A New Computation Model for Cluster Computing

Foto N. Afrati[*] and Jeffrey D. Ullman[†]

December 30, 2009

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. 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. We develop, in this model, an algorithm for sorting that has a worst-case running time better than the obvious implementations of parallel sorting.

1 Introduction

Search engines and other data-intensive applications process large amounts of data that need special-purpose computations. The most well-known problem is the sparse-matrix-vector calculation involved with PageRank [BP98], 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

Central to this stack is a *distributed file system* such as the Google File System (GFS) [GGL03] or Hadoop File System (HFS) [Apa06]. Such file systems are characterized by:

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

[*]School of Electrical and Computing Engineering, National Technical University of Athens, 15780 Athens, Greece

[†]Department of Computer Science, Stanford University, CA 94305

A powerful tool for building applications on such a file system is map-reduce [DG08] or its open-source equivalent Hadoop [Apa06]. 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 a Reduce function to combine all the values associated with one key and produce a single result for that key.

Map-reduce also 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 [CDG+08], or Yahoo!'s PNUTS [CRS+08], which are essentially object stores. At a still higher level, Yahoo!'s PIG/PigLatin [ORS+08] translates relational operations such as joins into map-reduce computations. [AU10] shows how to do multiway joins optimally using map-reduce.

1.2 Contribution of this Paper

There are concerns that as effective the map-reduce framework might be for certain tasks, there are issues that are not effectively addressed by this framework. [DS08] argues that the efficiency of a DBMS, as embodied in tools such as indexes, are missing from the map-reduce framework. Of note is the recent work on Clustera [DPR+08], which uses the same distributed file system as do map-reduce systems, but allows a far more general set of interacting processes to collaborate in a calculation.

If there is to be a serious study of algorithms that take advantage of distributed file systems and cluster computing, there must be a computational model that reflects the costs associated with computations in this new environment. The purpose of this paper is to propose such a model. We justify the aspects of the model in Section 2. In later sections we explore its consequences for the fundamental problems of sorting and merging.

In the proposed model, algorithms are acyclic networks of interconnected processes. We evaluate algorithms by the amount of data that must be moved among the processes, both in total and along any path in the network of processes. Map-reduce algorithms are a special case of algorithms in this model. However, many interesting algorithms are not of the map-reduce form.

The sorting task has often been used for testing computational environments about data management applications. For example, the goal in [ADADC+97, BCLC99, RSRK07] 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 [NBC+94, RSRK07].

In this paper, we offer sorting algorithms that follow our model. They are not well suited to the map-reduce form of computation, but can be expressed easily as a network of processes. In broad outline, we follow a Mergesort strategy, because that approach offers the opportunity to give a strong upper bound on running time. Competitors for the Terasort [Ter09] competition (large-scale sorting benchmarks) normally follow a Quicksort style of algorithm, which has a bad worst case that is irrelevant in such a competition.

- In Section 4.1, we offer an algorithm with four layers of processes (compared with two layers in map-reduce algorithms) that we argue can be used for sorting 10^{15} elements using today's commodity hardware.
- Based on similar ideas, we develop a merging algorithm in Section 4.7.
- We develop a sorting algorithm for any number n of elements in Sections 4.6 through 4.9. The algorithm has been designed to minimize the worst-case communication cost, which we show is $O(n \log^{1.7} n)$.
- In Section 4.5 we argue that there is an $\Omega(n \log n)$ lower bound on sorting in our model.
- In Section 4.8 we give an asymptotically better sorting algorithm with communication $O(n \log n \log^2 \log n)$. However, this algorithm may, in practice, use more communication than the earlier algorithm because the base of the logarithms involved is much smaller.

2 A Model for Cluster-Computing Algorithms

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 networks of processes. Each process operates on data stored as files. Processes are executed by processor nodes and are executed by a single processor. Here are the assumptions we make about files, processes and processors.

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. Typically, b is 64 Megabytes, and these blocks, often called *chunks*, are not related to the blocks used in conventional disk storage. Unusual assumptions about files are:

1. We assume the order of tuples in a file cannot be predicted. Thus, these files are really relations as in a relational DBMS.
2. 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.
3. 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.

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). There is also an upper limit on the amount of data a process can receive. This limit, which we denote by s, can represent one of several things, such as:

1. 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 relatively low-speed interconnect, e.g., gigabit Ethernet. In that environment, even loading main memory can take a minute, assuming no contention for the interconnect.
2. The amount of available main memory at a processor. This choice makes sense if we want to avoid the cost of moving data between main memory and local disk. Note that, depending on what a process does, only a fraction of the total main memory may be available to hold input data. For instance, a number of algorithms implemented on Clustera have used a quarter of a gigabyte as the value of s [DeW09].

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.

Managing Processes As is normal in any operating system, processes may be created, named, and referred to by other processes. We assume there is a *master process* that represents the algorithm as a whole and is responsible for creating at least some of the processes that constitute the algorithm. It is also possible for processes to be created dynamically, by other processes, so an algorithm's processes may not be determined initially. The operations we assume are available to manage processes and to transmit data among them are:

- *Create(P)*: If a process with name P does not exist, it is created. If P already exists, then this operation has no effect. We assume that the code to be executed by the process is implied by the name, so there can be no ambiguity about what a process is supposed to do.
- *Push(D,P)*: A process may send (part of) its output data, designated by D to the process named P.
- *Pull(D,P)*: A process Q may request some other process P to send Q (part of) its output data, designated by D. The data D need not be available at the time this command is executed by Q; the data will be sent when ready.

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 processor, but to 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.[1]

2.2 Cost Measures for Algorithms

An *algorithm* in our model is an acyclic graph of processes (a network of processes) with an arc from process P_1 to process P_2 if P_1 generates output that is (part of) the input to P_2. Processes cannot begin until all of their 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. Each process is characterized by:

- A *communication cost*, which we define to be the size n of the input data to this particular process.
- A *processing cost*, which is the running time of the process.

Typically, the processes themselves perform tasks that are of low computational complexity, so the processing cost of a process is dominated by the time it takes to get the data to the processor, i.e., the processing cost is proportional to the communication cost.

In addition, we assume a lower bound b and an upper bound s on the communication cost of each process. That is, an algorithm is not allowed to feed any process with data of size larger than s and when we count the communication of a process this count cannot be less than b (since we have to pay that price in our distributed file system). However, we allow bounds on the communication cost of a process to be adhered to in a "big-oh" sense; that is, a process can have $O(s)$ communication cost if s is the upper bound.

Our goal is to study efficient computation, so we need to measure the communication cost and processing cost of algorithms as a whole. Thus, we define:

- The *total communication cost* (*total processing cost*, respectively) is the sum of the communication (processing, respectively) costs of all processes comprising 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.

On the assumption that the processes themselves do a fairly elementary operation on their data, the time taken to deliver the data to a process dominates the total time taken by a process. If you are using a public cloud to do your computing, you "rent" time on processors, so the sum, over all processes, of the time taken by that process is what you pay to run the algorithm. Thus the total communication cost measures the monetary cost of executing your algorithm.

[1]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. We do not take into account, when we consider the cost measures for algorithms in Section 2.2, of the possibility that some time can be saved by not having to move data between certain pairs of processes. However, in the algorithms we propose, this additional cost issue would not result in an order-of-magnitude difference in the results.

On the other hand, elapsed communication cost measures the wall-clock time the algorithm requires. Again, we assume that the running time of any process is dominated by the time to ship its data, and we also assume that we can obtain as many processors as we need. If processor availability is not limited, then processes execute as soon as all their predecessors in the graph of processes have completed, and their data has been delivered. Thus, elapsed communication cost measures the time from the beginning to the end of an algorithm on a computing cloud.

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 distributed file systems 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 [HK81] 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 ([AJ87], [UY91]) 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.

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 Bulk-Synchronous Parallel Model In 1990, Valiant [Val90] introduced the BSP model, 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 [GV94], a probabilistic algorithm for sorting is developed for this model. This algorithm is based on quicksort and its 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). Recently in [Val08], Valiant proposed the multi-BSP model which extends BSP. 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. [PS84] is a central work in this area, although the first studies were based on VLSI complexity, e.g. [Tho79] — 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 [PS84].

3 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 $(x_0, 0)$, $(x_1, 1)$, $(x_2, 2), \ldots, (x_{n-1}, n-1)$ 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.

3.1 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 [Bat68] to get $O(\log^2 n)$ elapsed cost. A similar approach is to use the $O(n \log^2 n)$ version of Shell sort [She59] developed by V. Pratt ([Pra71] or see [Knu98]).

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. 1. 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.

There is a simple pattern that determines which comparator outputs are connected to which

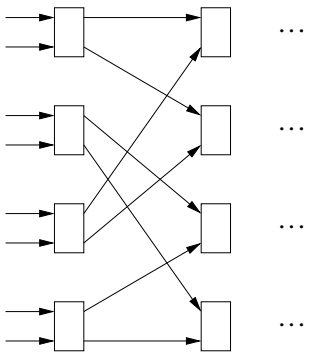

Figure 1: Part of a Batcher sorting network

comparator inputs of the next column. If we think of the outputs of the comparators as numbered in order $0, 1, \ldots, n-1$, then the connections from one column to the next are determined by writing the integer number of each comparator 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 3.1 Suppose $n = 8$ and $k = 2$. Output 5 is represented by 101 in binary. If we rotate right 2 places, the number becomes 011, or 3 in decimal. 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. □

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 3.1, 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 ith output of a process to the $h_k(i)$th process for the next column.

3.2 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 $O(s \log^2 n)$.

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 3.1 *The algorithm in Section 3.1 sorts n elements with the following costs: a) total communication $O(n \log^2 n)$, b) elapsed communication $O(s \log^2 n)$, c) total processing time $O(n \log^2 n)$ and d) elapsed processing time $O(s \log^2 n)$.* □

3.3 Sorting Under Hadoop

It turns out that because of the way Hadoop is implemented, it offers a trivial way to sort.[2] 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.

4 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 10^8 (one hundred million) elements — a number of elements that should fit in a typical main memory — then even our first algorithm can sort 10^{12} (one trillion) elements.

4.1 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 $O(p^3)$ total 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.

[2]Thanks to Ragho Murthy for making this point.

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. Let the sentinels in sorted order be $a_0, a_1, \ldots, a_{p^2-1}$.

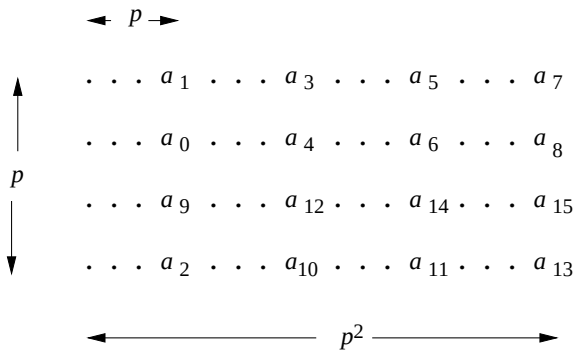

Figure 2: Sorting: lines and sentinels

EXAMPLE 4.1 Figure 2 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 distributed. For instance, we have suggested in Fig. 2 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. □

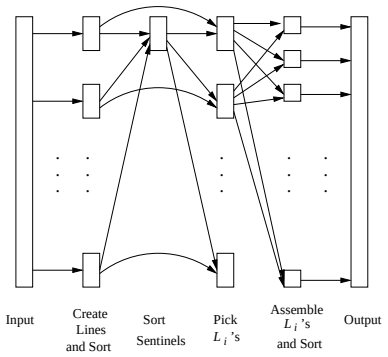

Figure 3: The processes involved in the sorting algorithm of Section 4.1

Figure 3 illustrates the network of processes tha constitute 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. 3. Their goal is to construct the p^2 sets of elements $L_0, L_1, \ldots, L_{p^2-1}$ 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. These processes are constructed by the master process and given a name that incorporates the index i if the process is to handle the elements of L_i. We need to divide the lines among the processes, and that division

is the job of Step 3. The processes of Step 3 can deduce the name of the process for L_i and can push data to each such process, as needed. Before proceeding, consider the following example.

EXAMPLE 4.2 Let us consider how Steps 3 and 4 would work if the outcome of Step 2 were as seen in Fig. 2. 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. □

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. 3, 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. 3) 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. Create p processes and push the data of the lines to them. 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. Create p^2 processes for Step 4.

Step 3: Create p processes and pull the data from Steps 1 and 2 as explained below. 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 $a_0, a_1, \ldots, a_{p^2-1}$. 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 $i = 0, \ldots, p^2 - 1$. 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 obtains its associated set of elements L_i, sorts them 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.

4.2 Communication and Processing Costs of ConsDep-Sorting

The communication and processing costs for the processes in each of the columns of Fig. 3 are easy to analyze. We can then sum them to get the total costs. All paths through the network of Fig. 3 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 4.1 *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: 1) Total communication cost: $O(n)$ 2) Elapsed communication cost: $O(s)$ 3) Total processing cost: $O(n \log s)$ 4) 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 $O(ps) = O(n)$. Thus 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 $O(s \log s)$. ∎

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.

4.3 Dealing With Large Block Size

There is one last concern that must be addressed to make the analysis of Section 4.2 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.

4.4 Comparison With Batcher Sort

The sorting algorithm of Section 4.1 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.

Morever, 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.

4.5 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 4.2. The argument is a simple generalization of the lower bound on comparisons.

Theorem 4.2 *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: First we argue that we cannot sort more than n elements where $n! = (s_1!)(s_2!)\cdots$ assuming that we have some number of processes whose inputs consist of $s_1, s_2, \ldots$ elements. Once a process reads a fixed s_i 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 $s_i!$. We 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_i elements. Thus, if we have p processes with input $s_1, s_2, \ldots, s_p$ elements and we suppose they can sort n elements, then the following must be satisfied: $n! \leq (s_1!)(s_2!)\cdots(s_p!)$.

Next, we argue that assuming a certain total communication k, and that each process gets input at most s then we cannot sort more than n elements with $n! = (s!)^{k/s}$. Since we are assuming communication k, we know that the sum $s_1 + s_2 + \cdots + s_p$ is fixed at k. We also constrain each s_i to be at most s. The product $(s_1!)(s_2!)\cdots$ consists of p 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. Thus we have number of processes $p = k/s$ each with input s; hence the following must be satisfied: $(s!)^p \geq n!$, or $p \geq (\log n!)/(\log s!)$.

If we use the asymptotic Stirling approximation $\log x! = x \log x$, then we will obtain the lower bound on communication which is $\Omega(s(n \log n)/(s \log s)) = \Omega(n(\log n)/(\log s)) = \Omega(n \log_s n)$. ∎

4.6 Extending the Constant-Depth Sort Recursively

We have so far developed a network, Fig. 3, 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 3/2th power, and implement the network for that larger s. Each of the boxes in Fig. 3 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. 3 by the network of Fig. 3 itself.

It should be evident that Steps 1 and 2 of Algorithm Cons-Dep-Sorting (columns 1 and 2 of Fig. 3) 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. Considering a communication function $C(n)$ that is smooth, the following arguments are sufficient to derive that the communication cost of s boxes, when replaced by networks, is no greater than that of $\sqrt{s}$ sorters of s elements each:

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 $s^{3/2}$.

For the particular class of communication functions which we may consider here, the above observation is stated in the following lemma whose proof can be found in the Appendix.

Lemma 4.1 *Suppose that a function $C(n)$ is such that $C(n) = an\log_s^{1+b} n$, where a and b are constants. Suppose that $i_1 + i_2 + \ldots + i_n = n^{3/2}$ and*

$$max(i_1, i_2, \ldots, i_n) \leq n$$

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

$$C(i_1) + C(i_2) + \ldots + C(i_n) \leq n^{1/2} C(n)$$

□

Also, in Step 4, each sentinel should compute its final rank by adding up all the $n^{1/2}$ partial ranks that this sentinel was attached to in Step 3. This involves adding $n^{1/2}$ counts in $n^{1/2}$ separate networks, which can be done by a network with $O(n)$ communication (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. The summing network does not affect the asymptotic upper bounds on the communication and processing costs, so we shall ignore it in our calculations.

Finally Step 3 can be implemented either by sorting or merging networks (we discuss merging networks in the next section). Note that in Step 3 each element should also decide in which network of Step 4 it is forwarded to. This is done by subtracting the rank of the element in its network in Step 3 from the rank of the element in its network in the first step; the result is the rank of the sentinel (among all the sentinels, i.e., its rank computed in Step 2) which is the first in the pair of sentinels that define the process in the fourth step.

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 4.2 *The solution to the recurrence*

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

with a basis in which $C(s)$ is linear in s for $a > 1$ and $b > 0$, is $C(n) = O(n \log_s^u n)$, where $u = \log a / \log(1+b)$. □

Moreover, Lemma 4.2 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.

4.7 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. 3

by merging networks, then we replace the constant $a = 3$ in Lemma 4.2 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. 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 ith 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 $i < j$.

For a basis, we can surely merge two sorted lists of length s each in a single process with $O(s)$ communication cost. First we describe a constant depth network which merges two lists of size s^2.

We shall explain in detail this algorithm in four steps, corresponding to the four columns in Figure 4 but we give first a succinct description of each column (note that in this case only the two middle columns of Figure 4 are implemented by merging networks while the first column only filters its input and the fourth only sums up intermediate ordinals to find the final ordinal):

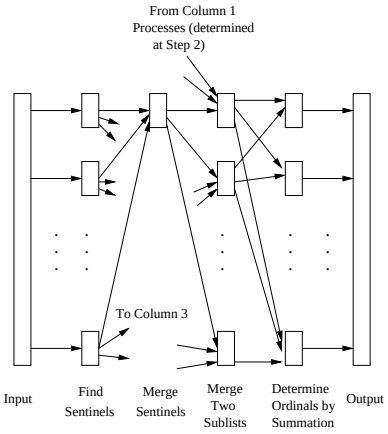

Figure 4: The processes involved in the merging algorithm of Section 4.7

- Each process in Column 1 filters the sorted list in its input to find the merge-sentinels.
- The process in Column 2 merges two sorted lists (the merge-sentinels).
- Each process in Column 3 merges two sorted lists.
- Each process in Column 4 determines (by summation) the ordinal for each element.

Supposing $O(s)$ is the limit on communication for a process, we shall build a network that merges two sorted lists of size $n = s^2$ each. In this constant-depth network, the boxes in Fig. 4 are really single processes (not networks of processes). In detail the four columns represent the following four steps.

Step 1: Suppose the input is two sorted lists, S_1 and S_2, of length $n = s^2$ each. Choose the elements of S_1 and S_2 at positions $0, s, 2s, \ldots$; 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. Each process simply identifies the merge-sentinels by their associated ordinals, which are divisible by s. Thus, this column does not require anything for the recursion, and remains a single column if we apply this algorithm recursively.

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 $0, 1, 2, \ldots$.

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.[3] These processes are the boxes in the third column. To give these processes all the input they might need, we do the following: We feed process P_{ab} with the elements of S_1 with rank between si to $s(i+1)-1$ and with the elements of S_2 with rank between sj to $s(j+1)-1$, where i and j are computed from the ranks of elements a, b: 1. If a, b are from different lists (i.e., one from S_1 and one from S_2) then i corresponds to the sentinel-rank of a in the list S_1 (supposing a belongs in S_1 and $a \leq b$) and j is the sentinel-rank of b in the list S_2 minus one. 2. If a, b are from the same list (suppose this is S_1) then P_{ab} gets from list S_1 all the elements in the interval $[a, b]$, i.e., with rank between the rank of a and the rank of b. The elements fed to P_{ab} from the list S_2 are computed from j as we explained above. This j is computed by subtracting the rank of a found in Step 2 (i.e., among all $2s$ sentinels) from the rank of a in S_1 as a merge-sentinel (i.e., the rank of a in S_1 divided by s). Note that some of the elements in process P_{ab} will be either smaller than a or greater than b; we will ignore/delete such elements in the output of each process of Step 3. Lemma 4.3 below proves that we correctly choose the data to feed each process 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 with:

Step 3: There is one process P_{ab} for each pair of consecutive sentinels a, b and each process is fed with input from Column 1 as explained above. The process merges the two sorted lists in its input (technically, it needs first to subtract the ordinal of the first element of each list from the ordinals of the subsequent elements in order for the ranks to be $0, 1, 2, \ldots$ instead of starting from an arbitrary number). Notice that not all element fed into P_{ab} need be in the interval $[a, b]$. The ones that are not, are simply deleted and the process transmits to Column 4 only the non-deleted elements (it is easy to see that the deleted elements will occur in some other process of Column 3 as non-deleted; moreover, because of this deletion, each element is transmitted only once in the fourth column). Each process P_{ab} 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 rank of a in its list and the rank of c in its list, where c is the closest to a element of the non-containing-a list (i.e., of the other list).

The following is a succinct description of the algorithm RMerge(s^2).

Algorithm RMerge(s^2)

Step 1: Create s processes where each process gets s elements. Filter the $2s$ merge-sentinels and push them to the process in Step 2.

Step 2: Merge the two lists of s merge-sentinels using one process. Create s processes for

[3] As a special case, if b is the first sentinel, then a is "minus infinity."

Step 3, one for each merge-sentinel. Create similar processes for Step 4.

Step 3: The s processes for this step are named $P_{a,b}$ for each pair (a, b) of consecutive merge-sentinels. Each process pulls data from Step 1 and Step 2. Each process receives two sorted lists of s elements each (one from S_1 and one from S_2) and merges them. The elements in the lists are chosen so that they are the only ones that may be in the interval $[a, b]$. The output of each process is also a count of the number of elements between a and b.

Step 4: Determine by summation the rank of each element.

The following is the basic lemma for the correctness of the algorithm.

Lemma 4.3 *The elements fed into the process P_{ab} are the only elements of the lists S_1 and S_2 which may be $\geq a$ and $\leq b$.* □

Proof: 1. Suppose a, b are from different lists. Suppose a is from S_1 and b from S_2. Then all the elements of S_2 greater than b are outside the interval $[a, b]$. Suppose c is the first merge-sentinel of S_2 which is less than b. Notice that $c < a$ because a is the merge-sentinel next to b in the list of merged sentinels. Hence all the elements of S_2 less than c are also less than a.

2. Suppose a, b are both from list S_2. Clearly we correctly chose the elements that lie between a and b in the list S_2. For the elements chosen from the list S_1 we argue as follows: Let a', b' be merge-sentinels of list S_1 such that the following holds: a' is the first merge-sentinel of list S_1 less than a and b' is the first merge-sentinel of list S_1 greater than b. Then $a' < a$ and $b' > b$. Hence all elements of S_1 that either less than a' or greater than b' are excluded from being in the interval $[a, b]$. The rank of a' in S_1 is computed by subtracting the rank of a found in Step 2 (i.e., among all $2s$ sentinels) from the rank of a in S_1 as a merge-sentinel (i.e., the rank of a in S_1 divided by s). This computation is correct because the rank of a in the sentinel list says that there are so many sentinels preceding a in the total of sentinels (of both lists). The rank of a as a sentinel in S_2 says that there are so many S_2-sentinels preceding a. Thus the subtraction leaves the number of S_1-sentinels preceding a, hence it denotes the rank (as a sentinel) of a' (the first S_1-sentinel less than a) in S_1. ∎

The following theorem states the costs of the constant depth algorithm. The proof is along the lines of the proof of Theorem 4.1.

Theorem 4.3 *Algorithm* RMERGE(s^2) *constructs a network of processes of depth four which has the following costs: 1) Total communication cost: $O(n)$ 2) Elapsed communication cost: $O(s)$ 3) Total processing cost: $O(sn)$ 4) Elapsed processing cost: $O(s)$* □

The recursive Merging Network 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 $0, \sqrt{n}, 2\sqrt{n}, \ldots$. 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 computing the ordinal of the first element of each network $N'_{a,b}$ (which replaces the process $P'_{a,b}$ in the recursive merge), where a, b are consecutive sentinels in the merged list of sentinels. It further involves adding this ordinal to the partial rank of each element of $N'_{a,b}$ (i.e., partial rank in the sense that it is the rank of the element only among the elements that lie in the interval $[a, b]$). Thus, Steps 1 and 4 together require $O(n)$ communication. The following theorem states the costs for this recursive algorithm.

Theorem 4.4 *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: 1) Total communication cost: $O(n \log_2 \log_s n)$ 2) Elapsed communication cost: $O(s \log_s n \log_2 \log_s n)$ 3) Total processing cost: $O(n \log_2 \log_s n)$ 4) Elapsed processing cost: $O(s \log_s n \log_2 \log_s n)$.* □

Proof: Correctness is proved in a similar way as in Theorem 4.1.

1) 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 4.2 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.

2) The elapsed communication is given by

$$C_e(n) = 2C_e(\sqrt{n}) + O(1), C_e(s) = O(s)$$

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

3) 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.

4) Similarly, the elapsed processing cost is the same as the elapsed communication. ∎

4.8 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 ith stage we merge two lists of length $s2^{i-1}$. The resulting algorithm (which we shall discuss in detail in the rest of this subsection) has costs that are each $\log_2(n/s)$ times the costs given in Theorem 4.4.

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 4.7 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.

The algorithm MMERGE$(S_1, S_2, \ldots, S_m)$ merges a number of sorted lists $S_1, S_2, \ldots, S_m$ using algorithm RMERGE. In the first stage of building the network we use networks $N_1, N_2, \ldots, N_{m/2}$[4]

[4]If m is odd then the last network is not actually a network but the list S_m itself.

(of appropriate size) where network N_i is a network built from RMERGE to merge the two lists S_{2i-1} and S_{2i}. The output of the networks built at this stage is a number of lists $S'_1, S'_2, \ldots, S'_{m/2}$, where list $S'_{(i+1)/2}$ is the merged result of lists S_i and S_{i+1}. Thus in the second stage we need to merge the $m/2$ lists $S'_1, S'_2, \ldots, S'_{m/2}$. We repeat the same procedure as in the first stage, i.e., we build networks $N'_1, N'_2, \ldots, N'_{m/4}$, where network N'_i is a network built from RMERGE to merge the two lists S'_{2i-1} and S'_{2i}. Thus we will be done after $\log_2 m$ stages.

The following theorem states the costs of a network built by the algorithm MMERGE$(S_1, S_2, \ldots, S_m)$. The costs are measured as functions of $\Sigma|S_i|$.

Theorem 4.5 *We use algorithm* MMERGE$(S_1, S_2, \ldots, S_m)$ *to construct a network of processes. Suppose* $\Sigma|S_i| = n$ *and* $|S_i| = s$ *where* s *is the upper bound of communication per process. The costs of the network are as follows: 1) total communication cost:* $O(n \log_2(n/s) \log_2 \log_s n)$ *2) elapsed communication cost:* $O(s \log_2(n/s) \log_2 \log_s n)$ *3) total processing cost:* $O(n \log_2(n/s) \log_2 \log_s n)$ *4) elapsed processing cost:* $O(s \log_2(n/s) \log_2 \log_s n)$. □

Proof: First, we observe that we have $O(\log_2(n/s))$ stages. We compute the elapsed costs. In the worst case there is a longest path in the network in which in stage i the elapsed communication cost is equal to the communication cost of a network built by RMERGE$(2^i s)$. Thus by summing up we have the elapsed communication cost to be $O(s \log_2(n/s) \log_2 \log_s n)$ (actually, since we have log log function, we can assume that the elapsed communication for each stage is $O(s \log_2 \log_s n)$; thus all we have to do to sum up is multiply by the number of stages). The elapsed processing cost is computed exactly in the same way.

In order to compute the total communication cost we compute the total communication cost of each stage. For a certain stage i, we have that each of the RMERGE networks in this stage has input $j_i = 2^i s$ and also the summation over all inputs in this stage is equal to n. Hence, the total communication in stage i is

$$\Sigma j_i \log_2 \log_s j_i \leq n \log_2 \log_s n$$

(because $(\log_2 \log_s j_i)/(\log_2 \log_s n) < 1$, thus, if we divide both sides by $\log_2 \log_s n$ then the left hand side becomes smaller than Σi_j and hence smaller than n). So the total communication per stage is $O(n \log_2 \log_s n)$. Since we have $O(\log_2(n/s))$ stages, the total communication cost is $O(n \log_2(n/s) \log_2 \log_s n)$. The total processing cost is computed exactly in the same way. ∎

An asymptotically better algorithm Asymptotically we can do better than Theorem 4.7 if we use in Step 4 of the recursive sort a network built by algorithm MMERGE which is described above (instead of sorting networks). In order to compute the costs we need to extend Theorem 4.6 for the case when each $|S_i| \geq s$. This can be done using techniques similar to the techniques in Lemma A.1. It will give the following recurrence for the total communication:

$$C(n^{3/2}) = (\sqrt{n})C(n) + O(n \log_2(n/s) \log_2 \log_s n) +$$

$$\textit{lower order terms}$$

The solution is: $O(n \log_2(n/s) \log_2^2 \log_s n)$ which is asymptotically better that the one in Theorem 4.7 but, as we explained in Subsection 4.8, for realistic values of n and s, the communication in Theorem 4.7 may be preferable.

The following theorem states the costs for this asymptotically better sorting algorithm:

Theorem 4.6 *We construct a network for sorting n elements using the sorting algorithm of Figure 3, where in the third column we use the merging network* RMERGE *and in the fourth column we use the network* RMERGE.

This network has the following costs: 1) total communication cost: $O(n \log_2(n/s) \log_2^2 \log_s n)$, 2) elapsed communication cost: $O(s \log_s^2 n \log_2 \log_s n)$, 3) total processing cost: $O(n \log_2(n/s) \log_2^2 \log_s n)$ and 4) elapsed processing cost: $O(s \log_s^2 n \log_2 \log_s n)$. □

4.9 Completing the Recursive Sort

We can now complete the plan outlined in Section 4.6. ALGORITHM RECURS-SORTING

1. For any n, we can build a network as in Fig. 3 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

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

from which we conclude $C(n) = O(n \log_s^{1.7} n)$ by Lemma 4.2. The cost measures are summarized in the following theorem.

Theorem 4.7 *Algorithm* RECURS-SORTING *constructs a network of processes which correctly sorts n elements. Moreover, the costs of the network are: 1) Total communication cost: $O(n \log_s^{1.7} n)$ 2) Elapsed communication cost: $O(s \log_s^{2.7} n)$ 3) Total processing cost: $O(n \log_s^{1.7} n \log_2 s)$ 4) 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 4.1.

1) The argument is as discussed above.

2) 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 $C_e(s) = O(s)$, gives us the solution.

3) 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.

4) The same argument as for (3) applies to the elapsed processing cost. ∎

5 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 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 cluster 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. 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"?

3. One of the major objections of [DS08] 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.

References

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. *SIGMOD Rec.*, 26(2):243–254, 1997.

[AJ87] Rakesh 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.

[Apa06] Apache. Hadoop. http://hadoop.apache.org/, 2006.

[AU10] F. N. Afrati and J. D. Ullman. Optimizing joins in a mapreduce environment. In *EDBT*, 2010.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[BCLC99] Philip Buonadonna, Joshua Coates, Spencer Low, and David 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.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine, 1998.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[DeW09] D. DeWitt. Private communication, 2009.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DPR+08] David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey F. Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[DS08] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html, 2008.

[GGL03] Sanjay Ghemawat, Howard Gobioff, , and Shun-Tak Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.

[HK81] Jia-Wei Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981.

[Knu98] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A risc machine sort. In *SIGMOD Conference*, pages 233–242, 1994.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[Pra71] V Pratt. Shellsort and sorting networks. Ph.D. thesis, Stanford University, 1971.

[PS84] Christos H. Papadimitriou and Michael Sipser. Communication complexity. *J. Comput. Syst. Sci.*, 28(2):260–269, 1984.

[RSRK07] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD Conference*, pages 365–376, 2007.

[She59] Donald L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.

[Ter09] Sort benchmark home page. http://sortbenchmark.org/, 2009.

[Tho79] 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.

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. The input/output complexity of transitive closure. *Ann. Math. Artif. Intell.*, 3(2-4):331–360, 1991.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[Val08] Leslie G. Valiant. A bridging model for multi-core computing. In *ESA*, pages 13–28, 2008.

A Proof of Lemma 4.1

We prove here Lemma 4.1 which is restated as Lemma A.1 below.

Lemma A.1 *Suppose that a function $C(n)$ is such that $C(n) = anlog_s^{1+b}n$, where a and b are constants. Suppose that $i_1 + i_2 + \ldots + i_n = n^{3/2}$ and*

$$max(i_1, i_2, \ldots, i_n) \leq n$$

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

$$C(i_1) + C(i_2) + \ldots + C(i_n) \leq n^{1/2}C(n)$$

□

Proof: The proof essentially emanates from the following claim:

Claim 1: If $i_1 + i_2 + \ldots + i_n = n^{3/2}$ and $max(i_1, i_2, \ldots, i_n) \leq n$ where i_j are all positive integers then

$$i_1 \log_s^{1+b} i_1 + i_2 \log_s^{1+b} i_2 + \ldots + i_n \log_s^{1+b} i_n \leq$$

$$(2/3)^{1+b} n^{3/2} \log_s^{1+b} n^{3/2}$$

Proof of Claim 1.

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

$$i_1 \log_s^{1+b} i_1 + i_2 \log_s^{1+b} i_2 + \cdots + i_n \log_s^{1+b} i_n \leq$$

$$\Sigma i_j \log_s i_j \log_s^b n \leq 2/3 n^{3/2} \log_s n^{3/2} \log_s^b n =$$

$$(2/3)^{1+b} n^{3/2} \log_s^{1+b} n^{3/2}$$

■

Lemma A.2 *If* $i_1 + i_2 + \ldots + i_n = n^{3/2}$ *and*

$$max(i_1, i_2, \ldots, i_n) \leq n$$

where i_j *are all positive integers then*

$$i_1 \log_s i_1 + i_2 \log_s i_2 + \ldots + i_n \log_s i_n < n^{3/2} \log_s n$$

□

Proof:

The proof is a straightforward consequence of Claim 2:

Claim 2: Under the assumption that $1/n^{3/2} \leq p_i \leq 1/n^{1/2}$ and $\Sigma p_1 = 1$, the following holds:

$$p_1 \log_s p_1 + p_2 \log_s p_2 + \ldots + p_n \log_s p_n < -1/2 \log_s n \quad (1)$$

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 $n_1 + n_2 + \ldots + n_n = n^{3/2}$, we get immediately the desired inequality.

Proof of Claim 2: We know from Information Theory that if we tend to "unbalance" the p_is (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_is either to $1/n^{1/2}$ or to $1/n^{3/2}$. Suppose we change x p_is to $1/n^{1/2}$ and y p_is to $1/n^{3/2}$. We want to hold:

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

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

Supposing x and y are integers, we have:

$$p_1 \log_s p_1 + p_2 \log_s p_2 + \ldots + p_n \log_s p_n$$

$$< [(n^{1/2} - 1)/(1 - 1/n)]/n^{1/2} \log_s (1/n^{1/2})$$

$$+[(n - n^{1/2})/(1 - 1/n)]/n^{3/2} \log_s (1/n^{3/2}) =$$

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

If x and y are not integers then we want to hold:

$x+y=n-1$ and $x/n^{1/2}+y/n^{3/2}+w=1$.

The w denotes the one value that remains and is not equal to either of the bounds. There is only one w, because, if there are more, then we can replace the one value to be equal to one of the bounds and replace another by subtracting the difference. Then if we solve again, we get a similar analysis as the one above by replacing the w to one of the bounds to obtain the upper bound we state in the claim.

This concludes the proof of Claim 2. ∎