication depends on the number of Map and Reduce processes. However, note that each path through the process

graph consists of one Map and one Reduce. We can, in principle, use as many processes as we like, up to the limit that each process must receive at least one block of input. Thus the elapsed communication cost can always be a constant in our model due to the large parallelization of the problem.

We may also compute the total and elapsed processing costs. A Map process surely takes time proportional to its input size. A Reduce process can be designed to implement a hash join, for example, and thus to run in time proportional to its input size plus its output size. We conclude that in this case, communication and processing times are, after an appropriate scaling factor, the same.

### 3.4 Three-Way Join

Now, let us consider the join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ . Assume  $R$ ,  $S$ , and  $T$  have sizes  $n_R$ ,  $n_S$ , and  $n_T$ , respectively. Intermediate relations  $R(A, B) \bowtie S(B, C)$  and  $S(B, C) \bowtie T(C, D)$  have sizes  $n_{RS}$  and  $n_{ST}$  respectively, and

$$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$$

has size  $n_{RST}$ . Without loss of generality, assume  $n_{RS} \leq n_{ST}$ .

If we compute the three-way join by a cascade of two two-way joins, and we implement each join by the map-reduce method of Section 3.1, then the communication cost is

$$O(n_R + n_S + n_T + n_{RS} + n_{RST})$$

The reason is that, since  $n_{RS} \leq n_{ST}$ , it is more efficient to join  $R \bowtie S$  first, and then to join that result with  $T$ . The following theorem summarizes the discussion about computing a three-way join as a cascade of two two-way joins based on the analysis of Section 3.3 about the algorithm for a two-way join presented in Section 3.1.

**THEOREM 3.1.** *Using the algorithm in Section 3.1 for computing a two-way join, we can compute the join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$  as a cascade of two two-way joins with communication cost  $O(n_R + n_S + n_T + n_{RS} + n_{RST})$ .*

An alternative algorithm involves joining all three relations at once. Our strategy is to send tuples of  $R$  and  $T$  to many different processes, although each tuple of  $S$  is sent to only one process. The duplication of data increases the communication cost above the theoretical minimum, but in compensation, we do not have to communicate intermediate results. As we shall see, the THREEWAYJOIN algorithm below incurs less communication cost when  $n_{RS}$  and  $n_{ST}$  are both large.

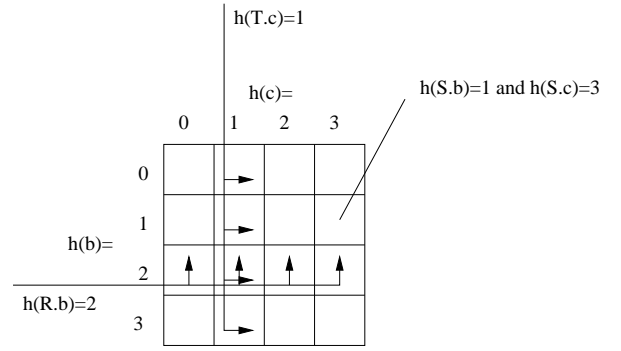
We assign the tuples of all three relations to  $m^2$  join processes, based on hashing values of both  $B$  and  $C$ . This assignment is done by another set of distribution processes (not further discussed), similar to the Map processes of Section 3.1.

Let  $h$  be a hash function with range  $0, 1, \dots, m-1$ , and denote joining-process numbers by pairs  $(i, j)$ , where integers  $i$  and  $j$  are each between 0 and  $m-1$ . Each tuple  $S(b, c)$  is sent to the process numbered  $(h(b), h(c))$ . Each tuple  $R(a, b)$  is sent to all the processes numbered  $(h(b), x)$ , for any  $x$ . Each tuple  $T(c, d)$  is sent to all join processes numbered  $(y, h(c))$  for any  $y$ . Thus, each process  $(i, j)$  gets  $1/m^2$ th of  $S$ , and  $1/m$ th of  $R$  and  $T$ . An example, with  $m = 4$ , is shown in Fig. 1.

The input size for a join process is therefore  $(n_R + n_T)/m + n_S/m^2$ . The output size for a process is  $n_{RST}/m^2$ . The limitation on  $m$  due to input size is  $(n_R + n_S)/m + n_S/m^2 \leq s$ , and the limitation due to output size is  $n_{RST}/m^2 \leq s$ . On the assumption that  $m$  is fairly large, and therefore  $m^2$  is much bigger than  $m$ , the limitation on input size tells us that, approximately,

$$m \geq (n_R + n_T)/s$$

The limitation on output size tells us  $m \geq \sqrt{n_{RST}/s}$ .



**Figure 1: Distributing tuples of  $R$ ,  $S$ , and  $T$  among  $m^2$  processes**

### 3.5 Communication Cost of the THREEWAYJOIN

There are  $m^2$  join processes, and each has an input-plus-output size of  $(n_R + n_T)/m + n_S/m^2 + n_{RST}/m^2$ . The total communication cost of the join processes is thus  $O(m(n_R + n_T) + n_S + n_{RST})$ . This cost surely exceeds the  $O(m(n_R + n_T) + n_S)$  communication of the distribution processes that send the needed data to the join processes, so we can neglect the cost of the distribution processes. The following theorem summarizes the results about the alternative algorithm for computing a three-way join.

**THEOREM 3.2.** *The algorithm THREEWAYJOIN computes the join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$  with communication cost*

$$O(m(n_R + n_T) + n_S + n_{RST})$$

If we compare the formula of Theorem 3.2 with the formula for the cost of the two two-way joins in Theorem 3.1, which is

$$O(n_R + n_S + n_T + n_{RS} + n_{RST})$$

we see that the THREEWAYJOIN could only involve significantly less communication if  $(m-1)(n_R + n_T) < n_{RS}$ . Since  $m \geq (n_R + n_T)/s$ , we approximately require that  $(n_R + n_T)^2/s < n_{RS}$ . Such a situation is possible when the intermediate join size  $n_{RS}$  is large (and therefore so is  $n_{ST}$ , which we assume is larger). We also require that the final result size,  $n_{RST}$  be small, since the square root of that quantity is another lower bound on  $m$ , which we have ignored.

**EXAMPLE 3.3.** *Suppose  $n_R = n_S = n_T = n$ ,  $n_{RS} = n_{ST} = n^2$ , and  $n_{RST}$  is much less than  $n^2$ . Then the communication cost of the two 2-way joins is  $O(n^2)$  while the communication of the THREEWAYJOIN is  $O(mn)$ . Taking  $m$  equal to its least possible value,  $2n/s$ , the 3-way join has communication cost  $O(n^2/s)$ .  $\square$*

### 3.6 Processing Cost of the THREEWAYJOIN

The preferred way to execute a join process is first to build indexes on the tuples from  $R$  and the tuples from  $T$ , e.g., hash tables. The time to create these indexes is proportional to the number of tuples from the two relations, which is  $O(n_R + n_T)/m$ . Then, we use these indexes to look up matching tuples for each of the  $n_S/m^2$  tuples from  $S$ , taking time  $O((n_S + n_{RST})/m^2)$  time to assemble the output. Multiply the processing time of a single process by  $m^2$  to get the processing cost of the 3-way join:  $O(m(n_R + n_T) + n_S + n_{RST})$ . The following theorem summarizes.

**THEOREM 3.4.** *The algorithm THREEWAYJOIN computes the join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$  with processing cost*

$$O(m(n_R + n_T) + n_S + n_{RST})$$

As with communication cost, the expression in Theorem 3.4 can be larger or smaller than the expression for two 2-way joins; it is smaller when the intermediate joins are relatively large.

**EXAMPLE 3.5.** *For the same relations as in Example 3.3, the processing cost of the 3-way join is  $O(mn)$ , assuming  $n_{RST}$  is small. Thus, for both the 3-way join and the cascade of 2-way joins, processing cost is proportional to communication cost.*  $\square$

We can also study the elapsed communication cost and elapsed processing cost of the THREEWAYJOIN. The network of processes for the 3-way join is simply a set of distribution processes feeding a set of join processes. Thus, the elapsed cost of each type is the sum of the corresponding costs for one process of each type. For the cascade of 2-way joins, the network has paths with two Map and two Reduce processes. Thus, the THREEWAYJOIN may offer a small (factor of 2) advantage in elapsed costs, even in situations where the total costs are similar.

### 3.7 3-Way-Join Implementation Under Hadoop

In Section 3.2 we pointed out that Hadoop can perform the partition of keys into any number of Reduce processes automatically. If keys are the pairs of hash values used in Section 3.4, then we can implement the algorithm of that section under Hadoop, as a single map-reduce operation.

However, we cannot use pairs of  $B$ -values and  $C$ -values as keys, even though we were able to use unhashed  $B$ -values for the 2-way join, as discussed in Section 3.2. The reason is that the Map processes would have to map each tuple  $(a, b)$  of  $R$  to a set of key-value pairs with keys of the form  $(b, c)$  for all possible  $C$ -values  $c$ . However, no Map process can be sure it has seen all possible  $C$ -values. An analogous problem occurs for tuples of  $T$ . While we could use a preliminary map-reduce operation to accumulate all possible values of  $B$  and all possible values of  $C$ , so doing results in a cascade of two map-reduce operations, just as for the cascade of two 2-way joins.

## 4. Sorting by Standard Methods

Now, we shall take up the familiar problem of sorting. As we shall see, there is a tradeoff between the elapsed time/communication costs and the total costs.

To begin, since our model does not support sorted lists, one might ask if it is even possible to sort. A reasonable substitute for ordering elements is assigning the proper ordinal to every element. That is, for our purposes a sorting algorithm takes a file of  $n$  pairs  $(x_0, 0), (x_1, 1), (x_2, 2), \dots, (x_{n-1}, n-1)$  in which the  $n$  elements are each paired with an arbitrary, unique integer from 0 to  $n-1$ , and produces a set of  $n$  pairs  $(x, i)$ , where  $x$  is one of the input elements, and  $i$  is its position in the sorted order of the elements.

We begin by implementing some standard techniques in our model. However, they are not optimal, either in our model or in more familiar models of parallel computation. We then offer an algorithm that uses communication cost close to the optimal and polylogarithmic elapsed cost.

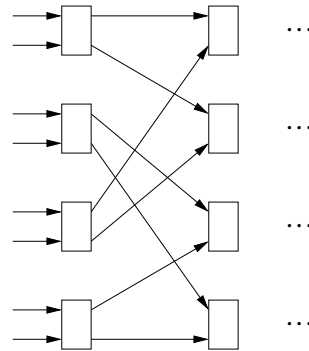
### 4.1 Batcher Sort

For a sorting algorithm that is guaranteed to be  $O(n \log n)$  and works well with non-main-memory data, you usually think of some variant of merge-sort. However, in our model, merging large sorted files is problematic, because merging appears sequential. Obvious implementations would imply  $O(n)$  elapsed communication/processing costs.

Known parallel sorting algorithms allow us to get close to the optimal  $O(\log n)$  elapsed processing cost. For example, we can use a Batcher sorting network [4] to get  $O(\log^2 n)$  elapsed cost. A similar

approach is to use the  $O(n \log^2 n)$  version of Shell sort [21] developed by V. Pratt ([19] or see [15]).

While we shall not go into the details of exactly how Batcher's sorting algorithm works, at a high level, it implements merge sort with a recursive parallel merge, to make good use of parallelism. The algorithm is implemented by a sorting network of comparators (devices that take two inputs, send the higher out on one output line and the lower out on the second output line), a simple instance of which is suggested by Fig. 2. There are  $O(\log^2 n)$  columns of  $n/2$  comparators each. The  $n$  input elements enter in any order on the left and emerge in sorted order on the right. For convenience, we shall describe the algorithm for the case where  $n$  is a power of 2.



**Figure 2: Part of a Batcher sorting network**

There is a simple pattern that determines which comparator outputs are connected to which comparator inputs of the next column. If we think of the outputs of the comparators as numbered in order  $0, 1, \dots, n-1$ , then the connections from one column to the next are determined by writing the integer number of the output in binary, and rotating the binary number right by  $k$  bits, where  $k$  depends on the column. This operation is called a  $k$ -shuffle.

**EXAMPLE 4.1.** *Suppose  $n = 8$  and  $k = 2$ . Output 5 is represented by 101 in binary. If we rotate right 2 places, the number becomes 011. That is, if the connection between one column and the next is governed by  $k = 2$ , then the output number 5 becomes the input number 3 at the next column.*

*Output 1 is represented by 001, and if we rotate right 2 places, the number becomes 010. That is, output 1 becomes input 2. There is an important connection between outputs 5 and 1 in this case. Since they differ only in their last bit after rotation, they will be inputs to the same comparator at the next column.*  $\square$

Now, let us implement Batcher sort in our model. Each column of comparators will be implemented by  $n/s$  processes, where as usual,  $n$  is the number of elements to be sorted, and  $s$  is the limit on input size for a process. The only constraint on which elements are assigned to which process are that the two inputs to a comparator must be sent to the same process. As we saw in Example 4.1, numbers that differ only in the bit that will become leftmost after a  $k$ -shuffle are sent to the same process. Thus, for each value of  $k$ , we must choose a hash function  $h_k$  that does not depend on the  $k+1$ st bit from the right end. If the Batcher sort moves data from one column to the next using a  $k$ -shuffle, then we send the element on the  $i$ th output of a process to the  $h_k(i)$ th process for the next column.

### 4.2 Analysis of Batcher Sort

As we stated, there are  $O(\log^2 n)$  columns in Batcher's network. Simulating the network in our model requires that all  $n$  elements be communicated from column to column, so the total communication

is  $O(n \log^2 n)$ . Paths in the connection graph are  $O(\log^2 n)$  long, so the elapsed communication is  $O(\log^2 n)$ .

We can execute a process in  $O(s)$  time. The justification is that its  $s$  inputs can be arranged in linear time so elements whose associated integers differ only in the last bit can be compared, and their associated integers swapped if they are out of order. We can then apply to each element's integer the proper hash function to implement the next shuffle. Since there are  $n/s$  processes per column of the network, and  $O(\log^2 n)$  columns, the total processing time is  $O(n \log^2 n)$ . Likewise, we can argue that the elapsed processing time is  $O(s \log^2 n)$ . Thus we have proved for Batcher sort (all logarithms are in base 2):

**THEOREM 4.2.** *The algorithm in Section 4.1 sorts  $n$  elements with the following costs: a) total communication  $O(n \log^2 n)$ , b) elapsed communication  $O(\log^2 n)$ , c) total processing time is  $O(n \log^2 n)$  and d) elapsed processing time  $O(s \log^2 n)$ .*

### 4.3 Sorting Under Hadoop

It turns out that because of the way Hadoop is implemented, it offers a trivial way to sort.<sup>2</sup> The input to any Reduce process is always sorted by key. Thus, a single Map process and a single Reduce process, both implementing the identity function, will result in the input being sorted. However, this algorithm is inherently serial, and is not a solution to the problem of sorting very large sets.

## 5. Efficient Sorting Algorithms

While Batcher or Shell sorts implemented in our model can be fairly efficient and simple, they are not the best we can do. We shall show next an algorithm that is close to the theoretical lower bound for sorting in this model. We begin by discussing a particular algorithm that sorts  $n = s^{3/2}$  distinct elements with a network of processes with constant depth, and then show how to extend the idea to sort any number of elements with polylogarithmic network depth.

Note, however, that  $n = s^{3/2}$  is sufficient for most practical purposes. For example, if  $s$  is ten gigabytes, then the first algorithm can sort a petabyte of data.

### 5.1 The Constant-Depth Algorithm

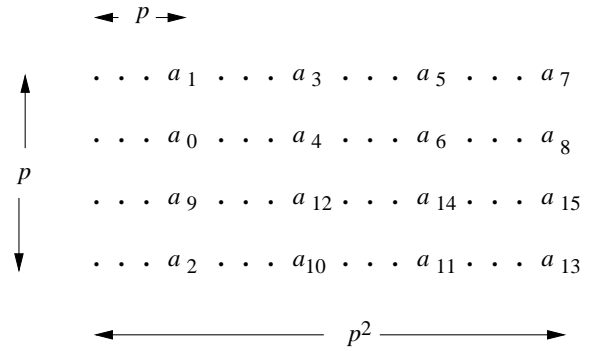
For convenience, we shall assume that communication sizes are measured in elements rather than in bytes. Thus,  $s$  is the number of elements (with their attached ordinal numbers) that can be input or output to any process, and  $n$  is the total number of elements to be sorted. The algorithm uses a parameter  $p$ , and we define  $n = p^3$  and  $s = p^2$ . The following method sorts  $n = p^3$  elements, using  $O(p^3)$  communication, with communication cost for each process equal to  $O(p^2)$ . We begin with an outline, and later discuss how the work is assigned to processes and how the processes communicate.

**Step 1:** Divide the input into  $p$  lines, each having  $p^2$  elements. Sort each line using a single process. Recall that "sorting" in this context means attaching to each element an integer that indicates where in the sorted order, the element would be found. The output is a set of (element, ordinal) pairs, rather than a sorted list.

**Step 2:** For each line, the  $p$ -th,  $2p$ -th,  $3p$ -th, and so on up to element  $p^2$  is a *sentinel*. The number of sentinels in each line is  $p$ , and the total number of sentinels is  $p^2$ . We sort the sentinels, using another process. Suppose the sentinels in sorted order are  $a_0, a_1, \dots, a_{p^2-1}$ .

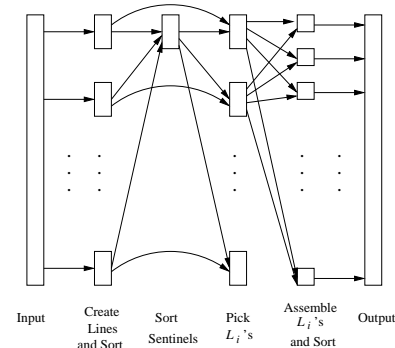
**EXAMPLE 5.1.** *Figure 3 illustrates a possible outcome of Steps 1 and 2 for the case  $p = 4$ . If there were a large number of randomly ordered elements to be sorted, we would expect that the first  $p$  sentinels would be the first sentinels from each line, in some order. However, it is in principle possible for the sentinels to be unevenly*

<sup>2</sup>Thanks to Ragho Murthy for making this point.



**Figure 3: Sorting: lines and sentinels**

distributed. For instance, we have suggested in Fig. 3 that the first sentinel of the third line follows all the sentinels of the first two lines. On the other hand, since the lines are sorted, the sentinels within a line must appear in sorted order.  $\square$



**Figure 4: The processes involved in the sorting algorithm of Section 5.1**

Figure 4 illustrates the network of processes that constitute the entire sorting algorithm. The first two columns of processes correspond to the first two steps. There are two more steps, corresponding to the last two columns of Fig. 4. Their goal is to construct the  $p^2$  sets of elements  $L_0, L_1, \dots, L_{p^2-1}$  such that all elements in  $L_i$  are less than or equal to  $a_i$  and greater than  $a_{i-1}$  (unless  $i = 0$ , in which case there is no lower bound). In Step 4 we have one process for each  $L_i$ , but we need to divide the lines among the processes, and that is the job of Step 3. Before proceeding, consider the following example.

**EXAMPLE 5.2.** *Let us consider how Steps 3 and 4 would work if the outcome of Step 2 were as seen in Fig. 3. For the first line, the elements of subgroup  $L_0$  (i.e., those elements that are at most  $a_0$ ) are found among the first  $p = 4$  elements of the line (i.e., those up to and including  $a_1$ , although surely  $a_1$  is not among them). For the second line, the elements of subgroup  $L_0$  are exactly all  $p = 4$  first elements of the line.*

*Now, consider  $L_5$  (i.e., those elements that are larger than  $a_4$  but no greater than  $a_5$ ). They are found among the third  $p = 4$  elements of the first and second lines, among the first group in the third line, and among the second group of the last line. For the first line, all subgroups  $L_i$  for  $i > 7$  are empty.  $\square$*

**Step 3:** The goal of this step is to partition each line into the various  $L_i$ 's. As seen in the third column of Fig. 4, there are  $p$  processes, one for each line. In addition to the sorted line as input, each of these

processes takes the entire sorted list of sentinels. Note that the input size is thus  $2p^2 = 2s$ , but we have, for convenience in description, allowed a process to take  $O(s)$  input rather than exactly  $s$ . The process for line  $j$  determines to which  $L_i$  each element belongs. It can do so, since it sees all the sentinels. The process also determines, for each sentinel  $a_i$ , how many elements in line  $j$  are less than  $a_i$  (this count will be  $i$  less than the position of  $a_i$  in the sorted order of the line and the sentinels, merged). Finally, each process sends to the process for  $L_i$  the set of elements from the line that belong to  $L_i$  and the count of elements less than  $a_i$ .

**Step 4:** With this information, the process for  $L_i$  (shown in the fourth column of Fig. 4) can determine the order of elements in  $L_i$ . Since it knows how many elements in the entire set to be sorted are less than  $a_i$ , it can also determine the position, in the entire sorted list, of each member of  $L_i$ . Finally, we note that, since no one line can have more than  $p$  elements belonging to any one  $L_i$  (and usually has many fewer than  $p$ ), the input to the process for  $L_i$  can not be larger than  $p^2 = s$ .

In summary, the sorting algorithm of this section is the following:

#### ALGORITHM CONSDEP-SORTING

**Step 1:** Divide the input into  $p$  lines, each having  $p^2$  elements. Sort each line using a single process.

**Step 2:** For each line, the  $p$ -th,  $2p$ -th,  $3p$ -th, and so on up to element  $p^2$  is a *sentinel*. Sort the sentinels, using another process.

**Step 3:** For each initial line construct a process with input this line and the sentinels. Sort the input of each process. Suppose the sentinels in sorted order are  $a_0, a_1, \dots, a_{p^2-1}$ . The output of each process is distributed as follows: The elements that are between  $a_{i-1}$  and  $a_i$  are fed to the process labeled  $L_i$ , for all  $i = 0, 1, \dots, p^2 - 1$ . Each process also determines, for each sentinel  $a_i$ , how many elements in line  $j$  are less than  $a_i$ ; this count is  $i$  less than the position of  $a_i$  in the sorted order of the line and the sentinels, combined.

**Step 4:** Each process  $L_i$  sorts its elements and computes their global order as follows. The count of  $a_i$  is the sum of the counts fed from the processes of Step 3. The global rank of an element  $x$  in process  $L_i$  is the count for  $a_i$  minus the number of elements of  $L_i$  that follow  $x$  in the order of elements at process  $L_i$ .

## 5.2 Communication and Processing Costs of CONSDEP-SORTING

The communication and processing costs for the processes in each of the columns of Fig. 4 are easy to analyze. We can then sum them to get the total costs. All paths through the network of Fig. 4 have exactly one process from each column, so elapsed costs are also easy to obtain. This analysis and the correctness of the algorithm are formally stated in the theorem below:

**THEOREM 5.3.** *Suppose the communication cost allowed for each process is less than  $s = p^2$ . Then algorithm CONSDEP-SORTING constructs a network of processes which correctly sorts  $n = p^3$  elements. Moreover, the costs of the network are: a) has  $O(n)$  total communication and  $O(n \log s)$  total processing cost, and b) has elapsed communication cost  $O(s)$  and elapsed processing cost  $O(s \log s)$ .*

**PROOF.** First it is clear that the algorithm uses only processes whose input and output do not exceed  $O(s)$ . The correctness of the algorithm is a consequence of the following facts: a) the sentinels of all lines are sorted in Step 2, b) the elements from each line that lie between two consecutive sentinels are found in Step 3 and c) each process in Step 4 sorts exactly all elements that lie between two consecutive sentinels, and places them in their correct global position. The analyses for the costs are done by column (or equivalently, step) of the algorithm:

1. For Step 1, each process takes input of size  $p^2 = s$  and makes output of size  $O(s)$ . Thus, the elapsed communication is  $O(s)$ .

Since it is sorting  $O(s)$  elements, presumably by an efficient sort, we take the elapsed processing time to be  $O(s \log s)$ . Thus, for this step the total communication cost for the  $p$  processes is  $O(ps) = O(n)$ . The total processing cost is  $O(ps \log s) = O(n \log s)$ .

2. Step 2 is similar to Step 1. The elapsed costs are the same, but since there is only one process instead of  $p$ , the total communication is only  $O(s)$ , and the total processing cost is  $O(s \log s)$ .
3. In Step 3, the processes take input of size  $O(p^2) = O(s)$  and make output of the same size. Thus, the elapsed communication cost is  $O(s)$ . One way to achieve the goals of this step is to merge the elements of the line with the sentinels, so we assert the elapsed processing cost is  $O(s)$ . Since there are  $p$  processes at this step, the total communication cost is  $O(n)$  and the total processing cost is  $O(n)$  as well.
4. Step 4 must be analyzed more carefully. There are  $p^2 = s$  processes, rather than  $p$  processes as in Steps 1 and 3. However:
  - (a) The sum of their input sizes is  $O(n) = O(p^3) = O(s^{3/2})$ , because each original input goes to only one of the processes. The additional information sent to each process for Step 4 is one count of elements below its sentinel from each of the processes of Step 3. That information is only  $O(p)$  additional input, or  $O(n)$  total for all  $p^2$  processes. Thus, the total communication for Step 4 is  $O(n)$ . We already observed that no one process at Step 4 gets more than  $O(s)$  input, so that is the elapsed cost upper bound for this step.
  - (b) For the processing time, each of the  $n$  elements is involved in the sorting that goes on at one of the processes of Step 4. There can be at most  $O(s)$  elements at one process, so the total processing time is  $O(n \log s)$ . The elapsed time at any one process is (at most)  $O(s \log s)$ .

If we add the costs of the four steps, we find that the total communication cost is  $O(n)$ . The total processing cost is  $O(n \log s)$ ; the elapsed communication cost is  $O(s)$ , and the elapsed processing cost is  $O(s \log s)$ .  $\square$

Note that since  $n \leq s^{3/2}$  was assumed,  $O(\log s)$  is the same as  $O(\log n)$ , so the total processing cost is  $O(n \log n)$ , as would be expected for an efficient sort. The elapsed processing cost is significantly less, because we are doing much of the work in parallel. However, the algorithm is not as parallel as the best sorts, because we are constrained to take fairly large chunks of data and pass them to a single process. Of course, if the process itself were executed on a multicore processor, we would get additional speedup due to parallelism that is not visible in our model.

## 5.3 Dealing With Large Block Size

There is one last concern that must be addressed to make the analysis of Section 5.2 precise. Recall that we assume there is a lower bound  $b$  on block size that may be significant in Step 4. There, we have  $p^2$  processes, each with an average of  $O(p) = O(\sqrt{s})$  input. If  $b > \sqrt{s}$ , as might be the case in practice, we could in fact require  $O(bp^2) > O(n)$  communication at Step 4.

Fortunately, it is not hard to fix the problem. We do not have to use  $p^2$  distinct processes at Step 4. A smaller number of processes will do, since we may combine the work of several of the processes of Step 4 into one process. However, combining the processes arbitrarily could cause one of the combined processes to get much more than  $s$  input, which is not permitted. Thus, we might need to introduce another columns of processes between Steps 3 and 4. Each process

reads the counts of elements below each of  $p$  consecutive sentinels, from each of the  $p$  lines, and combines them as needed into fewer than  $p$  groups of sentinels, as evenly as possible.

#### 5.4 Comparison With Batcher Sort

The sorting algorithm of Section 5.1 works for only a limited input size  $n$ ; that limit depends on the value chosen for  $s$ . However, for a realistic choice of  $n$  and  $s$ , say  $s = 10^{10}$  and  $n = 10^{15}$ , the algorithm requires a network of depth only four. In comparison, the Batcher network has depth over 1000 for  $n = 10^{15}$ . While we can undoubtedly combine many layers of the Batcher network so they can be performed by a single layer of processes with input size  $s = 10^{10}$ , it is not probable that we can thus get even close to four layers.

Moreover, we shall see in what follows that we can build networks for arbitrarily large  $n$  and fixed  $s$ . The communication cost for this family of networks is asymptotically better than that of Batcher sort.

#### 5.5 A Lower Bound for Sorting

There is a lower bound of  $\Omega(n \log_s n)$  for sorting in the model we have proposed which is proven in Theorem 5.4. The argument is a simple generalization of the lower bound on comparisons.

**THEOREM 5.4.** *Suppose the upper bound on the communication cost for each process is  $s$ . Then any network in our model that sorts  $n$  elements has total communication cost  $\Omega(n \log_s n)$ .*

**PROOF.** Once a process reads a fixed  $s$  elements, then all it can do is learn the order of these elements. That information reduces the number of possible orders of  $n$  elements by at most a factor of  $s!$ . You can assume the true order of the  $n$  elements is the one that leaves the most remaining possibilities after executing a process that examines  $s$  elements. Thus, the number of processes,  $p$ , must satisfy  $(s!)^p \geq n!$ , or  $p \geq (\log n!)/(\log s!)$ .

If we use the asymptotic Stirling approximation  $\log x! = x \log x$ , and we note that each process has  $s$  communication cost, we obtain the lower bound on communication  $\Omega(s(n \log n)/(s \log s)) = \Omega(n(\log n)/(\log s)) = \Omega(n \log_s n)$ .

One detail must be added to the proof. We have assumed that the least communication occurs when each process gets the full  $s$  inputs. To see why this is correct, suppose we have some number of processes whose inputs consist of  $s_1, s_2, \dots$  elements, where the sum  $s_1 + s_2 + \dots$  is fixed at  $k$  (i.e., the total communication of the processes is fixed). We constrain each  $s_i$  to be at most  $s$ . Then the product  $(s_1!)(s_2!) \dots$  consists of  $k$  integer factors. It is easy to see that, given the factors must form factorials, and none can exceed  $s$ , that the greatest product occurs when each  $s_i$  is  $s$ , and there are as few as possible. Since we need  $n! \leq (s_1!)(s_2!) \dots$ , we minimize communication by maximizing the product, i.e., by making all processes take  $s$  inputs.  $\square$

#### 5.6 Extending the Constant-Depth Sort Recursively

We have so far developed a network, Fig. 4, that uses processes with  $O(s)$  input/output each, and sorts  $s^{3/2}$  elements. We can use this network recursively, if we “pretend” that  $s$  is larger, say the true  $s$  raised to the  $3/2$ th power, and implement the network for that larger  $s$ . Each of the boxes in Fig. 4 can be implemented by a sorting algorithm. Thus, if we are to sort, say,  $s^{9/4}$  elements using boxes that we pretend can take  $s^{3/2}$  inputs each, then we must replace each box in Fig. 4 by the network of Fig. 4 itself.

It should be evident that Steps 1 and 2 of Algorithm Cons-Dep-Sorting (columns 1 and 2 of Fig. 4) can be implemented by sorting networks. The same is true for Step 4, although there is a subtlety. While Steps 1 and 2 use boxes that sort  $s$  elements, Step 4 (column

4) uses boxes that sort variable numbers of elements, up to  $s$ . However, we shall imagine, in what follows, that Step 4 uses  $\sqrt{s}$  boxes of  $s$  inputs each (see Appendix for a formal proof). Informally, the communication cost of these boxes, when replaced by networks, is no greater than that of  $\sqrt{s}$  sorters of  $s$  elements each, because:

1. The communication cost of sorting  $n$  elements is surely at least linear in  $n$ ,
2. No box in column 4 uses more than  $s$  input, and
3. The sum of the input/output sizes of all the boxes in column 4 is  $s^{3/2}$ .

We observe, therefore, that if we have a sorting network for  $n$  elements that sorts with communication cost  $C(n)$ , then we can build a sorting network that sorts  $n^{3/2}$  elements, using communication at most  $C(n^{3/2}) = (3\sqrt{n} + 1)C(n)$ . The justification is that there are  $\sqrt{n}$  boxes in each of columns 1 and 3, one box in column 2, and the communication in column 4 is no more than the communication of  $\sqrt{n}$  boxes; i.e., there are  $3\sqrt{n} + 1$  boxes or equivalents, each of which is replaced by a sorting network of communication cost  $C(n)$ .

The following lemma is useful and easy to prove:

**LEMMA 5.5.** *The solution to the recurrence*

$$C(n^{1+b}) = an^b C(n)$$

with a basis in which  $C(s)$  is linear in  $s$ , is  $C(n) = O(n \log_s^u n)$ , where  $u = \log a / \log(1 + b)$ .

Moreover, Lemma 5.5 holds even if we add low-order terms (i.e., terms that grow more slowly than  $n \log_s^u n$ ) on the right. The same holds if we add a constant to the factor  $an^b$  on the left, as is the case with the recurrence  $C(n^{3/2}) = (3\sqrt{n} + 1)C(n)$  discussed in connection with sorting.

For the latter recurrence, we have  $a = 3$  and  $b = 1/2$ . Thus, we achieve a solution  $C(n) = O(n \log_s^u n)$ , where

$$u = \log_2 3 / \log_2(3/2) = 2.7$$

Note that for fixed  $s$ ,  $C(n) = O(n \log_s^{2.7} n)$  is asymptotically worse than Batcher sort. However, we can do better, as we shall see next.

#### 5.7 A Merging Network

The improvement to our recursive sorting network comes from the fact that in Step 3, we do not need a sorting network; a merging network will do. If we can replace the third column in Fig. 4 by merging networks, then we replace the constant  $a = 3$  in Lemma 5.5 by  $a = 2$ , and we add to the right side of the recurrence a low-order term representing the cost of the merging networks, which does not affect the value of  $u$ . The resulting value of  $u$  is  $\log_2 2 / \log_2(3/2) = 1.7$ . The communication cost  $C(n) = O(n \log_s^{1.7} n)$  beats the Batcher network asymptotically.

To complete the argument for  $O(n \log_s^{1.7} n)$  communication-cost sorting, we need to show how to merge efficiently. For a basis, we can surely merge two sorted lists of length  $s$  each, in a single process with  $O(s)$  communication cost. The recursion uses a network with essentially the same structure as Fig. 4, and we shall explain this algorithm in four steps, corresponding to the four columns in that figure.

We shall describe a network  $M_n$  that merges two sorted lists of  $n$  elements each. Recall that by “sorted list,” we mean that each list is a set of pairs  $(x, i)$ , where element  $x$  is asserted to be the  $i$ th in the list. This set is “sorted,” in the sense that if  $(x, i)$  and  $(y, j)$  are two pairs, and  $x < y$ , then it must be that  $i < j$ .

Suppose we have a way to merge two sorted lists of length  $s$  each, e.g., a single process if  $O(s)$  is the limit on communication for a process. Let  $n = s^2$ . We build a network of processes resembling Fig. 4,

where the four columns represent the following four steps.

#### ALGORITHM RMERGE( $s^2$ )

**Step 1:** Suppose the input is two lists,  $S_1$  and  $S_2$ , of length  $n = s^2$  each. Choose the elements of  $S_1$  and  $S_2$  at positions  $0, s, 2s, \dots$ ; we call them merge-sentinels. Note that, since each list comes with elements attached to positions, this step only requires filtering the input; no merging or sorting is necessary. We simply divide each of the lists among  $s$  processes. Each process takes input of size  $s$  and identifies the merge-sentinels by their associated ordinals, which are divisible by  $s$ .

**Step 2:** Each of the processes from Step 1 passes its sentinels to a single merge process. Technically, the ordinals of the sentinels need to be divided by  $s$ , so the input to the process of column 2 is truly a sorted list, with ordinals  $0, 1, 2, \dots$ .

Our goal for the last two steps is to have a process  $P_{ab}$  for each pair  $a$  and  $b$  of consecutive merge-sentinels, that merges all the elements in the range from  $a$  to  $b$ .<sup>3</sup> These processes are the boxes in the fourth column. To give these processes all the input they might need, the third column consists of processes that, as in Fig. 4, each take the data from one of the processes of column 1, plus the merged sentinels from column 2. For each  $a$  and  $b$ , this process finds all the elements from the lists  $S_1$  and  $S_2$  that lie between  $a$  and  $b$ , and passes those to  $P_{ab}$ . Note that no  $P_{ab}$  can receive more than  $2s$  inputs, which is fine, because we allow  $O(s)$  communication for a process. Thus, we complete Algorithm RMerge with:

**Step 3:** There is one process for each of the processes of Step 1. A process of this step takes the same input as the corresponding Step-1 process, plus the sorted sentinels from the process of Step 2. The inputs, from  $S_1$  or  $S_2$  are merged with the sentinels. The process then distributes each element from one of the lists  $S_1$  or  $S_2$  that it finds in its input to the proper process  $P_{ab}$  in Step 4. It also transmits the number of elements it finds lower than  $a$  among its inputs.

**Step 4:** Each process  $P_{ab}$  determines the number of elements lower than  $a$  among the two lists  $S_1$  and  $S_2$  by adding the reports from each of the processes of Step 3, and assigns an ordinal to each element in the range  $a$ -to- $b$ .

Now, let us see how algorithm RMERGE( $n$ ) constructs a network for any  $n$ . Then the processes are really merging networks. In this case, merge-sentinels are taken at positions  $0, \sqrt{n}, 2\sqrt{n}, \dots$ . First, only Steps 2 and 3 require merging networks. Step 1 only involves filtering, and so its communication is  $O(n)$  as was in the case we had to sort only  $s^2$  elements. Also, Step 4 involves adding  $\sqrt{n}$  counts in  $\sqrt{n}$  separate networks, which can be done by a network with  $O(n)$  communication, again regardless of  $n$  (the depth of the network will grow as  $\log_s n$ , since each process is limited to  $s$  input). It also involves reassigning ordinals by addition of a base value, which requires no additional communication. Thus, Steps 1 and 4 require  $O(n)$  communication.

Let  $M_n$  denote a merging network that merges two lists of  $n$  elements. Step 2 requires one merging network  $M_{\sqrt{n}}$  while Step 3 requires  $\sqrt{n}$  merging networks  $M_{\sqrt{n}}$ . Let  $C_M(n)$  be the communication required to merge lists of length  $n$ . Then Steps 2 and 3 together require  $(\sqrt{n} + 1)C_M(\sqrt{n})$  communication. The complete recurrence is thus:

$$C_M(n) = (\sqrt{n} + 1)C_M(\sqrt{n}) + an$$

for some constant  $a$ . Note that Lemma 5.5 does not apply here, because the forcing function  $an$  is not low-order, compared with the solution  $C_M(n) = O(n)$  that this lemma would imply. However, we can solve the recurrence, with the basis  $C_M(s) = O(s)$ , by repeated expansion of the right side, to get the solution:

<sup>3</sup>As a special case, if  $b$  is the first sentinel, then  $a$  is “minus infinity.”

**THEOREM 5.6.** *Algorithm RMERGE constructs a network  $M_n$  of processes which correctly merges two sorted lists of  $n$  elements each. Moreover, the costs of the network are: a) total communication cost  $O(n \log_2 \log_s n)$  b) elapsed communication cost  $O(s \log_2 \log_s n)$  c) total processing cost  $O(n \log_2 \log_s n)$  d) elapsed processing cost  $O(s \log_2 \log_s n)$ .*

**PROOF.** Correctness is proved in a similar way as in Theorem 5.3.

a) The argument is as discussed above.

b) The elapsed communication is given by

$$C_e(n) = C_e(\sqrt{n}) + O(1), C_e(s) = O(s)$$

with solution  $C_e(n) = O(s \log_2 \log_s n)$ .

c) The total processing cost comes from the same recurrence as the communication and is  $O(n \log_2 \log_s n)$ . The reason is that all operations performed by processors are linear in their input size.

d) Similarly, the elapsed processing cost is the same as the elapsed communication.  $\square$

## 5.8 Merge Sort

We can use the merge algorithm to build the obvious implementation of merge sort. This will affect Step 4 (Step 2 has low costs anyway) where we need merge a number of sorted lists. We do that in a number of stages; at the  $i$ th stage we merge two lists of length  $s2^{i-1}$ . The resulting algorithm, which we shall not discuss in detail, has costs that are each  $\log_2(n/s)$  times the costs given in Theorem 5.6.

For example, the communication cost of merge-sort algorithm is  $O(n \log_2(n/s) \log_2 \log_s n)$ . This expression is asymptotically less than the communication cost we shall give in Theorem 5.7 for sorting based on Algorithm CONSDep-SORTING, which is  $O(n \log_s^{1.7} n)$ . However, for a reasonable value of  $s$ , such as  $10^9$ , and even an astronomically large value of  $n$ , like  $n = s^3 = 10^{27}$ ,  $\log_s^{1.7} n$  is much less than  $\log_2(n/s) \log_2 \log_s n$ . Note that  $\log_s n = 3$  for our example, while  $\log_2(n/s)$  is about 60. Also this comparison makes sense although the claims are in terms of  $O()$  because the hidden constants are small and independent of  $s$ .

## 5.9 Completing the Recursive Sort

We can now complete the plan outlined in Section 5.6.

#### ALGORITHM RECURS-SORTING

1. For any  $n$ , we can build a network as in Fig. 4 to sort  $n^{3/2}$  elements if we replace each box in columns 1 and 2 by recursively defined sorting networks for  $n$  elements.
2. For column 3, we use the merge network  $M_n$  just discussed.
3. For column 4 we use  $n$  sorting networks of the appropriate size.

For column 4, we rely on the lemma in the Appendix to justify that the cost cannot be greater than that of  $\sqrt{n}$  networks that sort  $n$  elements each. However, we must also pay attention to the fact that the various boxes in column 4 represent networks of different, and unpredictable, sizes. Thus, we need to precede each one by a tree of processes that sums the counts for each of these processes. In analogy with Step 4 of Algorithm RMERGE, the total communication cost of these networks cannot exceed  $O(n^{3/2})$  and their elapsed communication cost cannot exceed  $O(s \log_s n)$ . As in Algorithm RMERGE, the recursive case must use a network of processes of depth  $\log_s n$  to sum the counts of elements below each sentinel, but this additive term is low-order for this algorithm (although it is significant in RMERGE). The communication cost of the network for sorting  $n^{3/2}$  elements is thus given by

$$C(n^{3/2}) = (3\sqrt{n})C(n) + \text{lower order terms}$$

from which we conclude  $C(n) = O(n \log_s^{1.7} n)$  by Lemma 5.5. The four cost measures are summarized in the following theorem.

**THEOREM 5.7.** *Algorithm RECURS-SORTING constructs a network of processes which correctly sorts  $n$  elements. Moreover, the costs of the network are: a) total communication cost  $O(n \log_s^{1.7} n)$  b) elapsed communication cost  $O(s \log_s^{2.7} n)$  c) total processing cost  $O(n \log_s^{1.7} n \log_2 s)$  d) elapsed processing cost  $O(s \log_s^{2.7} n \log_2 s)$ .*

**PROOF.** Correctness is proved in exactly the same way as in Theorem 5.3.

a) The argument is as discussed above.

b) To sort  $n^{3/2}$  elements, the longest paths go through three sorting networks for  $n$  elements and either a merging network with elapsed communication  $O(s \log_2 \log_s n)$ , or the summing tree for implementing column 4, which has longest paths of length  $O(s \log_s n)$ , as discussed above. Thus, the recurrence for  $C_e$ , the elapsed communication cost, is

$$C_e(n^{3/2}) = 3C_e(n) + O(s \log_s n)$$

which, with the basis that  $C_e(s) = O(s)$ , gives us the solution stated.

c) The total processing cost will be  $\log_2 s$  times the total communication cost because the processes that sort  $s$  elements have  $O(s)$  communication but  $O(s \log_2 s)$  processing time.

d) The same argument as for (c) applies to the elapsed processing cost.  $\square$

## 6. Conclusion

We introduced a new computation model that reflects the assumptions of the map-reduce framework, but allows for networks of processes other than Map feeding Reduce. We illustrated the benefit of our model by showing how to improve the computation of the 3-way join and by developing algorithms for merging and sorting. The cost measures by which we evaluate the algorithms are the communication among processes and the processing time, both total over all processes and elapsed (i.e., exploiting parallelism).

The work in this paper initiates an investigation into the possibilities of using efficiently the new paradigm of rack-based computing. In that respect, both implementations of the extended framework, and theoretical research for developing algorithms for other problems that involve data-intensive computations are seen as future research. Some worthwhile directions include:

1. Examine the best algorithms within the model for other common database operations. For example, if we can sort efficiently, we can eliminate duplicates efficiently. Are there even better ways to eliminate duplicates?
2. Our study of multiway joins was limited to the join of three relations in a chain. The case studied surely generalizes to allow sets of attributes in place of  $A$ ,  $B$ ,  $C$ , and  $D$ . But what happens when the join is cyclic? What happens when there are more than three factors in the join, in the cases of a) a chain b) the general acyclic case c) the cyclic case?
3. The model itself should not be thought of as fixed. For example, we mentioned how Hadoop sorts inputs to all Reduce processes by key. There are highly efficient sorts for the amounts of data we imagine will be input to the typical process. What is the influence on algorithms of assuming all processes can receive sorted input “for free”?
4. One of the major objections of [11] was that indexing was not available under map-reduce. Can we build and use indexes in an environment like that of GFS or HFS, where block sizes are enormous? Or alternatively, is the model only appropriate for

full-relation operations, where there is no need to select individual records or small sets of records?

**Acknowledgment** We would like to thank Raghotham Murthy, Chris Olston, and Jennifer Widom for their advice and suggestions in connection with this work.

## 7. References

- [1] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proc. 13th Int'l Conf. on Very Large Data Bases*, pages 255–266, 1987.
- [2] Apache. Hadoop. <http://hadoop.apache.org/>, 2006.
- [3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. *SIGMOD Rec.*, 26(2):243–254, 1997.
- [4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
- [5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine, 1998.
- [6] P. Buonadonna, J. Coates, S. Low, and D. E. Culler. Millennium sort: a cluster-based application for windows nt using dcom, river primitives and the virtual interface architecture. In *WINSYM '99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 9–9, Berkeley, CA, USA, 1999. USENIX Association.
- [7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
- [8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.
- [9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
- [10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [11] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. <http://www.databasemagazine.com/2008/01/mapreduce-a-major-step-back.html>, 2008.
- [12] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.
- [13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, 2003.
- [14] J.-W. Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981.
- [15] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.
- [16] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. Alphasort: A risc machine sort. In *SIGMOD Conference*, pages 233–242, 1994.
- [17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
- [18] C. H. Papadimitriou and M. Sipser. Communication complexity. *J. Comput. Syst. Sci.*, 28(2):260–269, 1984.
- [19] V. Pratt. Shellsort and sorting networks. Ph.D. thesis, Stanford University, 1971.
- [20] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD Conference*, pages 365–376, 2007.
- [21] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.
- [22] C. D. Thompson. Area-time complexity for vlsi. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 81–88, New York, NY, USA, 1979. ACM.
- [23] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Ann. Math. Artif. Intell.*, 3(2-4):331–360, 1991.
- [24] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

## APPENDIX

### A. Formal Proof of the Bound on the Elapsed Communication of Step 4 in Section 5.6

LEMMA A.1. Suppose that a function  $C(n)$  is such that  $C(n) < n \log_s^{1.7} n$ . Suppose that  $n_1 + n_2 + \dots + n_n = n^{3/2}$  and

$$\max(n_1, n_2, \dots, n_n) \leq n$$

where the  $n_i$ 's are all positive integers. Then, the following holds:

$$C(n_1) + C(n_2) + \dots + C(n_n) < n^{3/2} \log_s^{1.7} n$$

PROOF. The proof essentially emanates from the following claim:

Claim 1: If  $n_1 + n_2 + \dots + n_n = n^{3/2}$  and  $\max(n_1, n_2, \dots, n_n) \leq n$  where  $n_i$  are all positive integers then

$$n_1 \log_s^{1.7} n_1 + n_2 \log_s^{1.7} n_2 + \dots + n_n \log_s^{1.7} n_n < n^{3/2} \log_s^{1.7} n$$

Proof of Claim 1.

The proof of Claim 1 uses Lemma A.2 and it is easy:

$$n_1 \log_s^{1.7} n_1 + n_2 \log_s^{1.7} n_2 + \dots + n_n \log_s^{1.7} n_n \leq$$

$$\sum n_i \log_s n_i \log_s^{0.7} n \leq 2/3 \lambda n^{3/2} \log_s n^{3/2} \log_s^{0.7} n$$

$$= (2/3)^{1.7} \lambda n^{3/2} \log_s^{1.7} n^{3/2} = (1/2) n^{3/2} \log_s^{1.7} n^{3/2}$$

Notice that  $(2/3)^{1.7} = 1/2$ .  $\square$

LEMMA A.2. If  $n_1 + n_2 + \dots + n_n = n^{3/2}$  and

$$\max(n_1, n_2, \dots, n_n) \leq n$$

where  $n_i$  are all positive integers then

$$n_1 \log_s n_1 + n_2 \log_s n_2 + \dots + n_n \log_s n_n < n^{3/2} \log_s n$$

PROOF. The proof is a straightforward consequence of Claim 2:

Claim 2: Under the assumption that  $1/n^{3/2} \leq p_i \leq 1/n^{1/2}$ , the following holds:

$$p_1 \log_s p_1 + p_2 \log_s p_2 + \dots + p_n \log_s p_n < -1/2 \log_s n$$

Before we prove Claim 2, we show how we finish the proof of the lemma based on the claim. We replace in the claim each  $p_i$  with  $n_i/n^{3/2}$  and using the fact that  $n_1 + n_2 + \dots + n_n = n^{3/2}$ , we get immediately the desired inequality.

Proof of Claim 2: We know from Information Theory that if we tend to "unbalance" the  $p_i$ s (i.e., change the value of the large ones to larger and the small ones to smaller while keeping their sum equal to 1) then the quantity on the left above becomes larger (actually the proof of this statement is not too complicated but we omit it because it is tedious and is contained in most textbooks). Thus, since we know that  $1/n^{3/2} \leq p_i \leq 1/n^{1/2}$ , we change all  $p_i$ s either to  $1/n^{1/2}$  or to  $1/n^{3/2}$ . Suppose we change  $x$   $p_i$ s to  $1/n^{1/2}$  and  $y$   $p_i$ s to  $1/n^{3/2}$ . We want to hold:

$$x + y = n \text{ and } x/n^{1/2} + y/n^{3/2} = 1. \text{ We solve and get:}$$

$$x = (n^{1/2} - 1)/(1 - 1/n) \text{ and } y = (n - n^{1/2})/(1 - 1/n)$$

Thus we have:

$$\begin{aligned} & p_1 \log_s p_1 + p_2 \log_s p_2 + \dots + p_n \log_s p_n \\ & < [(n^{1/2} - 1)/(1 - 1/n)]/n^{1/2} \log_s(1/n^{1/2}) \\ & + [(n - n^{1/2})/(1 - 1/n)]/n^{3/2} \log_s(1/n^{3/2}) = \\ & -(1/2)(1 - (1/n^{1/2}))/ (1 - 1/n) \log_s n - \end{aligned}$$

$$\begin{aligned} & (3/2)((1/n^{1/2}) - 1/n)/(1 - 1/n) \log_s n = \\ & -(1/2) \log_s n [(1 - (1/n^{1/2}) + (3/n^{1/2}) - 3/n)/(1 - 1/n)] \\ & < -(1/2)(1 + (1/n^{1/2})) \log_s n < -1/2 \log_s n \end{aligned}$$

This concludes the proof of Claim 2.  $\square$