Fast Sorting on a Distributed-Memory Architecture

David R. Cheng[1], Viral Shah[2], John R. Gilbert[2], Alan Edelman[1]
[1]Massachusetts Institute of Technology, [2]University of California, Santa Barbara

***Abstract*— We consider the often-studied problem of sorting, for a parallel computer. Given an input array distributed evenly over p processors, the task is to compute the sorted output array, also distributed over the p processors. Many existing algorithms take the approach of approximately load-balancing the output, leaving each processor with $\Theta(\frac{n}{p})$ elements. However, in many cases, approximate load-balancing leads to inefficiencies in both the sorting itself and in further uses of the data after sorting. We provide a deterministic parallel sorting algorithm that uses parallel selection to produce any output distribution exactly, particularly one that is perfectly load-balanced. Furthermore, when using a comparison sort, this algorithm is 1-optimal in both computation and communication. We provide an empirical study that illustrates the efficiency of exact data splitting, and shows an improvement over two sample sort algorithms.**

***Index Terms*— Parallel sorting, distributed-memory algorithms, High-Performance Computing.**

I. Introduction

Parallel sorting has been widely studied in the last couple of decades for a variety of computer architectures. Many high performance computers today have distributed memory, and commodity clusters are becoming increasingly common. The cost of communication is significantly larger than the cost of computation on a distributed memory computer. We propose a sorting algorithm that is close to optimal in both the computation and communication required. The motivation for the development of our sorting algorithm was to implement sparse matrix functionality in Matlab*P [13].

Blelloch et al. [1] compare several parallel sorting algorithms on the CM–2, and report that a sampling based sort and radix sort are good algorithms to use in practice. We first tried a sampling based sort, but this had a couple of problems. A sampling sort requires a redistribution phase at the end, so that the output has the desired distribution. The sampling process itself requires "well chosen" parameters to yield "good" samples. We noticed that we can do away with both these steps if we can determine exact splitters quickly. Saukas and Song [12] describe a quick parallel selection algorithm. Our algorithm extends this work to efficiently find $p-1$ exact splitters in $O(p \log n)$ rounds of communication, providing a 1-optimal parallel sorting algorithm.

II. Algorithm Description

Before giving the concise operation of the sorting algorithm, we begin with some assumptions and notation.

We are given p processors to sort n total elements in a vector v. Assume that the input elements are already load balanced, or *evenly distributed* over the p processors.[1] Note that the values may be arbitrary. In particular, we rank the processors $1 \dots p$, and define v_i to be the elements held locally by processor i. The *distribution* of v is a vector d where $d_i = |v_i|$. Then we say v is evenly distributed if it is formed by the concatenation $v = v_1 \dots v_p$, and $d_i \leq \lceil \frac{n}{p} \rceil$ for any i.

In the algorithm description below, we assume the task is to sort the input in increasing order. Naturally, the choice is arbitrary and any other comparison function may be used.

```
Algorithm.
Input: A vector $v$ of $n$ total elements,
evenly distributed among $p$ processors.
Output: An evenly distributed vector $w$
with the same distribution as $v$,
containing the sorted elements of $v$.
  1) Sort the local elements $v_i$ into a
     vector $v'_i$.
  2) Determine the exact splitting of
     the local data:
       a) Compute the partial sums $r_0 = 0$
          and $r_j = \sum_{k=1}^{j} d_k$ for $j = 1 \dots p$.
       b) Use a parallel select
          algorithm to find the elements
          $e_1, \dots, e_{p-1}$ of global rank
          $r_1, \dots, r_{p-1}$, respectively.
       c) For each $r_j$, have processor $i$
          compute the local index $s_{ij}$ so
          that $r_j = \sum_{i=1}^{p} s_{ij}$ and the first
          $s_{ij}$ elements of $v'_i$ are no larger
          than $e_j$.
  3) Reroute the sorted elements in $v'_i$
     according to the indices $s_{ij}$:
     processor $i$ sends elements in the
     range $s_{ij-1} \dots s_{ij}$ to processor $j$.
  4) Locally merge the $p$ sorted
     sub-vectors into the output $w_i$.
```

The details of each step now follow.

A. Local Sort

The first step may invoke any local sort applicable to the problem at hand. It is beyond the scope of this study to devise an efficient sequential sorting algorithm, as the problem is very well studied. We simply impose the restriction that the algorithm used here should be identical to the one used for a baseline comparison on a non-parallel computer. Define the computation cost for this algorithm on an input of size n to be $T_s(n)$. Therefore, the amount of computation done by processor i is just $T_s(d_i)$. Because the local sorting must

[1] If this assumption does not hold, an initial redistribution step can be added.

be completed on each processor before the next step can proceed, the global cost is $\max_i T_s(d_i) = T_s(\lceil \frac{n}{p} \rceil)$. With a radix sort, this becomes $O(n/p)$; with a comparison-based sort, $O(\frac{n}{p} \lg \frac{n}{p})$.

B. Exact Splitting

This step is nontrivial, and the main result of this paper follows from the observation that exact splitting over locally sorted data can be done efficiently.

The method used for simultaneous selection was given by Saukas and Song in [12], with two main differences: local ranking is done by binary search rather than partition, and we perform $O(\lg n)$ rounds of communication rather than $O(\lg cp)$ for some constant c. For completeness, a description of the selection algorithm is given below.

1) Single Selection: First, we consider the simpler problem of selecting just one target, an element of global rank r.[2] The algorithm for this task is motivated by the sequential methods for the same problem, most notably the one given in [2].

Although it may be clearer to define the selection algorithm recursively, the practical implementation and extension into simultaneous selection proceed more naturally from an iterative description. Define an active range to be the contiguous sequence of elements in v'_i that may still have rank r, and let a_i represent its size. Note that the total number of active elements is $\sum_{i=1}^{p} a_i$. Initially, the active range on each processor is the entire vector v'_i and a_i is just the input distribution d_i. In each iteration of the algorithm, a "pivot" is found that partitions the active range in two. Either the pivot is determined to be the target element, or the next iteration continues on one of the partitions.

Each processor i performs the following steps:

1) Index the median m_i of the active range of v'_i, and broadcast the value.
2) Weight median m_i by $\frac{a_i}{\sum_{k=1}^{p} a_k}$. Find the weighted median of medians m_m. By definition, the weights of the $\{m_i | m_i < m_m\}$ sum to at most $\frac{1}{2}$, as do the weights of the $\{m_i | m_i > m_m\}$.
3) Binary search m_m over the active range of v'_i to determine the first and last positions f_i and l_i it can be inserted into the sorted vector v'_i. Broadcast these two values.
4) Compute $f = \sum_{i=1}^{p} f_i$ and $l = \sum_{i=1}^{p} l_i$. The element m_m has ranks $[f, l]$ in v.
5) If $r \in [f, l]$, then m_m is the target element and we exit. Otherwise the active range is truncated as follows:
   Increase the bottom index to $l_i + 1$ if $l < r$; or decrease the top index to $f_i - 1$ if $r < f$.
   Loop on the truncated active range.

We can think of the weighted median of medians as a pivot, because it is used to split the input for the next iteration. It is a well-known result that the weighted median of medians can be computed in linear time [11], [4]. One possible way is to partition the values with the (unweighted) median, accumulate the weights on each side of the median, and recurse on the side that has too much weight. Therefore, the amount of computation in each round is $O(p) + O(\lg a_i) + O(1) = O(p + \lg \frac{n}{p})$ per processor.

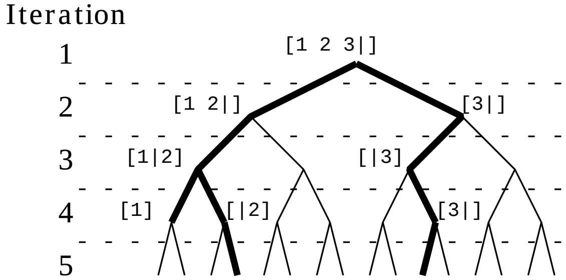

Fig. 1. Example execution of selecting three elements. Each node corresponds to a contiguous range of v'_i, and gets split into its two children by the pivot. The root is the entire v'_i, and the bold traces which ranges are active at each iteration. The array at a node represents the target ranks that may be found by the search path, and the vertical bar in the array indicates the relative position of the pivot's rank.

Furthermore, as shown in [12], splitting the data by the weighted median of medians will decrease the total number of active elements by at least a factor of $\frac{1}{4}$. Because the step begins with n elements under consideration, there are $O(\lg n)$ iterations. The total single-processor computation for selection is then $O(p \lg n + \lg \frac{n}{p} \lg n) = O(p \lg n + \lg^2 n)$.

The amount of communication is straightforward to compute: two broadcasts per iteration, for $O(p \lg n)$ total bytes being transferred over $O(\lg n)$ rounds.

2) Simultaneous Selection: The problem is now to select multiple targets, each with a different global rank. In the context of the sorting problem, we want the $p - 1$ elements of global rank $d_1, d_1 + d_2, \ldots, \sum_{i=1}^{p-1} d_i$. One simple way to do this would call the single selection problem for each desired rank. Unfortunately, doing so would increase the number communication rounds by a factor of $O(p)$. We can avoid this inflation by solving multiple selection problems independently, but combining their communication. Stated another way, instead of finding $p - 1$ paths one after another from root to leaf of the binary search tree, we take a breadth-first search with breadth at most $p - 1$ (see Figure 1).

To implement simultaneous selection, we augment the single selection algorithm with a set A of active ranges. Each of these active ranges will produce at least one target. An iteration of the algorithm proceeds as in single selection, but finds multiple pivots: a weighted median of medians for each active range. If an active range produces a pivot that is one of the target elements, we eliminate that active range from A (as in the leftmost branch of Figure 1). Otherwise, we examine each of the two partitions induced by the pivot, and add it to A if it may yield a target. Note that as in iterations 1 and 3 in Figure 1, it is possible for both partitions to be added.

In slightly more detail, we handle the augmentation by

[2] To handle the case of non-unique input elements, any element may actually have a range of global ranks. To be more precise, we want to find the element whose set of ranks contains r.

looping over A in each step. The local medians are bundled together for a single broadcast at the end of Step 1, as are the local ranks in Step 3. For Step 5, we use the fact that each active range in A has a corresponding set of the target ranks: those targets that lie between the bottom and top indices of the active range. If we keep the subset of target ranks sorted, a binary search over it with the pivot rank[3] will split the target set as well. The left target subset is associated with the left partition of the active range, and the right sides follow similarly. The left or right partition of the active range gets added to A for the next iteration only if the corresponding target subset is non-empty.

The computation time necessary for simultaneous selection follows by inflating each step of the single selection by a factor of p (because $|A| \leq p$). The exception is the last step, where we also need to binary search over $O(p)$ targets. This amount to $O(p+p^2+p\lg\frac{n}{p}+p+p\lg p) = O(p^2+p\lg\frac{n}{p})$ per iteration. Again, there are $O(\lg n)$ iterations for total computation time of $O(p^2\lg n + p\lg^2 n)$.

This step runs in $O(p)$ space, the scratch area needed to hold received data and pass state between iterations.

The communication time is similarly inflated: two broadcasts per round, each having one processor send $O(p)$ data to all the others. The aggregate amount of data being sent is $O(p^2\lg n)$ over $O(\lg n)$ rounds.

3) Producing Indices: Each processor computes a local matrix S of size $p \times (p+1)$. Recall that S splits the local data v'_i into p segments, with $s_{k0} = 0$ and $s_{kp} = d_k$ for $k = 1 \dots p$. The remaining $p-1$ columns come as output of the selection. For simplicity of notation, we briefly describe the output procedure in the context of single selection; it extends naturally for simultaneous selection. When we find that a particular m_m has global ranks $[f, l) \ni r_k$, we also have the local ranks f_i and l_i. There are $r_k - f$ excess elements with value m_m that should be routed to processor k. For stability, we assign s_{ki} from $i = 1$ to p, taking as many elements as possible without overstepping the excess. More precisely,

$$s_{ki} = \min\left\{ f_i + (r_k - f) - \sum_{j=1}^{i-1}(s_{kj} - f_j),\ l_i \right\}$$

The computation requirements for this step are $O(p^2)$ to populate the matrix S; the space used is also $O(p^2)$.

C. Element Rerouting

This step is purely one round of communication. There exist inputs such that the input and output locations of each element are different. Therefore, the aggregate amount of data being communicated in this step is $O(n)$. However, note that this cost can not be avoided. An optimal parallel sorting algorithm must communicate at least the same amount of data as done in this step, simply because an element must at least move from its input location to its output location.

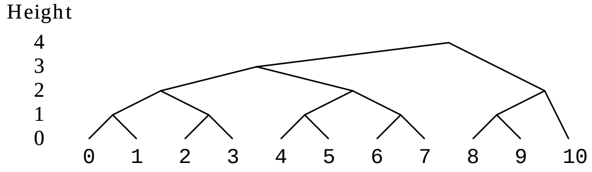

Fig. 2. Sequence of merges for p not a power of 2.

The space requirement is an additional $O(d_i)$, because we do not have the tools for in-place sending and receiving of data.

D. Merging

Now each processor has p sorted sub-vectors, and we want to merge them into a single sorted sequence. The simple approach we take for this problem is to conceptually build a binary tree on top of the vectors. To handle the case of p that are not powers of 2, we say a node of height i has at most 2^i leaf descendants, whose ranks are in $[k \cdot 2^i, (k+1) \cdot 2^i)$ for some k (Figure 2). It is clear that the tree has height $\leq \lceil \lg p \rceil$.

From this tree, we merge pairs of sub-vectors out-of-place, for cache efficiency. This can be done by alternating between the temporary space necessary for the rerouting step, and the eventual output space.

Notice that a merge will move a particular element exactly once (from one buffer to its sorted position in the other buffer). Furthermore, there is at most one comparison for each element move. Finally, every time an element gets moved, it goes into a sorted sub-vector at a higher level in the tree. Therefore each element moves at most $\lceil \lg p \rceil$ times, for a total computation time of $d_i \lceil \lg p \rceil$. Again, we take the time of the slowest processor, for computation time of $\lceil \frac{n}{p} \rceil \lceil \lg p \rceil$.

E. Theoretical Performance

We want to compare this algorithm against an arbitrary parallel sorting algorithm with the following properties:

1) Total computation time $T^*_s(n,p) = \frac{1}{p} T_s(n)$ for $1 \leq p \leq P$, linear speedup in p over any sequential sorting algorithm with running time $T_s(n)$.
2) Minimal amount of cross-processor communication $T^*_c(v)$, the number of elements that begin and end on different processors.

We will not go on to claim that such an algorithm is truly an optimal parallel algorithm, because we do not require $T_s(n)$ to be optimal. However, optimality of $T_s(n)$ does imply optimality of $T^*_s(n,p)$ for $p \leq P$. Briefly, if there were a faster $T'_s(n,p)$ for some p, then we could simulate it on a single processor for total time $pT'_s(n,p) < pT^*_s(n,p) = T_s(n)$, which is a contradiction.

1) Computation: We can examine the total computation time by adding together the time for each step, and comparing

[3] Actually, we binary search for the first position f may be inserted, and for the last position l may be inserted. If the two positions are not the same, we have found at least one target.

against the theoretical $T_s^*(n,p)$:

$$\begin{aligned} & T_s(\lceil \tfrac{n}{p} \rceil) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq\ & \frac{1}{p} T_s(n+p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ =\ & T_s^*(n+p,p) + O(p^2 \lg n + p \lg^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \end{aligned}$$

The inequality follows from the fact that $T_s^*(n) = \Omega(n)$.

It is interesting to note the case where a comparison sort is necessary. Then we use a sequential sort with $T_s(n) \leq c\lceil \frac{n}{p} \rceil \lg \lceil \frac{n}{p} \rceil$ for some $c \geq 1$. We can then combine this cost with the time required for merging (Step 4):

$$\begin{aligned} & c\lceil \tfrac{n}{p} \rceil \lg \lceil \tfrac{n}{p} \rceil + \lceil \tfrac{n}{p} \rceil \lceil \lg p \rceil \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \lg(n+p) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \lg n + c\lceil \tfrac{n}{p} \rceil \lg(1 + \tfrac{p}{n}) + \lceil \tfrac{n}{p} \rceil (\lceil \lg p \rceil - c \lg p) \\ \leq\ & \frac{cn \lg n}{p} + \lg n + 2c + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of } 2) \end{aligned}$$

With comparison sorting, the total computation time becomes:

$$T_s^*(n,p) + O(p^2 \lg n + p \lg^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of } 2) \qquad (1)$$

Furthermore, $T_s^*(n,p)$ is optimal to within the constant factor c.

2) Communication: We have already established that the exact splitting algorithm will provide the final locations of the elements. The amount of communication done in the rerouting phase is then the optimal amount. Therefore, total cost is:

$$T_c^*(v) \text{ in 1 round} + O(p^2 \lg n) \text{ in } \lg n \text{ rounds}$$

3) Space: The total space usage aside from the input is:

$$O\left(p^2 + \frac{n}{p}\right)$$

4) Requirements: Given these bounds, it is clear that this algorithm is only practical for $p^2 \leq \frac{n}{p} \Rightarrow p^3 \leq n$. Returning to the formulation given in Section II-E, we have $p = \lfloor n^{1/3} \rfloor$. This requirement is a common property of other parallel sorting algorithms, particularly sample sort (i.e. [1], [14], [9], as noted in [8]).

5) Analysis in the BSP Model: A bulk-synchronous parallel computer, described in [15], models a system with three parameters: p, the number of processors; L, the minimum amount of time between subsequent rounds of communication; and g, a measure of bandwidth in time per message size. Following the naming conventions of [7], define π to be the ratio of computation cost of the BSP algorithm to the computation cost of a sequential algorithm. Similarly, define μ to be the ratio of communication cost of the BSP algorithm to the number of memory movements in a sequential algorithm. We say an algorithm is c-optimal in computation if $\pi = c + o(1)$ as $n \to \infty$, and similarly for μ and communication.

We may naturally compute the ratio π to be Equation 1 over $T_s^*(n,p) = \frac{cn \lg n}{p}$. Thus,

$$\pi = 1 + \frac{p^3}{cn} + \frac{p^2 \lg n}{cn} + \frac{1}{c \lg n} = 1 + o(1) \text{ as } n \to \infty$$

Furthermore, there exist movement-optimal sorting algorithms (i.e. [6]), so we compute μ against $\frac{gn}{p}$. It is straightforward to verify that the BSP cost of exact splitting is $O(\lg n \max\{L, gp^2 \lg n\})$, giving us

$$\mu = 1 + \frac{pL \lg n}{gn} + \frac{p^3 \lg^2 n}{n} = 1 + o(1) \text{ as } n \to \infty$$

Therefore the algorithm is 1-optimal in both computation and communication.

Exact splitting dominates the cost beyond the local sort and the rerouting steps. The total running time is therefore $O(\frac{n \lg n}{p} + \frac{gn}{p} + \lg n \max\{L, gp^2 \lg n\})$. This bound is an improvement on that given by [8], for small L and $p^2 \lg^2 n$. The tradeoff is that we have decreased one round of communicating much data, to use many rounds of communicating little data. Our experimental results indicate that this choice is reasonable.

III. Results

The communication costs are also near optimal if we assume that p is small, and there is little overhead for a round of communication. Furthermore, the sequential computation speedup is near linear if $p \ll n$, and we need comparison-based sorting. Notice that the speedup is given with respect to a sequential algorithm, rather than to itself with small p. The intention is that efficient sequential sorting algorithms and implementations can be developed without any consideration for parallelization, and then be simply dropped in for good parallel performance.

We now turn to empirical results, which suggest that the exact splitting uses little computation and communication time.

A. Experimental Setup

We implemented the algorithm using MPI with C++. The motivation is for the code to be used as a library with a simple interface; it is therefore templated, and comparison based. As a sequential sort, it calls `stable_sort` provided by STL. For the element rerouting, it calls `MPI_Alltoallv` provided by LAM 6.5.9. We shall treat these two operations as primitives, as it is beyond the scope of this study to propose efficient algorithms for either of them.

We test the algorithm on a distributed-memory system built from commodity hardware, with 16 nodes. Each node contains a Xeon 2.40 GHz processor and 2 Gb of memory. Each node is connected through a Gigabit Ethernet switch, so the network distance between any two nodes is exactly two hops.

In this section, all the timings are based on the average of eight trials, and are reported in seconds.

B. Sorting Uniform Integers

For starters, we can examine in detail how this sorting algorithm performs when the input values are uniformly distributed random (32-bit) integers. After we get a sense of what steps tend to be more time-consuming, we look at other inputs and see how they affect the times.

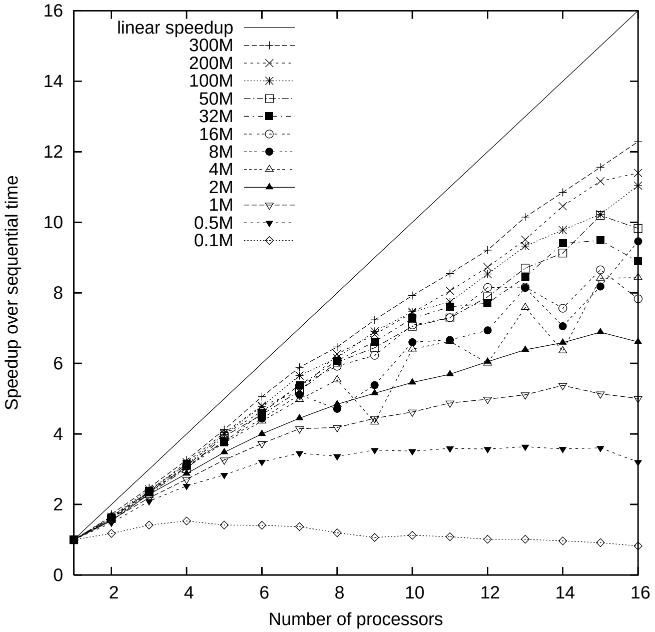

Fig. 3. Total speedup on various-sized inputs

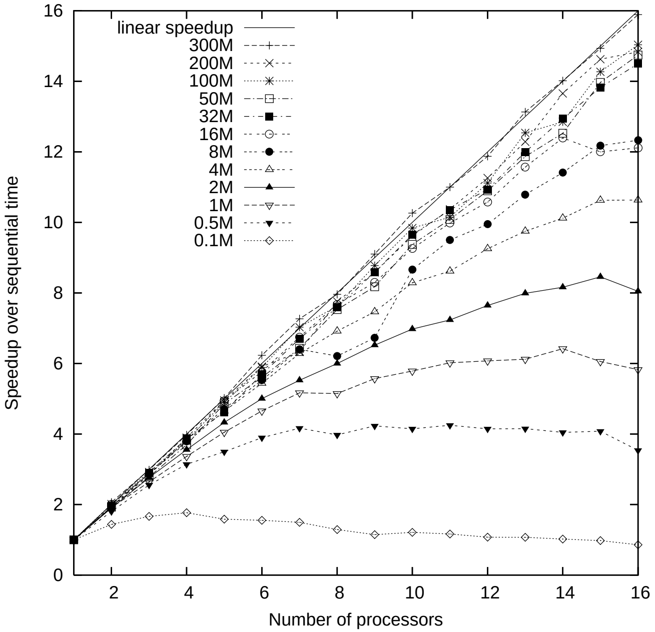

Fig. 4. Speedup when leaving out time for element rerouting

TABLE I

Absolute times for sorting uniformly distributed integers

0.5 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	0.097823	0.097823	0	0	0
2	0.066122	0.047728	0.004427	0.011933	0.002034
4	0.038737	0.022619	0.005716	0.007581	0.002821
8	0.029049	0.010973	0.012117	0.004428	0.001531
16	0.030562	0.005037	0.021535	0.002908	0.001082

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	7.755669	7.755669	0	0	0
2	4.834520	3.858595	0.005930	0.842257	0.127738
4	2.470569	1.825163	0.008167	0.467573	0.169665
8	1.275011	0.907056	0.016055	0.253702	0.098198
16	0.871702	0.429924	0.028826	0.336901	0.076051

300 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
1	84.331021	84.331021	0	0	0
2	48.908290	39.453687	0.006847	8.072060	1.375696
4	25.875986	19.532786	0.008859	4.658342	1.675998
8	13.040635	9.648278	0.017789	2.447276	0.927293
16	6.863963	4.580638	0.032176	1.557003	0.694146

Figure 3 displays the speedup of the algorithm as a function of the number of processors p, over a wide range of input sizes. Not surprisingly, the smallest problems already run very efficiently on a single processor, and therefore do not benefit much from parallelization. However, even the biggest problems display a large gap between their empirical speedup and the optimal linear speedup.

Table I provides the breakdown of the total sorting time into the component steps. As explained in Section II-C, the cost of the communication-intensive element rerouting can not be avoided. Therefore, we may examine how this algorithm performs when excluding the rerouting time; the speedups in this artificial setting are given by Figure 4. The results suggest that the algorithm is near-optimal for large input sizes (of at least 32 million), and “practically optimal” for very large inputs.

We can further infer from Table I that the time for exact splitting is small. The dominating factor in the splitting time is a linear dependence on the number of processors; because little data moves in each round of communication, the constant overhead of sending a message dominates the growth in message size. Therefore, despite the theoretical performance bound of $O(p^2 \lg^2 n)$, the empirical performance suggests the more favorable $O(p \lg n)$. This latter bound comes from $O(\lg n)$ rounds, each taking $O(p)$ time. The computation time in this step is entirely dominated by the communication. Figure 5 provides further support of the empirical bound: modulo some outliers, the scaled splitting times do not show a clear tendency to increase as we move to the right along the x-axis (more processors and bigger problems).

C. Sorting Contrived Inputs

As a sanity check, we experiment with sorting an input consisting of all zeros. Table II gives the empirical results. The exact splitting step will use one round of communication and exit, when it realizes 0 contains all the target ranks it needs. Therefore, the splitter selection should have no noticeable dependence on n (the time for a single binary search is easily swamped by one communication). However, the merge step is not so clever, and executes without any theoretical change in its running time.

We can elicit worst-case behavior from the exact splitting step, by constructing an input where the values are shifted: each processor initially holds the values that will end up on

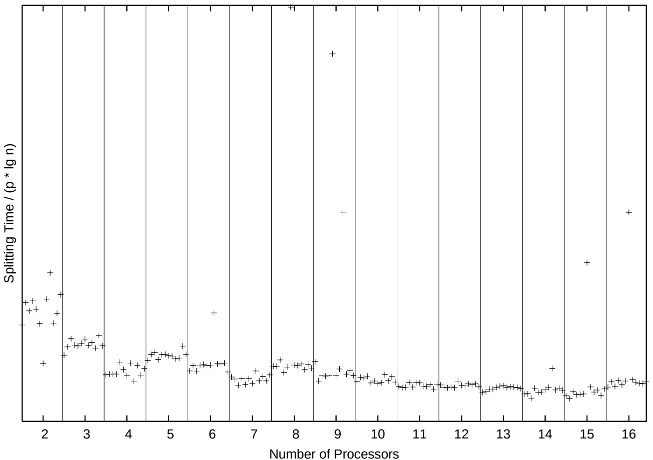

Fig. 5. Evidence of the empirical running time of $O(p \lg n)$ for exact splitting. Inside each slab, p is fixed while n increases (exponentially) from left to right. The unlabeled y-axis uses a linear scale.

TABLE II

ABSOLUTE TIMES FOR SORTING ZEROS

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
2	2.341816	2.113241	0.000511	0.1525641	0.075500
4	1.098606	0.984291	0.000625	0.0758215	0.037869
8	0.587438	0.491992	0.001227	0.0377661	0.056453
16	0.277607	0.228586	0.001738	0.0191964	0.028086
300 Million integers					
16	2.958025	2.499508	0.001732	0.1855574	0.271227

its neighbor to the right (the processor with highest rank will start off with the lowest $\lceil \frac{n}{p} \rceil$ values). This input forces the splitting step to search to the bottom, eliciting $\Theta(\lg n)$ rounds of communication. Table III gives the empirical results, which do illustrate a higher cost of exact splitting. However, the time spent in the step remains insignificant compared to the time for the local sort (assuming sufficiently large $\frac{n}{p}$). Furthermore, the overall times actually decrease with this input because much of the rerouting is done in parallel, except for the case where $p = 2$.

These results are promising, and suggest that the algorithm performance is quite robust against various types of inputs.

D. Comparison against Sample Sorting

Several prior works [1], [9], [14] conclude that an algorithm known as sample sort is the most efficient for large n and p. Such algorithms are characterized by having each processor distribute its $\lceil \frac{n}{p} \rceil$ elements into p buckets, where the bucket boundaries are determined by some form of sampling. Once the buckets are formed, a single round of all-to-all communication follows, with each processor i receiving the contents of the ith bucket from everybody else. Finally, each processor performs some local computation to place all its received elements in sorted order.

TABLE III

ABSOLUTE TIMES FOR SHIFTED INPUT

32 Million integers					
p	Total	Local Sort	Exact Split	Reroute	Merge
2	5.317869	3.814628	0.014434	1.412940	0.075867
4	2.646722	1.809445	0.018025	0.781250	0.038002
8	1.385863	0.906335	0.038160	0.384264	0.057105
16	0.736111	0.434600	0.061537	0.211697	0.028277
300 Million integers					
16	7.001009	4.601053	0.074080	2.044193	0.281683

The major drawback of sample sort is that the final distribution of elements is uneven. Much of the work in sample sorting is directed towards reducing the amount of imbalance, providing schemes that have theoretical bounds on the largest amount of data a processor can collect in the rerouting. The problem with one processor receiving too much data is that the computation time in the subsequent steps are dominated by this one overloaded processor. As a result, 1-optimality is more difficult to obtain. Furthermore, some applications require an exact output distribution; this is often the case when sorting is just one part of a multi-step process. Then an additional redistribution step would be necessary, where the elements across the boundaries are communicated.

We compare the exact splitting algorithm of this paper with two existing sample sorting algorithms.

1) A Sort-First Sample Sort: The approach of [14] is to first sort the local segments of the input, then use evenly spaced elements to determine the bucket boundaries (splitters). Because the local segment is sorted, the elements that belong to each bucket already lie in a contiguous range. Therefore, a binary search of each splitter over the sorted input provides the necessary information for the element rerouting. After the rerouting, a p-way merge puts the distributed output in sorted order. Note that the high-level sequence of sort, split, reroute, merge is identical to the algorithm presented in this paper. If we assume the time to split the data is similar for both algorithms, then the only cause for deviation in execution time would be the unbalanced data going through the merge. Define s to be the smallest value where each processor ends up with no more than $\lceil \frac{n}{p} \rceil + s$ elements. The additional cost of the merge step is simply $O(s \lg p)$. Furthermore, the cost of redistribution is $O(s)$. The loose bound given in [14] is $s = O(\frac{n}{p})$.

One of the authors of [9] has made available[10] the source code to an implementation of [14], which we use for comparison. This code uses a radix sort for the sequential task, which we drop into the algorithm given by this paper (replacing STL's `stable_sort`). The code also leaves the output in an unbalanced form; we have taken the liberty of using our own redistribution code to balance the output, and report the time for this operation separately. From the results given in Table IV, we can roughly see the linear dependence on the redistribution on n. Also, as $\frac{n}{p}$ decreases (by increasing p for fixed n), we see the running time of sample sort get closer to that of the exact splitting algorithm.

TABLE IV

COMPARISON AGAINST A SORT-FIRST SAMPLE SORT ON UNIFORM INTEGER INPUT

8 Million integers			
p	Exact	Sample	Redist
2	0.79118	0.842410	0
4	0.44093	0.442453	0.081292
8	0.24555	0.257069	0.040073

64 Million integers			
p	Exact	Sample	Redist
2	6.70299	7.176278	0
4	3.56735	3.706688	0.702736
8	2.01376	2.083136	0.324059

TABLE V

PERFORMANCE OF A SORT-LAST SAMPLE SORT ON A UNIFORM INTEGER INPUT

32 Million integers				
p	Sample	Bucket	Sort	Redist
2	0.000223	1.131244	3.993676	0.584702
4	0.000260	0.671177	1.884212	0.308441
8	0.000449	0.373997	0.926648	0.152829
16	0.000717	0.222345	0.472611	0.081180
300 Million integers				
16	0.000717	1.972227	4.785449	0.695507

The result of [9] improves the algorithm in the choice of splitters, so that s is bounded by $\sqrt{np}$. However, such a guarantee would not significantly change the results presented here: the input used is uniform, allowing regular sampling to work well enough. The largest excess s in these experiments remains under the bound of [9].

2) A Sort-Last Sample Sort: The sample sort algorithm given in [1] avoids the final merge by performing the local sort at the end. The splitter selection, then, is a randomized algorithm with high probability of producing a good (but not perfect) output distribution. Given the splitters, the buckets are formed by binary searching each of the $\lceil \frac{n}{p} \rceil$ input elements over the sorted set of splitters. Because there are at least p buckets, creating the buckets has cost $\Omega(\frac{n}{p} \lg p)$. The theoretical cost of forming buckets is at least that of merging.

Additionally, the cost of an imbalance s depends on the sequential sorting algorithm used. With a radix sort, the extra (additive) cost simply becomes $O(s)$, which is less than the imbalance cost in the sort-first approach. However, a comparison-based setting forces an increase in computation time by a super-linear $\Omega(s \lg \frac{n}{p})$.

We were unable to obtain an MPI implementation of such a sample sort algorithm, so implemented one ourselves. Table V contains the results, with the rerouting time omitted as irrelevant. By comparing against Table I, we see that the local sort step contains the expected inflation from imbalance, and the cost of redistribution is similar to that in Table IV. Somewhat surprising is the large cost of the bucket step; while theoretically equivalent to merge, it is inflated by cache inefficiencies and an oversampling ratio used by the algorithm.

IV. DISCUSSION

The algorithm presented here has much in common with the one given by [12]. The two main differences are that it performs the sequential sort after the rerouting step, and contains one round of $O(\frac{n}{p})$ communication on top of the rerouting cost. This additional round produces an algorithm that is 2-optimal in communication; direct attempts to reduce this one round of communication will result in adding another $\frac{n}{p} \lg \frac{n}{p}$ term in the computation, thus making it 2-optimal in computation.

Against sample sorts, the algorithm also compares favorably. In addition to being parameter-less, it naturally exhibits a few nice properties that present problems for some sample sort algorithms: duplicate values do not cause any imbalance, and the sort is stable if the underlying sequential sort is stable. Furthermore, the only memory copies of size $O(\frac{n}{p})$ are in the rerouting step, which is forced, and the merging, which is cache-efficient.

There lies room for further improvement in practical settings. The cost of the merging can be reduced by interleaving the p-way merge step with the element rerouting, merging subarrays as they are received. Alternatively, using a data structure such as a funnel (i.e. [3], [5]) may exploit even more good cache behavior to reduce the time. Another potential area of improvement is in the exact splitting. Instead of traversing search tree to completion, a threshold can be set; when the active range becomes small enough, a single processor gathers all the remaining active elements and completes the computation sequentially. This method, used by Saukas and Song in [12], helps reduce the number of communication rounds in the tail end of the step. Finally, this parallel sorting algorithm will directly benefit from future improvements to sequential sorting and all-to-all communication.

V. CONCLUSION

To the best of our knowledge, we have presented a new deterministic algorithm for parallel sorting that gives a strong case for exact splitting. Modulo some intricacies of determining the exact splitters, the algorithm is conceptually simple to understand, analyze, and implement. Finally, our implementation is available for academic use, and may be obtained by contacting any of the authors.

REFERENCES

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *SPAA*, 1991, pp. 3–16.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Linear time bounds for median computations," in *STOC*, 1972, pp. 119–124.

[3] G. S. Brodal and R. Fagerberg, "Funnel heap - a cache oblivious priority queue," in *Proceedings of the 13th International Symposium on Algorithms and Computation*. Springer-Verlag, 2002, pp. 219–228.

[4] T. T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. MIT Press, 1990.

[5] E. D. Demaine, "Cache-oblivious algorithms and data structures," in *Lecture Notes from the EEF Summer School on Massive Data Sets*, ser. Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[6] G. Franceschini and V. Geffert, "An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves," in *FOCS*, 2003, p. 242.

[7] A. V. Gerbessiotis and C. J. Siniolakis, “Deterministic sorting and randomized median finding on the bsp model,” in *SPAA*, 1996, pp. 223–232.

[8] M. T. Goodrich, “Communication-efficient parallel sorting,” in *STOC*, 1996, pp. 247–256.

[9] D. R. Helman, J. JáJá, and D. A. Bader, “A new deterministic parallel sorting algorithm with an experimental evaluation,” *J. Exp. Algorithmics*, vol. 3, p. 4, 1998.

[10] `http://www.eece.unm.edu/∼dbader/code.html`, 1999.

[11] A. Reiser, “A linear selection algorithm for sets of elements with weights,” *Information Processing Letters*, vol. 7, no. 3, pp. 159–162, 1978.

[12] E. L. G. Saukas and S. W. Song, “A note on parallel selection on coarse grained multicomputers,” *Algorithmica*, vol. 24, no. 3/4, pp. 371–380, 1999.

[13] V. Shah and J. R. Gilbert, “Sparse matrices in Matlab*p: Design and implementation,” *HiPC*, 2004.

[14] H. Shi and J. Schaeffer, “Parallel sorting by regular sampling,” *J. Parallel Distrib. Comput.*, vol. 14, no. 4, pp. 361–372, 1992.

[15] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.