Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers

Robert Cypher*
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120

C. Greg Plaxton†
MIT Lab for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

This paper presents a deterministic sorting algorithm, called Sharesort, that sorts n records on an n processor hypercube, shuffle-exchange or cube-connected cycles in $O(\log n(\log\log n)^2)$ time in the worst case. The algorithm requires only a constant amount of storage at each processor. The fastest previous deterministic algorithm for this problem was bitonic sort [3], which runs in $O(\log^2 n)$ time.

1 Introduction

Given n records distributed uniformly over the n processors of some fixed interconnection network, the *sorting problem* is to route the record with the ith largest associated key to processor i, $0 \le i < n$. One of the earliest parallel sorting algorithms is Batcher's bitonic sort [3], which runs in $O(\log^2 n)$ time on the hypercube, shuffle-exchange (SE) [14] and cube-connected cycles (CCC) [11]. More recently, Leighton [6] exhibited a bounded-degree, $O(\log n)$ time sorting network based on the $O(\log n)$ depth sorting circuit of Ajtai, Komlós and Szemerédi [1]. However, no efficient emulation of Leighton's sorting network is known for the hypercube, and it has been shown that such an emulation requires $\Omega(\log^2 n)$ time on the SE or CCC [5]. Hence, for these networks, the problem of closing the gap between the trivial $\Omega(\log n)$ lower bound and the $O(\log^2 n)$ upper bound remained open. A noteworthy breakthrough was provided by the randomized Flashsort algorithm of Reif and Valiant [12], which sorts every possible input permutation with high probability in $O(\log n)$ time on a CCC. In contrast, this work is the first to narrow the gap in terms of worst case, deterministic complexity.

The main result of this paper is a deterministic sorting algorithm, called Sharesort, that sorts n records in $O(\log n(\log\log n)^2)$ time on an n processor hypercube, SE or CCC. Sharesort is a stable, comparison-based sorting algorithm and requires only a constant amount of storage at each processor. A sketch of the main ideas underlying this algorithm will now be given; a more formal exposition can be found in Sections 3 to 6.

Let the *routing problem* refer to the special case of sorting in which the set of n keys forms a permutation of the integers 0 through $n-1$. While one might expect this restriction to simplify the problem considerably, the best known upper bound for the routing problem continues to be given by the complexity of sorting. This phenomenon may arise because the most natural schemes for parallel sorting are based on partitioning, recursive sorting, and merging. Within such a framework the additional information available to a routing algorithm, namely the final destination of each record, is of no apparent use beyond the top level of the recursion. Sharesort is an n^δ-way merging algorithm that succeeds in obtaining fast performance by reducing the sorting problem to *shared key sorting*, a special case of the routing problem that *can* be sustained within a recursive framework. Unfortunately, this reduction comes at the expense of a $\log\log n$ factor, which may be inherent.

A formal definition of the shared key sorting problem may be found in Section 4. A useful alternative formu-

*Supported in part by an NSF graduate fellowship
†Supported by an NSERC postdoctoral fellowship, and DARPA contracts N00014–87–K–825 and N00014–89–J–1988.

lation is given by the following two-stage problem. In the *planning* stage, the algorithm is given a hypercube (say) of dimension αd, where α is a constant strictly greater than 1, and a permutation π over a subcube X of dimension d. The algorithm is allowed to run the machine for time T_p. The planning stage is followed by the *routing* stage, in which the algorithm must route the permutation π over the subcube X, *using only the* 2^d *processors of* X. An algorithm that can successfully route any permutation π within time T_r in the routing stage solves the shared key sorting problem in time $T_p + T_r$.

Consider the performance obtained by applying known methods to this problem. Bitonic sort gives a solution with $T_p = 0$ and $T_r = O(\log^2 n)$. Using Nassimi and Sahni's parallel algorithm to set up the Benes permutation network [9] yields a solution with $T_p = O(\log^3 n)$ and $T_r = O(\log n)$. It is natural to ask whether any intermediate point exists between these two extremes with $T_p + T_r = o(\log^2 n)$, and in fact, Section 4.2 presents a solution with $T_p = T_r = O(\log n (\log\log n)^2)$. Section 4.3 sketches an improved version of this algorithm yielding $T_p = T_r = O(\log n \log\log n)$. A detailed description of the improved shared key sorting algorithm will be provided in the full paper.

But how is the sorting problem reduced to shared key sorting? Sharesort begins by recursively sorting subcubes of $n^{4/5}$ (say, see Section 6) records. Thus, the task that remains is to merge $n^{1/5}$ sorted lists of length $n^{4/5}$. A crucial observation is that the ordering information supplied by this recursive sort is sufficient to allow a large number, as many as $n^{4/5-\epsilon}$, of evenly-spaced selections to be performed over the entire set of n records. Sharesort computes $n^{4/9}$ such "splitters", that is, all of the records with ranks of the form $in^{5/9}$, $0 \le i < n^{4/9}$. Using these splitters, and once again taking advantage of the structure imposed by the recursive sort, Sharesort then partitions the n records into $2n^{29/45}$ *blocks* of length at most $n^{16/45}$. Each of these blocks is a sorted list with the additional property that the ranks of all of its constituent records lie between the same pair of adjacent splitters. Viewing the data as being arranged in a $2n^{29/45} \times n^{16/45}$ array with one block residing in each row, it becomes apparent that by first sorting all of the columns according to the permutation that sorts the first column, and then performing some appropriate compaction, every record can be brought to within $n^{5/9}$ of its correct final sorted position. The problem of sorting all of the columns according to the permutation that sorts the first column is precisely a shared key sort with $\alpha = 45/29$.

With some care in the compaction step, the remaining problem of sorting within groups of $n^{5/9}$ can be treated as a merge of $n^{1/5}$ sorted lists of length $n^{16/45}$. This merging task is performed in two stages. In the first stage, Sharesort merges sets of $n^{4/45}$ sorted lists of length $n^{16/45}$. This reorganizes each group of $n^{5/9}$ records into $n^{1/9}$ sorted lists of length $n^{4/9}$, which are then merged in the second stage. Letting $M(x, y)$ denote the task of merging n^x sorted lists of length n^y, the previous outline has reduced $M(1/5, 4/5)$ to finding splitters, building shared keys, shared key sorting, compaction, $M(4/45, 16/45)$ and $M(1/9, 4/9)$. Note that the ratio between the exponents in each of the merging tasks is exactly 4. Thus, the same procedure can be applied recursively to perform the smaller merging tasks.

2 Preliminaries

2.1 The Model of Computation

The time/space analysis of the algorithms presented in this paper assumes the following model of computation. For a hypercube, SE or CCC with n processors, the local memory of each processor is organized in $O(\log n)$-bit *words*. Each processor has a unique integer ID in the range 0 to $n-1$. The processors operate in a synchronous, SIMD fashion; at any one time, all of the processors are executing the same instruction. In a single time unit, a processor can either send a word of data to an adjacent processor and receive a word of data from an adjacent processor, or it can perform a single CPU operation on word-sized operands stored in its local memory. The ability to "unshuffle" data on the SE is assumed.

With regard to the particular problem of sorting, the objects to be sorted will be referred to as *records*. A record may have a number of associated fields, one of which contains a word-sized *key*. The ordering of the keys determines the sorted order of the records. The only operations we require on keys are copy and comparison. Note that ties can be broken in a stable manner by adding a field to each record that contains the ID of the originating processor.

2.2 Definitions

Given a cube of dimension d, let any string α of length d over the alphabet $\{0, 1, *\}$ correspond to that set of processors for which the ID "matches" α in the natural sense. It is often convenient to specify such a d-bit string as a tuple of the form $(d_0 : \alpha_0, \ldots, d_{r-1} : \alpha_{r-1})$, where r and the d_i's are nonnegative integers, $\sum_{0 \le i < r} d_i = d$, and α_i is either $*$ or a d_i-bit integer. Such a tuple corresponds to the string $\beta_0 \cdots \beta_{r-1}$, where β_i is the d_i-

bit string corresponding to the binary representation of α_i if $\alpha_i \neq *$, and $*^{d_i}$ otherwise.

Given a set of records S, and a record x in S, let *Rank*$(S,\ x)$ denote the rank of x in the set S, that is, the number of records in S having a lower associated key than x. Given an integer i, $0 \leq i < |S|$, let *Record*$(S,\ i)$ denote the record with rank i in S.

Given nonnegative integers a, b and c such that $a \geq b$ and $c < 2^{a-b}$, and a set of 2^a records S, let *Splitters*$(S,\ b,\ c)$ denote the set of 2^b records in S with ranks congruent to c modulo 2^{a-b}.

Let an $(a,\ b,\ c)$*–cube* be a subcube of dimension $a+b+c$ that is viewed as consisting of 2^c *levels*, each of which is an array with 2^a rows and 2^b columns. A set of 2^c locations that have the same row and column values will be called a *pile*.

2.3 Useful Operations

A number of previously known operations will be used to define Sharesort. These algorithms will be described in terms of the parameter n, a power of 2. With the exception of sparse enumeration sorting, all of these operations run on a hypercube, SE, or CCC with n processors in $O(\log n)$ time.

Prefix operations take as input an associative binary operator α and an array $A = A_0, A_1, \ldots, A_{n-1}$, and return the n values $\alpha(A_0, \alpha(A_1, \alpha(A_2, \ldots, \alpha(A_{i-1}, A_i))))$ where $0 \leq i < n$ [13]. One special type of prefix operation is the *segmented prefix operation* in which the input array A is divided into groups of adjacent elements and a prefix operation is applied in parallel within each of the groups.

Monotonic routing takes as input an array with n locations, m of which hold records, $0 \leq m \leq n$. Each record has associated with it a destination address in the range 0 through $n-1$, with the restriction that the destination addresses form a monotonically increasing sequence. The monotonic routing algorithm sends each of the m records to its destination address within the array [7]. Special cases of monotonic routing include the *concentrate*, in which the m records are routed to the first m array locations, the *inverse concentrate*, in which the m records are originally located in the first m array locations, and the *increment*, in which each of the m records is moved to the next higher array location.

Bit-Permute-Complement (BPC) routing performs a permutation of n records where the destination addresses are calculated by permuting and complementing the bits of the source addresses [8]. *Broadcasting* copies a record from one processor to all n processors [7].

Bitonic merging is the basic operation underlying Batcher's bitonic sort. Given two sorted lists, each of length at most n, this operation merges them into a single sorted list. A BPC route must be used to reverse one of the two lists before the merge can be performed.

Odd-even bitonic merges are used to completely sort a cube that is almost sorted. Formally, suppose that $n = 2^d$ and that a cube of dimension d', $d' > d$, is given in which every record is within n positions of its final sorted position. In parallel, sort each of the $2^{d'-d}$ subcubes of dimension d of the form $(d'-d : i,\ d : *)$, $0 \leq i < 2^{d'-d}$. Next, perform two sets of bitonic merge operations, one between subcubes of the form $(d'-d-1 : i,\ 1 : 0,\ d : *)$ and $(d'-d-1 : i,\ 1 : 1,\ d : *)$, $0 \leq i < 2^{d'-d-1}$, and the other between subcubes of the form $(d'-d-1 : i,\ 1 : 1,\ d : *)$ and $(d'-d-1 : i+1,\ 1 : 0,\ d : *)$, $0 \leq i < 2^{d'-d-1} - 1$. One may verify that these operations leave the entire cube sorted. The pair of bitonic merges that follow the sorting of the subcubes will be called *odd-even bitonic merges*. Note that a monotonic route must be performed both before and after the latter set of merges. The cost of these monotonic routes is $O(d')$.

Sparse enumeration sort is a useful sorting technique for the case when the number of records to be sorted, n, is much smaller than the number of processors available, p [10]. Sparse enumeration sort runs in $O(\log n \log p / \log(p/n))$ time.

2.4 Time Analysis

When implementing Sharesort on a hypercube computer, the running time can be calculated by simply adding together the running times of the subroutines from which it is composed. When implementing Sharesort on the SE, the running time must also include the time spent shuffling and unshuffling the data between calls to subroutines. This time is proportional to the distance between the last bit position used in one subroutine and the first bit position used in the following subroutine. It is easily verified that this cost does not change the overall complexity of the algorithm. Finally, when implementing Sharesort on a CCC there is an additional complication caused by the fact that certain bit positions require more time for communication than do others. However, it has been shown that this complication can be managed in time proportional to the running time of the SE implementation [4].

The following technical lemma will be useful in analyzing the running times of several subroutines.

Lemma 2.1 *Let a, b and ϵ be constants where $a \geq 0$, $b > 0$ and $0 < \epsilon \leq 1/2$. If for each sufficiently large value of n, there exist real numbers x and y such that*

$\epsilon \le x \le y \le 1 - \epsilon$ and $x + y \le 1 + \log^{-2} n$ and

$$f(n) \le f(\lfloor xn \rfloor) + f(\lfloor yn \rfloor) + bn \log^a n$$

then there exists a constant k (which is a function of a, b and ϵ) such that $f(n) \le kn \log^{a+1} n$ for all sufficiently large n.

Proof: The proof is by induction on n. The induction hypothesis is that the lemma holds for $n \le m$, and it will be shown that this implies the lemma holds for $n \le m/(1-\epsilon)$. Let $z = -\log(1-\epsilon)$. Note that $z > 0$ and $\log x \le \log y \le -z$.

For $n \le m/(1-\epsilon)$,

$$\begin{aligned} f(n) &\le kxn \log^{a+1}(xn) + kyn \log^{a+1}(yn) + bn \log^a n \\ &\le (1 + \log^{-2} n) kn (\log n - z)^{a+1} + bn \log^a n \\ &= kn \log^{a+1} n - (a+1) kzn \log^a n + bn \log^a n \\ &\quad + O(n \log^{a-1} n) \\ &\le kn \log^{a+1} n \end{aligned}$$

provided that $k > b/(a+1)z$ and that n is sufficiently large. □

3 Splitter Finding

This section defines the algorithm $\mathsf{FindSplitters}(a,\ b,\ c)$.

Input: A set S of 2^{a+b} records organized as 2^a sorted lists of length 2^b, where a and b are positive integers, and an integer c in the range $0 \le c < b$. Let S_i denote the ith sorted list, $0 \le i < 2^a$, and let S denote the entire set of 2^{a+b} records. The records of S are stored in a subcube of dimension $a + b$, with $\mathit{Record}(S_i,\ j)$ stored in processor $(a : i,\ b : j)$, $0 \le i < 2^a$, $0 \le j < 2^b$.

Processors: The 2^{a+b} processors of the subcube containing S.

Output: The set $T \stackrel{\text{def}}{=} \mathit{Splitters}(S,\ c,\ 0)$, with $\mathit{Record}(T,\ k)$ stored in processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$.

Running time: $O((a+b)^2/(b-c))$.

Example: Given n records organized as $n^{1/2}$ sorted lists of length $n^{1/2}$, this algorithm can be used to compute $n^{1/4}$ evenly-spaced splitters in $O(\log n)$ time.

Algorithm $\mathsf{FindSplitters}(a,\ b,\ c)$

1. Let $d = \lfloor \frac{b-c}{2} \rfloor$. Let $X = \bigcup_{0 \le i < 2^a} \mathit{Splitters}(S_i,\ b - d,\ 0)$. Sort the set X. Note that $|X| = 2^{a+b-d}$. Implementation: sparse enumeration sort. Running time: $O((a+b)^2/(b-c))$.
2. Let $r(k) = \mathit{Rank}(S,\ \mathit{Record}(T,\ k)) = k2^{a+b-c}$, $L = \mathit{Splitters}(X,\ c,\ 0)$, and $U = \mathit{Splitters}(X,\ c,\ 2^a - 1)$. Observe that $i2^d - 2^{a+d} + 2^a + 2^d - 1 \le \mathit{Rank}(S,\ \mathit{Record}(X,\ i)) \le i2^d$. Hence, $r(k) - 2^{a+d} + 2^a + 2^d - 1 \le \mathit{Rank}(S,\ \mathit{Record}(L,\ k)) \le r(k)$ and $r(k) + 2^a - 1 \le \mathit{Rank}(S,\ \mathit{Record}(U,\ k)) \le r(k) + 2^{a+d} - 2^d$. Mark every record x in S such that $\mathit{Record}(L,\ k) \le x < \mathit{Record}(U,\ k)$ for some k, $0 \le k < 2^c$. No record belongs to more than one such interval since the choice of d guarantees that $\mathit{Rank}(S,\ \mathit{Record}(U,\ k)) < \mathit{Rank}(S,\ \mathit{Record}(L,\ k + 1))$, $0 \le k < 2^c - 1$. Let Y denote the set of marked records. Note that $T \subset Y$. Implementation: concentrate, broadcast, bitonic merge, prefix sum. Running time: $O(a+b)$.
3. Compute $r_0(k) = \mathit{Rank}(S,\ \mathit{Record}(L,\ k))$ and store the result in processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$. Implementation: concentrate, sum. Running time: $O(a+b)$.
4. Compute $r_1(k) = \mathit{Rank}(Y,\ \mathit{Record}(L,\ k))$ and store the result in processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$. Implementation: prefix sum, concentrate, sum. Running time: $O(a+b)$.
5. At processor $(a + b - c : 0,\ c : k)$, compute $\mathit{Rank}(Y,\ \mathit{Record}(T,\ k)) = r_1(k) + r(k) - r_0(k)$, $0 \le k < 2^c$. Running time: $O(1)$.
6. Sort the set Y. Note that $|Y| \le 2^c(2^{a+d+1} - 2^a - 2^{d+1} + 2) < 2^{a+c+d+1}$. Implementation: sparse enumeration sort. Running time: $O((a+b)^2/(b-c))$.
7. Use the ranks computed in Step 5 to extract the desired set T from the sorted set Y. Implementation: inverse concentrate, concentrate. Running time: $O(a+b)$.

4 Shared Key Sorting

This section defines the shared key sorting algorithm. The subroutine $\mathsf{Balance}()$ will be presented first and the main algorithm follows.

4.1 Color Balancing

This section defines the color balancing subroutine $\mathsf{Balance}(r,\ c,\ l,\ k,\ p)$.

Input/Processors: A subcube of 2^{r+c+l} processors, which will be viewed as an $(r,\ c,\ l)$*-cube*, and a set S of records distributed one per processor over some subset of the processors in the first 2^{c-1} columns of level 0 of the cube. The parameters r, c, l, k and p are all nonnegative integers, where $l > c$ and p is a prime, $2^{c-1} < p < 2^c$. Let s denote the greatest integer such that $|S| \leq 2^{r+c-s-1}$.

Output: A subset U of S such that $|S \setminus U| \leq 2^{r+c-2s-2}$, called the "balanced records", and for each record x in U an assignment to the fields $Color(x)$, $Row(x)$, $Column(x)$ and $Stage(x)$, where $Color(x) = \lfloor 2^r Rank(S,\ x)/|S| \rfloor$, $0 \leq Row(x) < 2^r$, $0 \leq Column(x) < p$ and $Stage(x) = k$. For any balanced records x and y, $Row(x) = Row(y) \Rightarrow Column(x) \neq Column(y)$ and $Color(x) = Color(y) \Rightarrow Column(x) \neq Column(y)$. Furthermore, two monotonic routes that can be implemented entirely within level 0 are sufficient to move the balanced records from their original positions to the positions given by $Row(x)$ and $Column(x)$.

Running time: $O(r^2/(l-c)+l)$.

Example: The input is a cube of $2n$ processors arranged in $n^{1/4}$ rows, $2n^{1/4}$ columns and $n^{1/2}$ levels with $n^{5/12}$ records in the first $n^{1/4}$ columns of level 0. The records are assigned colors in the range 0 through $n^{1/4}-1$. The algorithm balances all but $n^{1/3}/2$ records in $O(\log n)$ time.

Algorithm Balance$(r,\ c,\ l,\ k,\ p)$

1. Append a field to each record that saves the ID of the processor originally holding the record. Concentrate the records into the first $|S|$ processors and sort them. Compute $|S|$ and then for each record x in S, compute $Color(x)$. Implementation: concentrate, sparse enumeration sort, prefix operations. Running time: $O((r+l)^2/l)$.

2. Sort the records of the set S according to their original positions. Then return them to their original positions. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O((r+l)^2/l)$.

3. If $|S| \leq 2^r$ then every record in S has been assigned a different color. Let $U = S$, and for each record x in S, set $Row(x)$ and $Column(x)$ to the input position of x, set $Stage(x) = k$, and return. Otherwise, go to Step 4. Running time: $O(1)$.

4. Copy each record to the same row and column position in each of the first p levels. Let $S_{i,j}$ be the pile of p records in row i and column j. Permute the copies of the records in the first p levels of the cube as follows. In level h, $0 \leq h < p$, move each record from row i and column j to row i and column $j + ih \bmod p$. Implementation: broadcast, monotonic routes. Running time: $O(c)$.

5. Define a *collision* to be the mapping of two records in the same level and with the same color to the same column. Calculate the number of collisions in each of the first p levels. Note that for any pair of piles, $S_{i,j}$ and $S_{i',j'}$, that have the same color there is at most one collision between records in $S_{i,j}$ and records in $S_{i',j'}$ in the first p levels. Because there are at most 2^{c-s-1} piles with each color, the total number of collisions in the first p levels is less than $2^{2c-2s-3}2^r = 2^{r+2c-2s-3}$ and one of the first p levels must have fewer than $2^{r+2c-2s-3}/p < 2^{r+c-2s-2}$ collisions. Determine which level in the first p levels has the fewest collisions. Implementation: sparse enumeration sort, prefix operations. Running time: $O(r+c+r^2/(l-c))$.

6. Let level i contain the smallest number of collisions. For each record x in level i that is not involved in a collision with a record from a lower row, set $Row(x)$ and $Column(x)$ to the row and column positions to which it was permuted, and set $Stage(x) = k$. These records form the set of balanced records U. Note that each collision prevents at most one record from becoming balanced, so at most $2^{r+c-2s-2}$ records in level i will remain unbalanced. Then undo the permutation by sending each record back to its original location. Finally, each balanced record x in level i sends $Row(x)$, $Column(x)$ and $Stage(x)$ back to level 0. Implementation: prefix operation, monotonic routes. Running time: $O(r+c)$.

4.2 Shared Key Sorting

This section defines the shared key sorting routine SharedKeySort$(a,\ b)$.

Input: A set S of 2^{a+b} records organized as 2^a lists of 2^b records each, where the 2^b records in each list have the same key field, a and b are positive integers and $b - a/2 = \Theta(a)$. Let S_i denote the ith list of 2^b records, $0 \leq i < 2^a$, and let $T = \bigcup_{0 \leq i < 2^a} Record(S_i,\ 0)$. The records of S are stored in a subcube of dimension $a+b$, with the jth element of the list S_i being stored in processor $(a : i,\ b : j)$, $0 \leq i < 2^a$, $0 \leq j < 2^b$.

Processors: The 2^{a+b} processors of the subcube containing S.

Output: The (stably) sorted set S. In other words, if x is the jth record in list S_i, then x is moved to processor $(a : Rank(T,\ x),\ b : j)$, $0 \leq i < 2^a$, $0 \leq j < 2^b$.

Running time: $O(a \log^2 a)$. This is improved to $O(a \log a)$ in Section 4.3.

Example: Given n records organized as $n^{1/2}$ lists of length $n^{1/2}$, and assuming that records belonging to the same list have the same key value, this algorithm sorts the n records in $O(\log n (\log\log n)^2)$ time.

The operation $\mathsf{SharedKeySort}(a,\ b)$ is performed by calling $\mathsf{SharedKeySort}'(a,\ b,\ 0)$, defined below. The third parameter keeps track of the depth of recursion.

Algorithm $\mathsf{SharedKeySort}'(a,\ b,\ depth)$

1. If $a \le depth$, then perform a bitonic sort of the 2^{a+b} records, and return. Running time: $O(a^2)$ if $a \le depth$, $O(1)$ otherwise.

2. Let $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$ and $d = \lceil \log c \rceil + 2$. The processors will be viewed as forming an $(r,\ c,\ b)$*–cube*. For each record x in S, let $Level(x)$ denote the level of the processor in which x was originally located. Let p be the smallest prime between 2^c and 2^{c+1}. Note that p is guaranteed to exist [2]. Calculate p by comparing each integer between 2^c and 2^{c+1} with all smaller positive integers and testing for divisibility. Implementation: broadcast, divide, prefix operations. Running time: $O(a)$.

3. For $i = 0$ to d, do the following. Simulating a machine with 2^{a+b+1} processors arranged as an $(r,\ c+1,\ b)$*–cube*, call $\mathsf{Balance}(r,\ c+1,\ b,\ i,\ p)$ on the set of unbalanced records in T. Return each balanced record to its original row and column and broadcast its stage, color, row and column values to the $2^b - 1$ other records in its pile. At most 2^{a-2^i} piles remain unbalanced after the ith stage. Let $U_i = \{x \mid Stage(x) = i\}$, $0 \le i \le d$. Running time: $O(a \log a)$.

4. For $i = 0$ to d, do the following. First, simulating a machine with 2^{a+b+1} processors, move the records in U_i to their balanced positions. In other words, for each record x in U_i, send x to processor $(r : Row(x),\ c+1 : Column(x),\ b : Level(x))$. Next, map the array of balanced records from row-major to column-major storage. This moves each record x in U_i to processor $(c+1 : Column(x),\ r : Row(x),\ b : Level(x))$. Then concentrate the set of records U_i in the new column-major order, and if $i > 0$ then route the concentrated records so that they are stored immediately following the records in U_{i-1}. In other words, store the records of U_i in the $|U_i|$ consecutive processors beginning with processor number $\sum_{j=0}^{i-1} |U_i|$. At this point, each list S_i is stored in 2^b consecutive processors and any set of (at most 2^r) lists S_i that share the same stage and column numbers is stored in a set of (at most 2^{r+b}) consecutive processors. Implementation: monotonic route, BPC route, concentrate, monotonic route. Running time: $O(a \log a)$.

5. When the records in each set U_i are moved to their balanced positions (given by their row and column fields), they can be sorted by first sorting the columns, then monotonically routing within columns to the rows given by their color fields, and then sorting within rows. Steps 5a through 5g sort the columns of each set U_i.

   (a) Permute the data so that the record that was stored in processor $(c : W,\ r : X,\ b-r : Y,\ r : Z)$ is sent to processor $(c : W,\ b-r : Y,\ r : X,\ r : Z)$. Implementation: BPC route. Running time: $O(a)$.

   (b) Note that the 2^r records residing in any subcube of the form $(a+b-r : X,\ r : *)$ share the same key, stage, color and column values. For each such subcube, save these four values in additional fields associated with the record in processor $(a+b-r : X,\ r : depth)$. Running time: $O(1)$.

   (c) Sort the level 0 records, resolving comparisons by stage first, column second and color third. Replace the key field of each level 0 record by the rank that it achieves in this sort, undo the sort, and copy the new key value throughout each pile. Implementation: sparse enumeration sort (twice), broadcast. Running time: $O(a)$.

   (d) Divide the records into groups of 2^{2r} consecutive records and recursively call $\mathsf{SharedKeySort}'(r,\ r,\ depth+1)$ to sort each group.

   (e) The key, stage, color and column fields corresponding to this level of the recursion have been overwritten. Restore these fields from the copies saved in Step 5b. Implementation: broadcast. Running time: $O(a)$.

   (f) Perform odd-even bitonic merges of sorted lists of length 2^{2r}, resolving comparisons as in Step 5c. Running time: $O(a)$.

   (g) Permute the data so that the record that was stored in processor $(c : W,\ b-r : Y,\ r : X,\ r : Z)$ is sent to processor $(c : W,\ r : X,\ b-r : Y,\ r : Z)$. Implementation: BPC route. Running time: $O(a)$.

6. For $i = 0$ to d, do the following. First, simulating a machine with 2^{a+b+1} processors, move each record x in U_i to processor $(c+1 : Column(x),\ r :$

$Color(x)$, $b : Level(x)$). Next, map the array of balanced records from column-major to row-major storage. This moves each record x in U_i to processor $(r : Color(x),\ c+1 : Column(x),\ b : Level(x))$. Then concentrate the set of records U_i in the new row-major order, and if $i > 0$ then route the concentrated records so that they are stored immediately following the records in U_{i-1}. In other words, store the records of U_i in the $|U_i|$ consecutive processors beginning with processor number $\sum_{j=0}^{i-1} |U_i|$. Implementation: monotonic route, BPC route, concentrate, monotonic route. Running time: $O(a \log a)$.

7. At this point, each set U_i is sorted by color. Steps 7a through 7g complete the sorting of each set U_i by sorting within color groups.

   (a) Permute the data so that the record that was stored in processor $(r-1 : W,\ c+1 : X,\ b-c-1 : Y,\ c+1 : Z)$ is sent to processor $(r-1 : W,\ b-c-1 : Y,\ c+1 : X,\ c+1 : Z)$. Implementation: BPC route. Running time: $O(a)$.

   (b) Note that the 2^{c+1} records residing in any subcube of the form $(a+b-c-1 : X,\ c+1 : *)$ have the same key and stage values. For each such subcube, save these two values in additional fields associated with the record in processor $(a+b-r : X,\ r : depth)$. Running time: $O(1)$.

   (c) Sort the level 0 records, resolving comparisons by stage first and key second. Replace the key field of each level 0 record by the rank that it achieves in this sort, undo the sort, and copy the new key value throughout each pile. Implementation: sparse enumeration sort (twice), broadcast. Running time: $O(a)$.

   (d) Divide the records into groups of 2^{2c+2} consecutive records and call SharedKeySort$'(c+1,\ c+1,\ depth+1)$ to sort each group.

   (e) The key and stage fields corresponding to this level of the recursion have been overwritten. Restore these fields from the copies saved in Step 7b. Implementation: broadcast. Running time: $O(a)$.

   (f) Perform odd-even bitonic merges of sorted lists of length 2^{2c+2}, resolving comparisons as in Step 7c. Running time: $O(a)$.

   (g) Permute the data so that the record that was stored in processor $(r-1 : W,\ b-c-1 : Y,\ c+1 : X,\ c+1 : Z)$ is sent to processor $(r-1 : W,\ c+1 : X,\ b-c-1 : Y,\ c+1 : Z)$. At this point, the records within each stage are sorted correctly. All that remains is to merge the records from different stages. Implementation: BPC route. Running time: $O(a)$.

8. Sort the records in level 0 to determine their final sorted position. Broadcast the final sorted position of each record in level 0 to the remaining records in its pile. This gives the final sorted position of each record. Implementation: sparse enumeration sort, broadcast. Running time: $O(a)$.

9. For $i = 0$ to d, move the records in U_i to their final sorted positions. Implementation: monotonic route. Running time: $O(a \log a)$.

Space analysis: Each record contains a key, stage, color and column field. The row and level fields have been introduced for expository purposes only. Note that for $i > 0$, the level i records remain within the set of level i processors at all times. Certain records make use of four additional fields in order to save the values of the key, stage, color and column fields computed at a particular depth of recursion. This is performed by Step 5b, before the first recursive call, and by Step 7b, before the second recursive call (in the latter case, only the key and stage fields are saved). No processor saves more than a single set of fields, since the stack-like storage scheme ensures that each level is used by at most one depth of the recursion. Finally, there are never more than 2 records located at a processor. Hence, the algorithm requires only a constant number of memory words per processor.

Time analysis: Let $SKS(a,b,d)$ denote the running time of SharedKeySort$'(a,\ b,\ d)$. If $a \le d$ then $SKS(a,b,d) = O(d^2)$ and if $a > d$ then

$$\begin{aligned} SKS(a,b,d) &= O(a\log a) + SKS(r,r,d+1) \\ &\quad + SKS(c+1,c+1,d+1), \end{aligned}$$

where $r = \lfloor a/2 \rfloor$ and $c = \lceil a/2 \rceil$. Thus, a top-level call to SharedKeySort$'(a,\ b,\ 0)$ generates only recursive calls of the form $SKS(a',a',d)$ where

$$a' \le \frac{a}{2^d} \prod_{0 \le i < d} \left(1 + \frac{2^{i+2}}{a}\right).$$

The logarithm of the product term is easily seen to be $O(1)$ for $d \le \log a$. Hence, $a' = O(a/2^d)$ for $d \le \log a$, and the maximum depth of recursion is $\log a - \log\log a + O(1)$. This bound on the maximum depth of recursion implies that the cost of all of the bitonic sorts performed in Step 1 is $O(a \log a)$. Now consider the total amount of time spent at some depth of recursion d that is less than the maximum depth. Lemma 2.1 shows that this time is also bounded by $O(a \log a)$. Therefore, $SKS(a,b,0) = O(a \log^2 a)$.

4.3 Improved Shared Key Sorting

This section sketches the main ideas underlying a somewhat more complicated implementation of SharedKeySort$(a,\ b)$ running in $O(a \log a)$ time. The new version also requires only a constant amount of storage at each processor. A complete description of the improved shared key sorting routine will be provided in the full paper.

The new algorithm is similar to that described in Section 4.2, except that the record coloring and balancing operations have been separated from the record moving operations. The record coloring and balancing is performed by a subroutine called PlanRoute() and the record motion is performed by subroutine DoRoute(). As before, the shared key sort can be viewed as a tree of recursive calls to smaller shared key sorts. The subroutine PlanRoute() performs the coloring and balancing for all levels of the recursion, and the subroutine DoRoute() performs the record motions for all levels of the recursion. Thus the coloring and balancing of records at all levels of the recursion is performed before any records are moved. The coloring and balancing information needed by DoRoute() is provided by records, called *routing records*, that are created by PlanRoute(). Each routing record consists of a constant number of words of data, and at most one routing record will be stored at any single processor.

The separation of record coloring and balancing from record motion permits a more efficient algorithm. Specifically, the sparsity of the input to PlanRoute() allows the recursive calls to be performed in parallel in different subcubes. Unfortunately, each of the recursive calls generated by DoRoute() must handle one record per processor, and so the same technique cannot be applied. Rather, the complexity of subroutine DoRoute() is reduced by moving the records from all of the different stages in parallel. For instance, DoRoute() performs the data movement of Step 4 of the algorithm of Section 4.2 in $O(\log n)$ time as opposed to $O(\log n \log\log n)$. There are two main modifications to the algorithm that make such an improvement possible. First, a simple subroutine is defined that allows n records belonging to k stages to be separated (by stage) in $O(\log n)$ time using nk processors. Second, a more powerful balancing suboutine is defined that can balance all but a very small fraction of the records in a single stage. The idea behind this balancing routine is to reduce the number of colors by a $\log\log n$ factor and allow a corresponding increase in the number of records of a given color that can be mapped to the same column in a single stage. This has the effect of increasing the size of the recursive calls over the rows by a factor of $\log\log n$, but Lemma 2.1 can be used to show that this increase has no effect on the asymptotic complexity of the algorithm.

5 Merging

This section defines the algorithm ShareMerge$(a,\ b)$. Let the *skew* of a call to either ShareMerge$(a,\ b)$ or SharedKeySort$(a,\ b)$ denote the value of the ratio b/a.

Input: A set S of 2^{a+b} records organized as 2^a sorted lists of length 2^b, where a and b are positive integers such that the $b/a - \frac{1}{2}(3+\sqrt{13}) = \Theta(1)$. Let S_i denote the ith sorted list, $0 \le i < 2^a$, and let S denote the entire set of 2^{a+b} records. The records of S are stored in a subcube of dimension $a+b$, with $Record(S_i,\ j)$ stored in processor $(a:i,\ b:j)$, $0 \le i < 2^a$, $0 \le j < 2^b$. It will be helpful to view these 2^{a+b} processors as being arranged in a two-dimensional array with 2^a rows and 2^b columns.

Processors: The 2^{a+b} processors of the subcube containing S.

Output: The sorted set S, that is, processor $(a+b:i)$ contains a copy of $Record(S,\ i)$, $0 \le i < 2^{a+b}$.

Running time: $O(a \log^2 a)$.

Example: Given n records organized as $n^{1/5}$ sorted lists of length $n^{4/5}$, this algorithm produces a single sorted list of length n in $O(\log n (\log\log n)^2)$ time.

Algorithm ShareMerge$(a,\ b)$

1. If $a \le \tau$, where τ is a positive integer to be specified below, then perform the entire merging task with a sequence of a bitonic merges, and return. Otherwise, go to Step 2. Running time: $O(\tau a)$.

2. Let $c = \lfloor b^2/(a+2b) \rfloor$. Compute $Splitters(S,\ b-c,\ 0)$ by calling FindSplitters$(a,\ b,\ b-c)$. Note that $b - c = \Theta(b)$. Running time: $O(a)$.

3. Broadcast the sorted list $Splitters(S,\ b-c,\ 0)$ to each row. Running time: $O(a)$.

4. For each record x in S, define the *color* of x to be $\lfloor Rank(S,\ x)/2^{a+c} \rfloor$. Thus, there are 2^{a+c} records of color i, $0 \le i < 2^{b-c}$. The set of records of a given color forms a *color class*. Note that the boundaries between color classes are given by the splitters computed in Step 2. Steps 4a to 4c are performed simultaneously within each row. Intuitively, the objective within row i is to partition the 2^b records of set S_i into monochromatic sorted lists of length 2^c. This goal is generally unattainable, since the number of records in S_i of some color may not be a multiple of 2^c. By introducing 2^b dummy records in

each row, however, it is possible to partition S_i into 2^{b-c+1} monochromatic sorted lists of length 2^c. To take care of the additional factor of 2, each row of 2^b processors will simulate 2^{b+1} virtual processors.

(a) Merge *Splitters*$(S,\ b-c,\ 0)$ with S_i, and compute α_j, the number of records of color j in S_i, $0 \leq j < 2^{b-c}$. Implementation: bitonic merge, segmented sum, concentrate. Running time: $O(a)$.

(b) Compute $\beta_j = \sum_{0 \leq k < j}(-\alpha_k \bmod 2^c)$. Broadcast β_j to every record of color j. Implementation: prefix sum, inverse concentrate, segmented prefix operation. Running time: $O(a)$.

(c) Simulating 2^{b+1} virtual processors in each row, route *Record*$(S_i,\ k)$, which has some color j, to processor $(a : i,\ b+1 : k+\beta_j)$. Note that $k + \beta_j \leq 2^b - 1 + (2^{b-c}-1)(2^c-1) < 2^{b+1}$. Every virtual processor that does not receive a record creates a dummy record with color $+\infty$. Implementation: inverse concentrate. Running time: $O(a)$.

5. The preceding operations have organized the set S into $2^{a+b-c+1}$ monochromatic (with respect to the non-dummy records) sorted lists of length 2^c, which will be referred to as *blocks*. Simulating 2^{a+b+1} processors, call SharedKeySort$(a+b-c+1,\ c)$ to separate the color classes. Running time: $O(a \log a)$ (see the discussion below).

6. Steps 6a to 6c eliminate the dummy records along with the associated factor of 2 simulation overhead. This is done in order to prevent the simulation overhead from growing exponentially with the depth of recursion, which would adversely affect the running time of the algorithm. Note that a straightforward compaction of the non-dummy records (prefix sum, concentrate) is inappropriate because it would not preserve the sortedness of the blocks in the sense required by Step 7.

   (a) A block that contains at least one dummy record will be referred to as *underpopulated.* A block that is not underpopulated is *overpopulated.* Note that there are at most 2^a underpopulated blocks of any particular color. Route the ith underpopulated block of color j to subcube $(b-c : j,\ a : i,\ c : *)$, $0 \leq i < 2^a$, $0 \leq j < 2^{b-c}$. Implementation: prefix sum, monotonic route. Running time: $O(a)$.

   (b) Mark every processor that did not receive a record in the previous step. Compute the rank of each marked processor, that is, the number of marked processors with lower IDs. Mark every non-dummy record that did not get routed in the previous step. Compute the rank of each marked record, that is, the number of marked records in virtual processors with lower IDs. Route the ith marked record to the ith marked processor. Implementation: prefix sums, monotonic route. Running time: $O(a)$.

   (c) Now every processor contains a single record, and every subcube of the form $(b-c : j,\ a : i,\ c : *)$ contains 2^c records of color j. Such a subcube is not necessarily sorted because it may have received sorted sublists from more than one block during the previous two steps. However, it received records from at most one underpopulated block, and at most two overpopulated blocks. Hence, the records in such a subcube represent the concatenation of at most three sorted lists. Sort each of these subcubes of dimension c. Implementation: increment route, prefix sum, constant number of monotonic routes and bitonic merges. Running time: $O(a)$.

7. The task that remains is to merge 2^a sorted lists of length 2^c within each color class. These merges will be performed with two recursive calls. Let $d = \lfloor ab/(a+2b) \rfloor$, and partition the records of each color class into 2^{a-d} groups of 2^d sorted lists of length 2^c. The first recursive call, ShareMerge$(d,\ c)$, sorts the records within each group. The second recursive call, ShareMerge$(a-d,\ c+d)$, sorts the records within each color class.

Analysis: Let the *skew* of a call to ShareMerge$(a,\ b)$ be the ratio b/a and let $M(a,b)$ denote the running time of a call to ShareMerge$(a,\ b)$ with skew ξ where $\xi - \frac{1}{2}(3+\sqrt{13}) = \Theta(1)$. Thus, $M(a,b)$ satisfies the recurrence

$$M(a,b) \leq M(a',b') + M(a'',b'') + O(a \log a),$$

where $a' = \lfloor ab/(a+2b) \rfloor$, $b' = \lfloor b^2/(a+2b) \rfloor$, $a'' = a - a'$ and $b'' = a' + b'$, as long as the skew ξ associated with every recursive call satisfies $\xi - \frac{1}{2}(3+\sqrt{13}) = \Theta(1)$. The motivation for defining the recursive calls in this manner is that, ignoring floors, $b/a = b'/a' = b''/a''$.

Of course, the effect of taking floors cannot be ignored, but it is a straightforward exercise to prove that both b'/a' and b''/a'' lie in the range $(b/a)(1 \pm O(1/a))$. The skew cannot grow by more than a constant factor, because a large skew implies that both a and b decrease geometrically, which implies that the total change to the skew is bounded by a constant. Similarly, the skew cannot become too small. For example, assume for that the original skew ξ is such that $\xi - \frac{1}{2}(3+\sqrt{13}) = \epsilon$ for some positive constant ϵ and that there is some recursive call for which the skew is less than $\frac{1}{2}(3+\sqrt{13}) + \epsilon/2$. Let i be the depth at which such a small skew is first encountered. Then the skew at all depths less than i is greater than $\frac{1}{2}(3+\sqrt{13})$, which implies that both a and b had decreased geometrically, and the skew could not have drifted such a large distance unless the subproblems at depth i are of constant size. Thus, setting τ to a sufficiently large constant will prevent the skew from becoming too small.

Note that such variations in skew do not affect the asymptotic complexity of the operations performed within $\mathsf{ShareMerge}()$. In particular, every call to $\mathsf{SharedKeySort}()$ generated by Step 5 can be forced to have skew ξ such that $\xi - 1/2 = \Theta(1)$, which leads to the stated running time. Finally, because the skew remains sufficiently large, both a and b decrease geometrically and Lemma 2.1 can be used to show that the recurrence solves to give $M(a,b) = O(a \log^2 a)$.

6 Sorting

This section defines the algorithm $\mathsf{ShareSort}(a)$.

Input/Processors: A set S of 2^a records stored, one per processor, in a subcube of dimension a.

Output: The sorted set S, that is, processor $(a : i)$ contains a copy of $Record(S,\ i)$, $0 \le i < 2^a$.

Running time: $O(a \log^2 a)$.

Algorithm $\mathsf{ShareSort}(a)$

1. If $a \le \tau$ then sort the set S using bitonic sort. Running time: $O(\tau^2)$.

2. Let $b = \lceil \xi a \rceil$, where ξ is a real constant satisfying $\frac{1}{6}(1+\sqrt{13}) < \xi < 1$. Partition the input subcube into 2^{a-b} subcubes of dimension b, where the ith such subcube corresponds to $(a-b : i,\ b : *)$, $0 \le i < 2^{a-b}$. Recursively execute $\mathsf{ShareSort}(b)$ within each of these subcubes in parallel.

3. Call $\mathsf{ShareMerge}(a-b,\ b)$ to complete the sort. Note that $b = \Omega(a)$. Running time: $O(a \log^2 a)$.

Analysis: Let $S(a)$ denote the running time of $\mathsf{ShareSort}(a)$. If $a \le \tau$ then $S(a) = O(\tau^2)$, and if $a > \tau$ then

$$S(a) \le S(\lceil \xi a \rceil) + O(a \log^2 a).$$

Setting ξ and τ to suitable positive constants, this recurrence gives $S(a) = O(a \log^2 a)$.

7 Extensions

This paper has focused on the problem of sorting n records on an n processor hypercube, SE or CCC. The full version of the paper will describe extensions and applications of the shared key sorting technique to other sorting problems. The results will include:

1. A relaxation of the conditions under which the subroutine $\mathsf{SharedKeySort}(a,\ b)$ runs in $O(a \log a)$ time. Specifically, the requirement that $b - a/2 = \Theta(a)$ can be relaxed to $a = \Theta(b)$. This immediately implies that $\mathsf{ShareMerge}(a,\ b)$ runs in $O(a \log^2 a)$ time for $a = \Theta(b)$, and that the constant ξ defined in $\mathsf{ShareSort}(a)$ can take on an arbitrary value lying strictly between 0 and 1.

2. A non-constructive, non-uniform version of Sharesort that runs in $O(\log n \log\log n)$ time with $O(1)$ storage on the hypercube, SE and CCC. It is interesting to note that this *deterministic* algorithm is based upon known techniques for *randomized* routing.

3. An algorithm for shared key sorting that outperforms the obvious generalization of Sharesort when the the number of records being sorted, n, exceeds the number of processors available, p, by a sufficiently large polylogarithmic factor.

4. An $O(\log n \log\log n)$ time, $O(1)$ storage sorting algorithm for sorting $n/\log n$ records on an n processor multibutterfly computer, as defined by Upfal [15].

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] A. D. Aleksandrov, A. N. Kolmogorov, and M. A. Lavrent'ev. *Mathematics: Its Content, Methods and Meaning.* MIT Press, Cambridge, MA, 1963.

[3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[4] R. E. Cypher. *Efficient Communication in Massively Parallel Computers.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, August 1989.

[5] R. E. Cypher. Theoretical aspects of VLSI pin limitations. *SIAM J. Comput.*, 22:58–63, 1993.

[6] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34:344–354, 1985.

[7] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C–30:101–107, 1981.

[8] D. Nassimi and S. Sahni. A self routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers*, C–30:332–340, 1981.

[9] D. Nassimi and S. Sahni. Parallel algorithms to set up the Benes permutation network. *IEEE Transactions on Computers*, C–31:148–154, 1982.

[10] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM*, 29:642–667, 1982.

[11] F. P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24:300–309, 1981.

[12] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *JACM*, 34:60–76, 1987.

[13] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[14] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C–20:153–161, 1971.

[15] E. Upfal. An $O(\log N)$ deterministic packet-routing scheme. *JACM*, 39:55–70, 1992.