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 Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

March 5, 1996

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 Batcher's bitonic sort, which runs in $O(\log^2 n)$ time.

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

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, $0 \leq i < n$. One of the earliest parallel sorting algorithms is Batcher's bitonic sort [3], which runs in $O(\log^2 n)$ time on the hypercube [10], shuffle-exchange [17], and cube-connected cycles [14]. More recently, Leighton [9] 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 shuffle-exchange or cube-connected cycles [6]. 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 [15], which sorts every possible input permutation with high probability in $O(\log n)$ time on a cube-connected cycles. 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, shuffle-exchange, or cube-connected cycles. Sharesort is a stable, comparison-based sorting algorithm, and requires only a constant amount of storage at each processor. Although the asymptotic performance of Sharesort is markedly superior to that of bitonic sort, the multiplicative constant hidden by the O-notation is several orders of magnitude larger in the case of Sharesort. As a result, Sharesort runs more slowly than bitonic sort for all practical values of n.

A somewhat simplified overview of this algorithm will now be given; a more formal exposition can be found in Sections 3 to 7. Throughout this description, a and b will denote constants where $0 < a < b < 1$. Sharesort begins by partitioning the n records into n^{1-b} groups of n^b records. These groups are then sorted recursively and in parallel. Thus all that remains is to merge n^{1-b} sorted lists each of length n^b. This merging is accomplished by using a technique which is similar to bucket sort. In particular, the n records are assigned to n^a buckets, each of size n^{1-a}, such that for any i and j where $0 \leq i < j < n^a$, all of the records in bucket i have smaller keys than all of the records in bucket j. Next, the records are routed to these buckets. More specifically, the records in bucket i, where $0 \leq i < n^a$, are routed to processors in^{1-a} through $(i+1)n^{1-a} - 1$, and stored one per processor. Finally, the merge is completed by sorting the records within each of the buckets.

As is evident from the above description, the merging procedure consists of three main steps, namely: (i) assigning records to buckets, (ii) routing records to buckets, and (iii) sorting the records within each bucket. All of these steps make critical use of the fact that the records have been previously arranged in n^{1-b} sorted lists of length n^b. For example, consider the assignment of records to buckets. Because no assumptions are made regarding the distribution of the key values, this assignment cannot be performed by simply examining certain bit positions of the keys. Instead, Sharesort proceeds by computing the set of n^{1-b} records with ranks of the form in^b, where i is an integer such that $0 \leq i < n^{1-b}$. These records, called *splitters*, are the records with the smallest key values in each of the buckets.

By merging a sorted list of the splitters with each of the sorted lists of length n^b, each record can determine to which bucket it is assigned. Although the problem of calculating the splitters would be prohibitively expensive in a general setting, the fact that the records are arranged in sorted lists of length n^b allows the splitters to be found efficiently. Specifically, a deterministic sampling scheme is used which takes a small number of evenly-spaced records from each of the sorted lists. These samples are then sorted and used to obtain tight bounds on the values of the splitters.

The fact that the records have been arranged in sorted lists of length n^b is also critical to the routine which routes the records to the appropriate buckets. Consider an arbitrary sorted list of n^b records. The records in this list must be routed to n^a different buckets. Note that because the records in the list are sorted, the set of records assigned to each of the buckets forms a block of consecutive records. Thus if we associate a unique color to each of the buckets and if we color each record according to the bucket to which it must be routed, each sorted list of n^b records will consist of n^a blocks of consecutive, identically-colored records. As a result, the records can be routed to the buckets by sorting these blocks according to their colors. This restricted version of the sorting problem, in which the records to be sorted form blocks of consecutive records that share the same key value, will be called the *shared key sorting problem*. The overall efficiency of the Sharesort algorithm results largely from an efficient solution to the shared key sorting problem.

The third and final step in the merging procedure sorts the records within the buckets. As was just shown, a shared key sort is used to route the records to the correct buckets. Following this shared key sort each bucket consists of n^{1-b} blocks of consecutive records, where the records within each block are in sorted order (because they come from a single sorted list of n^b records and the shared key sort is stable). Thus the problem of *sorting* the records within a single bucket can be solved by *merging* these sorted lists. In fact, by performing a small amount of preprocessing, all of the sorted lists in each of the buckets can be made to be of equal length. The records within each bucket are then sorted by performing two recursive calls to the merging procedure.

The remainder of the paper is organized as follows. Section 2 defines the model of computation, examines some implementation and time analysis issues, introduces the notation that will be used, and reviews several previously known algorithms. Sections 3 to 7 contain a detailed description and analysis of the Sharesort algorithm. The algorithm is presented in a top-down fashion. Section 3 contains the top-level subroutine, which reduces the sorting task to a merging problem. This merging problem is solved by a call to the recursive subroutine ShareMerge(), which is presented in Section 4. At each level of recursion, ShareMerge() calls subroutine FindSplitters() to compute a large number of evenly-spaced splitter records and then calls subroutine SharedKeySort() to route the resulting blocks of identically-colored records to the appropriate buckets. Sections 5 and 6 present the FindSplitters() and SharedKeySort() subroutines, respectively. Finally, Section 7 describes a somewhat more intricate shared key sorting algorithm that improves upon the asymptotic performance of the routine given in Section 6 by a $\log \log n$ factor.

2 Preliminaries

2.1 The Model of Computation

The model of computation assumed throughout the remainder of this paper is that of a distributed memory parallel computer which operates in a synchronous, single-instruction-stream, multiple-data-stream (SIMD) mode. The computer consists of n processors, each of which has a local memory and a unique integer ID in the range 0 to $n-1$. Each local memory contains a constant number of *words*; a bound on the size of these words is given below. The processors communicate with one another by sending messages over communication links which connect certain pairs of processors. In a single time unit, a processor can either send a word of data over a communication link 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.

It will be assumed that the communication links are arranged in a *hypercube* [10], *shuffle-exchange* [10, 17], or *cube-connected cycles* [14] topology. In a hypercube, each processor x is connected to the $\log n$ processors which have binary representations that differ from x in exactly one bit position. The bit positions of the processor ID's will also be referred to as *dimensions*, and an n-processor hypercube will also be called a $(\log n)$-*dimensional* hypercube. An important property of the hypercube is that it can be partitioned into a number of smaller hypercubes. Specifically, given integers m and k where $k < m$, an m-dimensional hypercube can be partitioned into 2^{m-k} disjoint k-dimensional hypercubes by selecting a set of $m-k$ bit positions and considering any two processors to be in the same k-dimensional hypercube if and only if their ID's match in those $m-k$ bit positions. Such k-dimensional hypercubes will be referred to as *subcubes*.

In a shuffle-exchange each processor x is connected to processors x_s, x_u, and x_e via *shuffle*, *unshuffle*, and *exchange* links, respectively. The binary representation of x_s (respectively, x_u) is obtained from the binary representation of x by performing a left (respectively, right) circular shift. The binary representation of x_e is obtained from the binary representation of x by complementing the least significant bit. The reader is referred to the paper by Preparata and Vuillemin [14] for the definition of the cube-connected cycles.

The parallel computer will be used to sort a set of *records*. A record has a constant number of word-sized fields, one of which is a *key*. The relative order of the keys determines the sorted order of the records. The only operations that we require on keys are copy and comparison. It will be assumed that the keys are unique, as ties can be broken in a stable manner by adding a field to each record that contains the ID of the originating processor.

We now address the issue of word size. Letting r denote the maximum number of bits in any of the records to be sorted, the definition of a record implies that the number of bits in a word is $\Omega(r)$. Furthermore, our algorithm makes use of $O(\log n)$-bit auxiliary variables, independent of the size of the records. All of our time analysis assumes a machine with an $\Omega(r + \log n)$-bit word size. In most applications, $r = O(\log n)$, in which case $\Theta(\log n)$-bit words suffice in order to achieve the stated time bounds.

2.2 Implementation Issues and Time Analysis

The majority of the routines presented in this paper perform the same sequence of operations at each processor and are thus well-suited to an SIMD implementation. In particular, if the basic shared key sorting algorithm given in Section 6 is used, the entire algorithm naturally runs in an SIMD mode. However, the improved shared key sorting algorithm given in Section 7 uses a routine, $\mathsf{PlanRoute}'()$, which performs simultaneous recursive calls in subcubes of different sizes. As a result, although the same set of routines will be executed in all of the subcubes, some of them will require more iterations in the larger subcubes. Thus in order to obtain an SIMD implementation, it is necessary for the processors in the smaller subcubes to be idle while the additional iterations are performed in the larger subcubes. The time analysis for the routine $\mathsf{PlanRoute}'()$ assumes that the smaller subcubes require as much time as the larger subcubes, so the fact that some of the processors are forced to be idle does not change the overall running time of the algorithm.

When implementing Sharesort on a hypercube, 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 shuffle-exchange, 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 cube-connected cycles 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 always be managed in time proportional to the running time of the shuffle-exchange implementation [5]. Our bounds on the running time of Sharesort also apply to the butterfly topology (with or without "wrap-around" connections), which is closely related to the cube-connected cycles.

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 integer n there exist real numbers x and y, and a function $f(n)$ such that $\epsilon \leq x \leq y \leq 1-\epsilon$, $x + y \leq 1 + \log^{-2} n$, and*

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

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

Proof: The proof is by induction on n. For the base case, note that the lemma holds for $n \leq \lceil 1/\epsilon \rceil$. The induction hypothesis is that the lemma holds for $n \leq m$, where $m \geq \lceil 1/\epsilon \rceil$, and it will be shown that this implies the lemma holds for $n \leq \lfloor m/(1-\epsilon) \rfloor$. Note that $\lfloor m/(1-\epsilon) \rfloor \geq \lfloor m + \epsilon m \rfloor \geq m + 1$. Let $z = -\log(1-\epsilon)$, and observe that $z > 0$ and $\log x \leq \log y \leq -z$.

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

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

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

2.3 Notation and Definitions

The following notation and definitions will be used throughout this paper. Given a hypercube 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 $\alpha_i \ne *$, 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, $0 \le i < |S|$, let *Record*$(S,\ i)$ denote the record with rank i in S.

Given nonnegative integers a, b and c such that $a \ge b$ 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*.

2.4 Useful Operations

A number of previously known algorithms will be used as subroutines in 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 an n-processor hypercube, shuffle-exchange, or cube-connected cycles in $O(\log n)$ time.

Prefix operations take as input an associative binary operator α and an array $A = A_0, \ldots, A_{n-1}$, and return the n values $\alpha(A_0, \alpha(A_1, \ldots, \alpha(A_{i-1}, A_i)))$ where $0 \le i < n$ [16]. 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. *Suffix operations* and *segmented suffix operations* are identical to prefix operations and segmented prefix operations, respectively, except that the operator α is applied to suffixes of the array A.

Monotone 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 monotone routing algorithm sends each of the m records to its destination address within the array [10]. Special cases of monotone 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 [11].

Broadcasting copies a record from one processor to all n processors [10].

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 monotone route must be performed both before and after the latter set of merges. The cost of these monotone routes is $O(d')$.

Sparse enumeration sort is a useful sorting technique when the number of records to be sorted, n, is much smaller than the number of processors available, p [12]. Sparse enumeration sort runs in $O(\log n \, \log p / \log(p/n))$ time.

Finally, it is possible to efficiently simulate a large parallel computer with a small one. Specifically, for any constant c, an n processor hypercube, shuffle-exchange, or cube-connected cycles can simulate a cn processor machine of the same topology with only a constant factor slowdown [8].

3 The Top-Level Routine

This section defines the top-level routine of the Sharesort algorithm. Sharesort belongs to the class of "bottom-up" sorting algorithms based on the principle of recursive merging. Another member of this class is bitonic sort [3]. The bitonic sorting algorithm proceeds by splitting the input list into two equal-sized sublists, sorting these sublists recursively and in parallel, and then merging the resulting pair of sorted sublists. In order to achieve a smaller depth of recursion, Sharesort partitions the input into a much larger (polynomial, in fact) number of equal-sized sublists. The performance of any sorting algorithm of this kind is determined by the efficiency of its merging procedure. A fast subroutine for merging a polynomial number of polynomial-sized sorted lists will be presented in Section 4. A formal definition of the top-level routine of Sharesort will now be given.

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 $\mathit{Record}(S,\ i)$, $0 \leq i < 2^a$.

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

Algorithm $\mathsf{ShareSort}(a)$

1. **Base case.** If $a \leq \tau$ then sort the set S using bitonic sort, and return. Running time: $O(a^2)$ if $a \leq \tau$, $O(1)$ otherwise.

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

3. **Merge.** Call $\mathsf{ShareMerge}(a - b,\ b)$ to complete the sort. Note that the above definition of ξ guarantees that $b/(a - b) > (3 + \sqrt{13})/2$, as will be required. Running time: $O(a \log^2 a)$.

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

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

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

4 Merging

This section presents the subroutine $\mathsf{ShareMerge}(a,\ b)$, which uses 2^{a+b} processors to merge 2^a sorted lists of length 2^b into a single sorted list of length 2^{a+b}. There are six main ideas underlying the efficient performance of this subroutine.

The first idea is that if b is sufficiently large, then we can take advantage of the partial sortedness of the input in order to perform a large (polynomial) number of evenly-spaced selection operations. The details of the selection procedure are given in Section 5. The records returned by these selection operations will be referred to as "splitters".

The second idea is that the splitters can be used to partition the set of input records into large (polynomial-sized) subcubes of records with approximately the same rank in the final sorted order. We will refer to each such subcube of similar records as a "block" (two records are deemed to be "similar" if they lie between the same pair of adjacent splitters). A partitioning of very nearly the desired type can be obtained by independently partitioning each sorted list into a number of blocks. These partitionings can be performed by first merging each sorted list with a copy of the sorted list of splitter records, and then performing certain prefix operations.

Unfortunately, we have no control over the distribution of ranks within a particular sorted list. Hence, it is quite possible that, for example, a particular record cannot be put into the same block as any other record in its sorted list. This difficulty is overcome by the third idea, which is to recognize that there is an effective bound on the number of times that this kind of adverse phenomenon can occur within a single sorted list. In fact, by merely doubling the number of records in each sorted list through the addition of "wildcard" dummy records, it is possible to obtain a partitioning of the desired type.

The fourth idea is that the blocks of records formed in the previous step can be sorted efficiently using a shared key sort. The basic subroutine for performing this operation is described in Section 6. A more intricate version that achieves slightly better asymptotic performance is presented in Section 7.

The fifth idea is to recognize that once the blocks have been sorted with respect to one another, the remaining sorting task is equivalent to a smaller merging problem that can be reduced to two merging problems having the same "aspect ratio" as the original merging problem. Hence, the same merging algorithm can be used recursively. There is one problem to be overcome first, however. Namely, the factor of two increase in the number of records caused by the introduction of the dummy records must be dealt with; the number of dummy records cannot be allowed to grow exponentially with the depth of recursion.

The sixth and final idea is that the dummy records can be eliminated before the recursive calls by a carefully-chosen sequence of a bitonic merges, prefix operations, and monotone routes. Furthermore, the elimination of the dummy records can be performed without giving up the sortedness of the blocks, an effect that would otherwise cost an additional $\log a$ factor in performance.

In the following formal description of subroutine ShareMerge(), let γ denote the constant $(3+\sqrt{13})/2$.

Input/Processors: 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 $b/a > \gamma$ and $b/a - \gamma = \Theta(1)$. 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 $\mathit{Record}(S_i,\ j)$ 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.

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

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 $\mathsf{ShareMerge}(a,\ b)$

1. **Base case.** 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: $O(a^2)$ if $a \le \tau$, $O(1)$ otherwise.

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

3. **Distribute splitters.** Broadcast the sorted list $\mathit{Splitters}(S,\ b-c,\ 0)$ to each row. In other words, send a copy of the jth record in this list to every processor of the form $(a:*,\ b:j)$, $0 \le j < 2^{b-c}$. Running time: $O(a)$.

4. **Cluster records of similar rank.** For each record x in S, define the *color* of x to be $\lfloor \mathit{Rank}(S,\ x)/2^{a+c} \rfloor$. Thus, there are 2^{a+c} records of color i, $0 \le i < 2^{b-c}$. The set of all records of a given color forms a *color class.* Note that the boundaries between color classes are determined by the splitters computed in Step 2. Steps 4a to 4e are performed simultaneously within each row. Ideally, the objective within row i would be to partition the 2^b records of set S_i into monochromatic sorted lists of length 2^c. Unfortunately, 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. In order to handle the additional factor of 2, each row of 2^b processors will simulate 2^{b+1} virtual processors.

   (a) Merge $\mathit{Splitters}(S,\ b-c,\ 0)$ with S_i. Implementation: bitonic merge. Running time $O(a)$.

   (b) Compute α_j, the number of records of color j in S_i, and route the result to processor $(a:i,\ b:j)$, $0 \le j < 2^{b-c}$. Implementation: segmented prefix operation, concentrate. Running time: $O(a)$.

   (c) Compute $\beta_j = \sum_{0 \le k < j} (-\alpha_k \bmod 2^c)$, $0 \le j < 2^{b-c}$. Implementation: prefix operation. Running time: $O(a)$.

   (d) Broadcast the value β_j to every record of color j, $0 \le j < 2^{b-c}$. Implementation: inverse concentrate, segmented prefix operation. Running time: $O(a)$.

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

   Total running time: $O(a)$.

5. **Sort by color.** 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. Each of these lists will be referred to as a *block.* Simulating 2^{a+b+1} processors, call

SharedKeySort$(a + b - c + 1,\ c)$ to sort the blocks by color. It remains to sort within sets of blocks of the same color, which are now contiguous. Running time: $O(a \log a)$ (see Section 7, as well as the analysis below).

6. **Compaction.** Steps 6a to 6e 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 operation, concentrate) is inadequate because it would not preserve the sortedness of the blocks in the sense required by Step 7.

   (a) A block that does not contain any dummy records will be referred to as *complete*. A block that is not complete is *incomplete*. For each record, determine whether the parent block is complete or incomplete. Note that the dummy records of a given block, if any, reside in the highest-numbered processors of that block. Implementation: broadcast. Running time: $O(a)$.

   (b) Note that there are at most 2^a incomplete blocks of color j, $0 \le j < 2^{b-c}$. Route the records of the ith incomplete block of color j to subcube $(b-c : j,\ a : i,\ c : *)$, $0 \le i < 2^a$, $0 \le j < 2^{b-c}$. Implementation: segmented prefix operation, monotone route. Running time: $O(a)$.

   (c) Each processor received 0 or 1 records in Step 6b; mark every processor that did not receive a non-dummy record. Compute the rank of each marked processor, that is, the number of marked processors with lower IDs. Implementation: prefix operation. Running time: $O(a)$.

   (d) Mark every record that was not routed in Step 6b; this is exactly the set of records that belong to complete blocks. 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 operation, monotone route. Running time: $O(a)$.

   (e) 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. However, it received a sublist from at most one incomplete block, and at most two sublists from complete blocks. Hence, the list of records in such a subcube represent the concatenation of at most three sorted lists. The following merging procedure, applied within each of these subcubes of 2^c processors in parallel, forms a single sorted list in each subcube.

      i. Let x_i denote the value of key i, $0 \le i < 2^c$. Since the input represents the concatenation of at most three sorted lists, $x_{i-1} \le x_i$ for all but at most 2 values of i, $1 \le i < 2^c$, Mark those (at most 2) x_i's for which $x_{i-1} > x_i$. Implementation: increment route. Running time: $O(a)$.

      ii. Determine which records belong to the ith sorted sublist of the input, $i = 0, 1, 2$. Implementation: prefix operation. Running time: $O(a)$.

iii. Shift sorted list 1 so that it begins at processor 0 (i.e., the lowest-numbered processor in this subcube of dimension c). Implementation: prefix operation, monotone route. Running time: $O(a)$.

iv. Pad sorted lists 0 and 1 to length 2^c with dummy $+\infty$ records and merge the resulting lists. The prefix of the merged list that contains all of the non-dummy records will be referred to as sorted list 3. Implementation: bitonic merge. Running time: $O(a)$.

v. Shift sorted list 2 so that it begins at processor 0. Implementation: prefix operation, monotone route. Running time: $O(a)$.

vi. Pad sorted lists 2 and 3 to length 2^c with dummy $+\infty$ records and merge the resulting lists. Discard the dummy records. Implementation: bitonic merge. Running time: $O(a)$.

Total running time: $O(a)$.

7. **Recursive merges.** It remains to merge the 2^a sorted lists of length 2^c within each color class. These merges will be performed by 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, $\mathsf{ShareMerge}(d,\ c)$, sorts the records within each group. The second recursive call, $\mathsf{ShareMerge}(a-d,\ c+d)$, sorts the records within each color class.

Analysis: Let the *skew* of a call to either $\mathsf{ShareMerge}(a,\ b)$ or $\mathsf{SharedKeySort}(a,\ b)$ denote the value of the ratio b/a and let $M(a,b)$ denote the running time of a call to $\mathsf{ShareMerge}(a,\ b)$ with skew ξ where $\xi > \gamma$ and $\xi - \gamma = \Theta(1)$. Thus, $M(a,b)$ satisfies the recurrence

$$M(a,b) \le M(a',b') + M(a'',b'') + O(a \log a), \qquad (1)$$

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 $\xi > \gamma$ and $\xi - \gamma = \Theta(1)$. 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. To deal with this technicality, assume that a call to $\mathsf{ShareMerge}(a_i,\ b_i)$ generates a recursive call to $\mathsf{ShareMerge}(a_{i+1},\ b_{i+1})$, $1 \le i < k$, and let ξ_i denote the skew associated with the ith level of the recursion, b_i/a_i. Further assume that $\xi_1 > \gamma$ and $\xi - \gamma = \Theta(1)$, as required by the algorithm.

Lemma 4.1 *For $1 \le i < k$, $|\xi_{i+1} - \xi_i| = O(1/a_i + 1/b_i)$.*

Proof: To obtain a lower bound on the skew resulting from the first recursive call, note that $b'/a' \ge (b^2 - a - 2b)/ab = b/a + O(1/a + 1/b)$. Similarly, one may argue that $b'/a' \le b/a + O(1/a + 1/b)$, and that $b''/a'' = b/a + O(1/a + 1/b)$. $\Box$

Lemma 4.2 *There exists a constant $\rho < 1$ such that $a_{i+1} \le \rho a_i$, $1 \le i < k$. Furthermore,*

$$|\xi_{i+1} - \xi_1| = O(1/a_i),$$

and, for a sufficiently large choice of the constant τ in Step 1, $\xi_{i+1} > \gamma$ and $\xi_{i+1} - \gamma = \Theta(1)$, $1 \le i < k$.

Proof: Note that $a_i > \tau$, $1 \le i < k$. The lemma will be proven by induction on i. Take ρ to be $2/3$, say. Now certainly $b_1 > a_1$, so for a sufficiently large choice of the constant τ, $a_1/3 \le \lfloor a_1 b_1/(a_1 + 2b_1)\rfloor \le a_1/2$ and $a_1/2 \le a_1 - \lfloor a_1 b_1/(a_1 + 2b_1)\rfloor \le 2a_1/3$. Hence, either recursive call gives $a_2 \le \rho a_1$. The bound $|\xi_2 - \xi_1| = O(1/a_1)$ follows from the preceding lemma. Thus, $\xi_2 > \gamma$ and $\xi_2 - \gamma = \Theta(1)$ as long as τ is chosen sufficiently large.

Now consider the induction step. We will assume that the lemma holds up to some value of i strictly less than $k-1$, and prove it for $i+1$. By the induction hypothesis, $\xi_{i+1} > \gamma$ and $\xi_{i+1} - \gamma = \Theta(1)$. Hence, the inequality $a_{i+2} \le \rho a_{i+1}$ follows by the same argument as was used in the base case. Combining this with the induction hypothesis, we find that $a_{j+1} \le \rho^j a_1$, $1 \le j \le i+1$. By repeated application of the preceding lemma, we find that

$$\begin{aligned} |\xi_{i+2} - \xi_1| &= O\left(\sum_{1 \le j \le i+1} 1/a_j\right) \\ &= O(1/a_{i+1}), \end{aligned}$$

as required. This bound implies that for a sufficiently large choice of the constant τ, $\xi_{i+2} > \gamma$ and $\xi_{i+2} - \gamma = \Theta(1)$, which completes the proof. □

Note that minor variations in skew of the sort permitted by the preceding lemma do not affect the asymptotic complexity of the operations performed within ShareMerge(). In particular, one may easily verify that the definitions of c and γ guarantee that every call to SharedKeySort() generated in Step 5 will have skew ξ such that $\xi > 1/2$ and $\xi - 1/2 = \Theta(1)$, which leads to the stated running time. Finally, Lemma 4.2 implies that both a and b decrease geometrically with the depth of recursion; hence, Lemma 2.1 can be used to show that the recurrence of Equation 1 solves to give $M(a,b) = O(a \log^2 a)$.

5 Splitter Finding

This section defines the algorithm FindSplitters(a, b, c), which provides a method for performing a large number of selection operations in $O(\log n)$ time when the n-record input is presented as a number of sorted sublists. This bound is clearly optimal since it matches (to within a constant factor) the trivial $\Omega(\log n)$ lower bound based on diameter. Note that the lower bound applies even in the case where only a single selection operation is to be performed.

The sorted sublists must be sufficiently large in order for FindSplitters() to run in optimal $O(\log n)$ time. For example, the case in which the input consists of n sorted sublists of unit size corresponds to the most commonly considered version of the selection problem, and there is no known deterministic algorithm for cube-type computers that can perform even a single (general) selection in $O(\log n)$ time. Currently, the asymptotically fastest selection algorithm known for cube-type computers is based on the $O((\log \log n)^2)$ selection algorithm devised by Cole and Yap for the parallel comparison model [4], and runs in $O(\log n \log \log n)$ time [13].

If the input consists of $n^{1-\epsilon}$ sorted sublists of length n^ϵ for some constant $\epsilon > 0$, algorithm FindSplitters() can be used to perform $O(n^{\epsilon'})$ evenly-spaced selections in $O(\log n)$ time, for any constant $\epsilon' < \epsilon$. Furthermore, the algorithm can easily be adapted to perform an arbitrary set of $n^{\epsilon'}$ selections, rather than an evenly-spaced set. A more detailed definition and description of algorithm FindSplitters() will now be presented.

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 $Record(S_i,\ j)$ 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 $T \stackrel{\text{def}}{=} Splitters(S,\ c,\ 0)$, with $Record(T,\ k)$ 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 FindSplitters(a, b, c)

1. **Sort a sparse sample.** Let $d = \left\lfloor \frac{b-c}{2} \right\rfloor$. Let $X = \bigcup_{0 \le i < 2^a} Splitters(S_i,\ b-d,\ 0)$. Sort the set X using all 2^{a+b} processors. Note that $|X| = 2^{a+b-d}$. Implementation: sparse enumeration sort. Running time: $O((a+b)^2/(b-c))$.

2. **Broadcast approximate splitters.** Let $L = Splitters(X,\ c,\ 0)$, $U = Splitters(X,\ c,\ 2^a - 1)$, and $r(k) = Rank(S,\ Record(T,\ k)) = k2^{a+b-c}$. The motivation for defining L and U in this manner is that the kth record of L (respectively, U) provides a good lower (respectively, upper) approximation for the kth record in the desired set T. To verify this claim, first observe that $i2^d - 2^{a+d} + 2^a + 2^d - 1 \le Rank(S,\ Record(X,\ i)) \le i2^d$. Hence, $r(k) - 2^{a+d} + 2^a + 2^d - 1 \le Rank(S,\ Record(L,\ k)) \le r(k)$ and $r(k) + 2^a - 1 \le Rank(S,\ Record(U,\ k)) \le r(k) + 2^{a+d} - 2^d$. The following pair of steps broadcast a copy of the set $L \cup U$ to each subcube of the form $(a : i,\ b : *)$, $0 \le i < 2^a$.

   (a) Note that the set $L \cup U$ is a subset of X; hence, it is already sorted. Move the ith record of $L \cup U$ to processor $(a : 0,\ b : i)$, $0 \le i < |L \cup U| = 2^{c+1} \le 2^b$. Implementation: concentrate. Running time: $O(a+b)$.

   (b) Copy the ith record in $L \cup U$ to every processor in the subcube $(a : *,\ b : i)$, $0 \le i < |L \cup U|$. Implementation: broadcast. Running time: $O(a)$.

3. **Identify candidate records.** The following steps are now performed within each of the 2^a subcubes of the form $(a : i,\ b : *)$ in parallel, $0 \le i < 2^a$. The goal is to mark every record x in S such that $Record(L,\ k) \le x < Record(U,\ k)$ for some k, $0 \le k < 2^c$, and then to compute certain rank information. Note that no record can belong to more than one such interval, since the choice of d guarantees that $Rank(S,\ Record(U,\ k)) < Rank(S,\ Record(L,\ k+1))$, $0 \le k < 2^c - 1$. Let Y denote the set of marked records. Note that $T \subset Y$.

(a) Merge the sorted list S_i with the sorted list $L \cup U$. If $|L \cup U|$ is less than 2^b, then pad it to length 2^b with dummy $+\infty$ records before performing the merge. Implementation: bitonic merge. Running time: $O(b)$.

(b) Compute $r_0^{(i)}(k)$, the number of records in S_i with key value less than that of the kth record in L, and store the result in processor $(a : i,\ b : k)$, $0 \le k < 2^c$. Implementation: prefix operation, concentrate. Running time: $O(b)$.

(c) Mark those records in S_i belonging to the set Y. Implementation: prefix operation. Running time: $O(b)$.

(d) Compute $r_1^{(i)}(k)$, the number of marked records in S_i with key value less than that of the kth record in L, and store the result in processor $(a : i,\ b : k)$, $0 \le k < 2^c$. Implementation: prefix operation, concentrate. Running time: $O(b)$.

4. **Compute intermediate ranks.** Perform the following steps within each of the 2^c subcubes of the form $(a : *,\ b : k)$ in parallel, $0 \le k < 2^c$.

   (a) Compute $r_0(k) = \sum_{0 \le i < 2^a} r_0^{(i)}(k) = \mathit{Rank}(S,\ \mathit{Record}(L,\ k))$ and store the result in processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$. Implementation: prefix operation. Running time: $O(a)$.

   (b) Compute $r_1(k) = \sum_{0 \le i < 2^a} r_1^{(i)}(k) = \mathit{Rank}(Y,\ \mathit{Record}(L,\ k))$ and store the result in processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$. Implementation: prefix operation. Running time: $O(a)$.

5. **Compute splitter ranks.** At processor $(a + b - c : 0,\ c : k)$, compute $r_2(k) = r_1(k) + r(k) - r_0(k) = \mathit{Rank}(Y,\ \mathit{Record}(T,\ k))$, $0 \le k < 2^c$. Running time: $O(1)$.

6. **Sort candidate records.** Sort the set Y using all 2^{a+b} processors. 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. **Extract splitters.** The following steps make use of the ranks computed in Step 5 in order to extract the desired set T from the sorted set Y.

   (a) Route a packet containing the integer k from processor $(a + b - c : 0,\ c : k)$ to processor $(a + b : r_2(k))$, $0 \le k < 2^c$. Implementation: inverse concentrate. Running time: $O(a + b)$.

   (b) The record of Y stored at the processor that received k in the previous step is the kth record in the desired output set T. Route this record to processor $(a + b - c : 0,\ c : k)$, $0 \le k < 2^c$. Implementation: concentrate. Running time: $O(a + b)$.

6 Basic Shared Key Sorting

This section contains an algorithm for solving the shared key sorting problem. The input to the shared key sorting problem is a two-dimensional array of records, stored one per processor. All of the records in each row have identical key values, but records that are in different rows have different key values. The problem is to sort the input in row-major order, in a stable fashion. Because the set of records within each row have the same key value, the effect of this sort is equivalent to that obtained by sorting the columns.

Note that for all of the columns to be sorted, the same permutation must be applied to each of them. This observation suggests an approach to solving the shared key sorting problem. We can divide the algorithm into two parts: a *planning stage* and a *routing stage*. The planning stage determines the sorted position of each record and calculates a path from each record to its sorted position. The routing stage then sends each record to its sorted position along the path calculated in the planning stage.

Because the same permutation must be applied to all of the columns, the planning stage can focus on any one column and calculate paths that send the records in that column to their sorted positions. As long as these paths reside entirely within the given column, the same set of paths can be used to route within each of the other columns as well without causing any collisions between records in different columns. Thus the planning stage will focus on a single column of the array and will create a set of paths which remain within that column while sending each record to its sorted position. Then the routing stage will route the records in all of the columns in parallel, following the paths calculated by the planning stage.

The key to this approach is the sparsity of the data during the planning stage. Although all of the processors are available, only a small amount of data (the records from a single column) needs to be considered. The first part of the planning stage, namely calculating the sorted position of the records in the given column, is easy. Because of the sparsity of the data, the records can be sorted efficiently with a sparse enumeration sort. The challenge is the second part of the planning stage, namely calculating paths for the records. The sum of the amount of time spent calculating the paths and the amount of time required to route the records along the paths must be minimized.

In order to describe how these paths are calculated, we first introduce a convenient way of viewing the input to the shared key sorting problem. Assume that the input is a two-dimensional array of records, A, with 2^a rows and 2^b columns. We can also view this as a three-dimensional array, C, with 2^r rows, 2^c columns, and 2^b levels where $r = \lfloor a/2 \rfloor$ and $c = \lceil a/2 \rceil$. Thus each level of C corresponds to a column of A and each pile of 2^b records in C that have matching row and column positions corresponds to a row of A. The planning stage will focus on a single level of C and determine paths which remain within that level while sending each record in that level to its sorted position.

Now consider how the records in a single level of C can be sent to their sorted positions without leaving their current level. One technique is to send each record first to its destination row in C (without leaving its current column) and then to its destination column in C (without leaving its destination row). A difficulty with this approach is that several records that begin in the same column can be destined for the same row, thus creating collisions when each record is sent to its destination row. This difficulty can be avoided if the records in each row are first permuted (with an efficient algorithm) so that no two records that are in a single column are destined for the same row.

Thus one possible algorithm is as follows. The planning stage first performs a sparse enumeration sort to calculate the sorted positions of the records in a single level of C. Next, the planning stage assigns a "color" to each record which specifies the row in C to which the record must be sent. Then the planning stage determines an efficient algorithm for permuting the records in each row so that no two records that are in a single column have the same color. The routing stage first implements these permutations within the rows. Next, the routing stage sends each record to its destination row (without leaving its current column). Finally, the routing stage sends each record to its destination column (without leaving its destination row). The operation of sending the records to their destination rows (or columns) is implemented with a recursive call to shared key sort in which the key fields are the destination row (or column) values calculated during the planning stage.

Unfortunately, we have not been able to use this algorithm to obtain a fast shared key sort because we do not know how to implement the planning stage efficiently. In particular, no efficient techniques are known for calculating the permutations that send records with matching colors to different columns. However, by weakening the requirements of the planning stage, an efficient algorithm can be created. Specifically, the records will first be divided into a small number of different *classes*. Instead of guaranteeing that no two records with the same color appear in the same column, the planning stage will only guarantee that no two records that have the same color *and* are in the same class will appear in the same column.

Following the planning stage, the routing stage first separates the records according to class. Then it permutes the records in each class according to the permutations calculated in the planning stage. Next, the routing stage sends each record to its destination row (without leaving its current column). Then the routing stage sends each record to its destination column (without leaving its destination row). The operation of sending the records to their destination rows (or columns) is implemented with a recursive call to shared key sort in which the key fields are the class number followed by the destination row (or column) values calculated during the planning stage. At this point, the records in each class are completely sorted. Finally, the routing stage merges the different classes to obtain a single sorted array of records.

The remainder of this section presents the details of an $O(\log n\,(\log\log n)^2)$ shared key sorting algorithm. A slightly simpler version of this algorithm exists, but in order to simplify the overall presentation we have chosen to follow the same structure as in the $O(\log n \log\log n)$ shared key sorting algorithm given in Section 7. The subroutines $\mathsf{CalcPrime}()$ and $\mathsf{Balance}()$ are presented in Sections 6.2 and 6.3, respectively. The subroutine $\mathsf{CalcPrime}()$ is used to compute certain prime numbers. The purpose of $\mathsf{Balance}()$ is to separate the records into classes.

6.1 Shared Key Sorting

This subsection defines the algorithm $\mathsf{SharedKeySort}(a,\ b)$.

Input: A set S of 2^{a+b} records organized as 2^a lists of 2^b records each, where a and b are positive integers with $b > \lceil a/2 \rceil + 1$ and $b - \lceil a/2 \rceil - 1 = \Theta(a)$, and the 2^b records in each list have the same key field. Let S_i denote the ith list of 2^b records, $0 \le i < 2^a$, and let T represent that set of 2^a records obtained by taking the first record from each list S_i. The records of S are stored in a subcube of dimension $a + b$; the jth element of list S_i is stored in processor $(a : i,\ b : j)$, $0 \le i < 2^a$, $0 \le j < 2^b$. Each record x in S has fourteen fields, namely: $\mathit{Key}(x)$, $\mathit{Level}(x)$, $\mathit{SourceRow}(x)$, $\mathit{SourceColumn}(x)$, $\mathit{Class}(x)$, $\mathit{IntColumn}(x)$, $\mathit{Color}(x)$, $\mathit{DestRow}(x)$, $\mathit{DestColumn}(x)$, $\mathit{TempA}(x)$, $\mathit{TempB}(x)$, $\mathit{TempC}(x)$, $\mathit{TempD}(x)$, and $\mathit{TempE}(x)$. These fields will be called the key, level, source row, source column, class, intermediate column, color, destination row, destination column, temporary A value, temporary B value, temporary C value, temporary D value, and temporary E value of x, respectively.

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 sent to processor $(a : \mathit{Rank}(T,\ x),\ b : j)$, $0 \le i < 2^a$, $0 \le j < 2^b$.

Running time: $O(a \log^2 a)$; this is improved to $O(a \log a)$ in Section 7.

Example: Given n records organized as $n^{1/2}$ lists of length $n^{1/2}$, and assuming that the records belonging to the same list have the same key value, this algorithm uses n processors to sort 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 $\mathsf{SharedKeySort}'(a,\ b,\ \mathit{depth})$

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

2. **Compute dependent parameters.** Each processor locally calculates $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$, and $d = \lceil \log a \rceil + 1$. The processors will be viewed as forming an $(r,\ c,\ b)$–cube, in which the jth record in list S_i, $0 \le j < 2^b$, $0 \le i < 2^a$, is located in row $\lfloor i/2^c \rfloor$ and column $i \bmod 2^c$ of level j. Call $\mathsf{CalcPrime}(a)$ to determine the value of p, the smallest prime greater than 2^c. Running time: $O(a)$.

3. **Compute balanced positions.** For each record x in S, set $\mathit{Level}(x)$ to the level of the $(r,\ c,\ b)$–cube in which x is currently located. For each record x in T, set $\mathit{TempA}(x)$ to the ID of the processor currently holding x. For $i = 0$ to d, do the following. Initially, all of the records in T are "unbalanced".

   (a) Call $\mathsf{Balance}(r,\ c,\ b,\ i,\ p)$ on the set of unbalanced records in T. Note that at most 2^{a-2^i} records in T remain unbalanced after calling $\mathsf{Balance}(r,\ c,\ b,\ i,\ p)$. Running time: $O(a)$ per iteration.

   (b) Send each record that was balanced in the previous step to its location at the start of Step 3. That is, if record x was balanced in the previous step, then x is sent to processor $\mathit{TempA}(x)$. Implementation: monotone route. Running time: $O(a)$ per iteration.

   (c) In level 0 of the cube, concentrate the records in T that have not yet been balanced to the first set of processors in level 0. In other words, if x is the jth record in level 0 that has not yet been balanced, $0 \le j < 2^a$, then x is sent to processor $(a : j,\ b : 0)$. Implementation: concentrate. Running time: $O(a)$ per iteration.

From each record x in T, copy $\mathit{SourceRow}(x)$, $\mathit{SourceColumn}(x)$, $\mathit{IntColumn}(x)$, $\mathit{Color}(x)$, and $\mathit{Class}(x)$ to the $2^b - 1$ other records in its pile. Let U_i denote $\{x \in S \mid \mathit{Class}(x) = i\}$, $0 \le i \le d$. Implementation: broadcast. Running time: $O(a)$.

Total running time: $O(a \log a)$.

4. **Separate classes.** View the $(r,\ c,\ b)$–cube as occupying rows 0 through $2^r - 1$ of an $(r+1,\ c,\ b)$–cube. Simulating the 2^{a+b+1} processors in this $(r+1,\ c,\ b)$–cube, send each class of records, U_i, to its own block of 2^{a+b-i} consecutive processors (note that $|U_i| \le 2^{a+b-i}$). To accomplish this, perform the following steps for $i = 0$ to d.

   (a) In each of the levels in parallel, concentrate the records in U_i to the first set of processors in that level. That is, if x is the jth record in U_i within level $\mathit{Level}(x)$, $0 \le j < 2^a$, then x is sent to processor $(a+1 : j,\ b : \mathit{Level}(x))$. Implementation: concentrate. Running time: $O(a)$ per iteration.

   (b) In each of the levels in parallel, the records in U_i are routed to a block of 2^{a+b-i} consecutive processors as follows. First, each processor locally calculates $e(i) = \sum_{j=0}^{i-1} 2^{a-j}$. Then, for each record x in U_i, if x is the jth record in U_i within level $\mathit{Level}(x)$, $0 \le j < 2^a$, x is sent to processor $(a+1 : e(i)+j,\ b : \mathit{Level}(x))$. Implementation: inverse concentrate. Running time: $O(a)$ per iteration.

   Total running time: $O(a \log a)$.

5. **Move to balanced positions.** The records are currently stored in an $(r+1,\ c,\ b)$–cube. View this $(r+1,\ c,\ b)$–cube as occupying columns 0 through $2^c - 1$ of an $(r+1,\ c+1,\ b)$–cube. For $0 \le i \le d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j}$. Simulating the 2^{a+b+2} processors in the $(r+1,\ c+1,\ b)$–cube, order the records according to (in decreasing order of importance) intermediate column, class, source row, and level as follows.

   (a) In parallel, send each record to its pre-balanced position. That is, send each record x to processor $(r+1 : f(\mathit{Class}(x)) + \mathit{SourceRow}(x),\ c+1 : \mathit{SourceColumn}(x),\ b : \mathit{Level}(x))$. Implementation: monotone route. Running time: $O(a)$.

   (b) In parallel, send each record to its balanced position. That is, send each record x to processor $(r+1 : f(\mathit{Class}(x)) + \mathit{SourceRow}(x),\ c+1 : \mathit{IntColumn}(x),\ b : \mathit{Level}(x))$. Implementation: two monotone routes (one for the records going to higher-numbered processors, and one for the records going to lower-numbered processors). Running time: $O(a)$.

   (c) Map the array of balanced records from row-major to column-major storage. This moves each record that was in a processor of the form $(r+1 : X,\ c+1 : Y,\ b : Z)$ to processor $(c+1 : Y,\ r+1 : X,\ b : Z)$. The records are still viewed as occupying an $(r+1,\ c+1,\ b)$–cube, but now that cube is stored in column-major order. Implementation: BPC route. Running time: $O(a)$.

(d) In each of the levels in parallel, concentrate (in the new column-major order) the records to the first set of processors in that level. In other words, if x is the jth record (in column-major order) within level $Level(x)$, $0 \leq j < 2^a$, then x is sent to processor $(a+2 : j,\ b : Level(x))$. At this point, the 2^{a+b} records are stored in the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) intermediate column, class, source row, and level. Note that each list S_i is stored in 2^b consecutive processors and any set of (at most 2^r) lists that share the same class and intermediate column numbers is stored in a set of (at most 2^{r+b}) consecutive processors. Implementation: concentrate. Running time: $O(a)$.

Total running time: $O(a)$.

6. **Sort within columns.** A recursive call will be used to sort the records that are in the same class and have the same intermediate column. Before and after this sort, the records must be rearranged. The following steps implement the sort within columns.

(a) Permute the data so that each record that was in a processor of the form $(c : W,\ r : X,\ b-r : Y,\ r : Z)$ is sent to processor $(c : W,\ b-r : Y,\ r : X,\ r : Z)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. 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, class, color, and intermediate column values. In each such subcube, save these four values in the temporary B value, temporary C value, temporary D value, and temporary E value fields, respectively, of the record in processor $(a+b-r : X,\ r : depth)$. Running time: $O(1)$.

(c) Within each subcube of 2^{2r} consecutive processors, assign new key values that order the records according to (in decreasing order of importance) intermediate column, class, and color. These new key values are needed so that the key fields do not grow larger at successively deeper levels of the recursion. The following steps create the new keys.

i. Mark the records in processors of the form $(a+b-r : X,\ r : 0)$ as "leaders". Save the current processor number of each leader in its temporary A value field. Then, within each subcube of 2^{2r} consecutive processors, sort the leaders, resolving comparisons by (in decreasing order of importance) intermediate column, class, and color. Replace the key field of each leader by the rank that it achieves in this sort. Implementation: sparse enumeration sort. Running time: $O(a)$.

ii. Within each subcube of 2^{2r} consecutive processors, undo the sort that was performed in the previous step. That is, sort the leaders according to their temporary A values. Then send each leader x to processor $TempA(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

iii. Within each subcube of 2^{2r} consecutive processors, copy the key field from each leader to the key fields of the $2^r - 1$ records in the following processors. Implementation: broadcast. Running time: $O(a)$.

(d) Within each subcube of 2^{2r} consecutive processors, call SharedKeySort$'(r,\ r,\ depth+1)$ to (stably) sort each group according to the key values assigned in the previous step.

(e) The key, class, color, and intermediate column fields corresponding to this level of the recursion have been overwritten by the recursive call. Restore these fields from the copies saved in Step 6b. The level fields corresponding to this level of the recursion have also been overwritten by the recursive call. For each record x in S that is currently stored in a processor of the form $(c : W,\ b-r : Y,\ r : X,\ r : Z)$, set $Level(x) = (b - r : Y,\ r : Z)$. Implementation: broadcast. Running time: $O(a)$.

(f) Perform odd-even bitonic merges of sorted lists of length 2^{2r}, resolving comparisons according to (in decreasing order of importance) intermediate column, class, color, and level. Running time: $O(a)$.

(g) Permute the data so that each record that is stored in a processor of the form $(c : W,\ b-r : Y,\ r : X,\ r : Z)$ is sent to processor $(c : W,\ r : X,\ b-r : Y,\ r : Z)$. At this point, the 2^{a+b} records are ordered according to (in decreasing order of importance) intermediate column, class, color, and level. Implementation: BPC route. Running time: $O(a)$.

Total running time: $O(a)$ (plus a recursive call).

7. **Move to correct rows.** View the 2^{a+b} records as occupying columns 0 through $2^{c-1}-1$ of an $(r+1,\ c+1,\ b)$–cube. The records are stored in column-major order, so if x is the jth record in level $Level(x)$, then x is stored in processor $(c+1 : \lfloor j/2^{r+1} \rfloor,\ r+1 : j \bmod 2^{r+1},\ b : Level(x))$. Simulating the 2^{a+b+2} processors of this $(r+1,\ c+1,\ b)$–cube, order the records according to (in decreasing order of importance) class, color, intermediate column, and level as follows.

   (a) Send each record to its intermediate column and to the row given by its class and color as follows. For $0 \le i \le d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j}$. Send each record x to processor $(c + 1 : IntColumn(x),\ r + 1 : f(Class(x)) + Color(x),\ b : Level(x))$. Implementation: inverse concentrate. Running time: $O(a)$.

   (b) Map the array of balanced records from column-major to row-major storage. This moves each record that was in a processor of the form $(c+1 : X,\ r+1 : Y,\ b : Z)$ to processor $(r + 1 : Y,\ c + 1 : X,\ b : Z)$. Implementation: BPC route. Running time: $O(a)$.

   (c) In each of the levels in parallel, concentrate (in the new row-major order) the records to the first set of processors in that level. In other words, if x is the jth record (in row-major order) within level $Level(x)$, $0 \le j < 2^a$, then x is sent to processor $(a + 2 : j,\ b : Level(x))$. At this point, the 2^{a+b} records are stored in the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) class, color, intermediate column, and level. Note that each list S_i is stored in 2^b consecutive processors and any set of (at most 2^{c+1}) lists that share the same class and color is stored in a set of (at most 2^{b+c+1}) consecutive processors. Implementation: concentrate. Running time: $O(a)$.

Total running time: $O(a)$.

8. **Sort within rows.** A recursive call will be used to sort the records that are in the same class and have the same color. Before and after this sort, the records must be rearranged. The following steps implement the sort within rows.

   (a) Permute the data so that each record that was in a processor of the form $(r-1: W,\ c+1: X,\ b-c-1: Y,\ c+1: Z)$ is sent to processor $(r-1: W,\ b-c-1: Y,\ c+1: X,\ c+1: Z)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. Implementation: BPC route. Running time: $O(a)$.

   (b) Note that the 2^{c+1} records residing in any subcube of the form $(a+b-c-1: X,\ c+1: *)$ share the same key and class values. In each such subcube, save these two values in the temporary B value and temporary C value fields, respectively, of the record in processor $(a+b-c-1: X,\ c+1: \mathit{depth})$. Running time: $O(1)$.

   (c) Within each subcube of 2^{2c+2} consecutive processors, assign new key values which order the records according to (in decreasing order of importance) class and key value. These new keys are needed so that the key fields do not grow larger at successively deeper levels of the recursion. The following steps create the new keys.

      i. Mark the records in processors of the form $(a+b-c-1: X,\ c+1: 0)$ as "leaders". Save the current processor number of each leader in its temporary A value field. Then, within each subcube of 2^{2c+2} consecutive processors, sort the leaders, resolving comparisons by (in decreasing order of importance) class and key value. Replace the key field of each leader by the rank that it achieves in this sort. Implementation: sparse enumeration sort. Running time: $O(a)$.

      ii. Within each subcube of 2^{2c+2} consecutive processors, undo the sort that was performed in the previous step. That is, sort the leaders according to their temporary A values. Then send each leader x to processor $\mathit{TempA}(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

      iii. Within each subcube of 2^{2c+2} consecutive processors, copy the key field from each leader to the key fields of the $2^{c+1}-1$ records in the following processors. Implementation: broadcast. Running time: $O(a)$.

   (d) Within each subcube of 2^{2c+2} consecutive processors, call $\mathsf{SharedKeySort}'(c+1,\ c+1,\ \mathit{depth}+1)$ to (stably) sort each group according to the key values assigned in the previous step.

(e) The key and class fields corresponding to this level of the recursion have been overwritten by the recursive call. Restore these fields from the copies saved in Step 8b. The level fields corresponding to this level of the recursion have also been overwritten by the recursive call. For each record x in S that is currently stored in a processor of the form $(r-1 : W,\ b-c-1 : Y,\ c+1 : X,\ c+1 : Z)$, set $Level(x) = (b-c-1 : Y,\ c+1 : Z)$. Implementation: broadcast. Running time: $O(a)$.

(f) Perform odd-even bitonic merges of sorted lists of length 2^{2c+2}, resolving comparisons according to (in decreasing order of importance) class, key value, and level. Running time: $O(a)$.

(g) Permute the data so that each record that is stored in a processor of the form $(r-1 : W,\ b-c-1 : Y,\ c+1 : X,\ c+1 : Z)$ is sent to processor $(r-1 : W,\ c+1 : X,\ b-c-1 : Y,\ c+1 : Z)$. The 2^{a+b} records are now sorted according to (in decreasing order of importance) class, key value, and level. Implementation: BPC route. Running time: $O(a)$.

Total running time: $O(a)$ (plus a recursive call).

9. **Merge classes.** At this point, each list S_i is stored in 2^b consecutive processors. Furthermore, the lists are ordered with respect to one another according to (in decreasing order of importance) class and key value. Thus, all that remains is to merge the sorted classes. This merging task may be accomplished as follows.

(a) View the records as forming an $(r,\ c,\ b)$–cube. Save the current processor number of each record in level 0 in its temporary A value field. Sort the records in level 0 according to their key values to determine their final sorted positions. Set the destination row and destination column fields of the records in level 0 to match their current positions. Implementation: sparse enumeration sort. Running time: $O(a)$.

(b) Undo the sort that was performed in the previous step. That is, sort the records in level 0 according to their temporary A values. Then send each record x in level 0 to processor $TempA(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

(c) Copy the destination row and destination column fields from each record in level 0 to the destination row and destination column fields of the $2^b - 1$ other records in its pile. This gives the final sorted position of each record. Implementation: broadcast. Running time: $O(a)$.

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

Total running time: $O(a \log a)$.

Space analysis: Each record contains fourteen fields, each of which consists of a constant number of words. There are never more than a constant number of records located at a processor. Hence, the algorithm requires only a constant number of words of memory at each 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

$$SKS(a, b, d) = SKS(r, r, d + 1) + SKS(c + 1, c + 1, d + 1) + O(a \log a),$$

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 the bitonic sorts performed in Step 1 is $O(a \log a)$. Now consider the total amount of time spent on recursive calls at depths less than the maximum depth. Lemma 2.1 implies that this time is bounded by $O(a \log^2 a)$. Therefore, $SKS(a, b, 0) = O(a \log^2 a)$.

6.2 Prime Calculation

This subsection defines the subroutine CalcPrime(a).

Input/Processors: A subcube of 2^a processors, each of which holds the value a, and where a is an integer strictly greater than 2.

Output: The value p stored in each of the 2^a processors, where p is the smallest prime number greater than $2^{\lceil a/2 \rceil}$.

Running time: $O(a)$.

Algorithm CalcPrime(a)

1. Each processor calculates $c = \lceil a/2 \rceil$ locally. Note that the desired prime p is in the range $2^c < p < 2^{c+1}$ [2]. Running time: $O(1)$.
2. Simulate a subcube of 2^{2c+1} processors, viewed as forming a two-dimensional array with 2^c rows and 2^{c+1} columns. Calculate p as follows:
   (a) The processor in row i, $0 \le i < 2^c$ and column j, $0 \le j < 2^{c+1}$, determines whether or not $2^c + i$ is divisible by $j + 2$. Running time: $O(1)$.
   (b) The processor in row i, $0 \le i < 2^c$ and column 0 determines whether or not $2^c + i$ is prime. Implementation: suffix operation. Running time: $O(a)$.
   (c) The processor in row 0 and column 0 determines p. Implementation: suffix operation. Running time: $O(a)$.
3. The value of p is broadcast to all of the 2^a processors. Implementation: broadcast. Running time: $O(a)$.

6.3 Color Balancing

This subsection defines the subroutine Balance(), which is the primary subroutine for the planning stage. The input to Balance() is a three-dimensional array of processors, one level of which contains records. The records are first sorted with a sparse enumeration sort. They are assigned colors according to their row number when they are in sorted order. Then they are returned to their original positions.

The remainder of the Balance() subroutine is devoted to determining a permutation of the records that can be implemented efficiently and that distributes the records with matching colors to different columns. A copy of the entire set of input records is sent to each level of the three-dimensional array. Then each level applies a different permutation to its copy of the records, and the permutation that most successfully balances the colors between the columns is selected. The selected permutation may not balance the colors perfectly, so it is possible that several records with the same color are sent to the same column. Whenever this occurs, only one of the records with the given color that is sent to the given column is designated as a "balanced" record. The other records with the same color that are sent to the same column are designated as "unbalanced".

The planning stage performs a series of calls to Balance(). The records that are balanced by a single call to Balance() are considered to form one "class". The records that are unbalanced following a call to Balance() form the input for the next call to Balance(). As the input to Balance() becomes increasingly sparse, it becomes more efficient at balancing the records. As a result, only a small number of calls to Balance() are required.

The following lemma will be useful in proving the correctness of the subroutine Balance().

Lemma 6.1 *Let i, i', j, j', and p be nonnegative integers where p is a prime, $i < p$, $i' < p$, and $i \neq i'$. For any integers x, y, and z, let $f(x, y, z) = y + xz \bmod p$. Then there exists at most one integer h in the range $0 \leq h < p$ such that $f(i, j, h) = f(i', j', h)$.*

Proof: Assume for the sake of contradiction that there exist two distinct integers h_1 and h_2 such that $0 \leq h_1 < p$, $0 \leq h_2 < p$, $f(i, j, h_1) = f(i', j', h_1)$, and $f(i, j, h_2) = f(i', j', h_2)$. Then

$$\begin{aligned} j + ih_1 &\equiv j' + i'h_1 \pmod{p}, \text{ and} \\ h_1(i - i') &\equiv j' - j \pmod{p}. \end{aligned}$$

By similar reasoning,

$$h_2(i - i') \equiv j' - j \pmod{p},$$

and so

$$h_1(i - i') \equiv h_2(i - i') \pmod{p}.$$

Because h_1, h_2, and $i - i'$ are all integers in the range 1 to $p - 1$, and p is prime, it follows that $h_1 = h_2$, which is a contradiction. □

A detailed description of the subroutine Balance(r, c, b, k, p) will now be given.

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

Output: A subset U of T such that $|T \setminus U| \leq 2^{r+c-2s-1}$, called the "balanced records", and for each record x in U an assignment to the fields $\mathit{SourceRow}(x)$, $\mathit{SourceColumn}(x)$, $\mathit{IntColumn}(x)$, $\mathit{Color}(x)$, and $\mathit{Class}(x)$. These fields will be referred to as the source row, source column, intermediate column, color, and class, respectively, of record x. The fields $\mathit{SourceRow}(x)$ and $\mathit{SourceColumn}(x)$ are set to the row and column positions of x when Balance() was called. The field $\mathit{IntColumn}(x)$ is in the range $0 \leq \mathit{IntColumn}(x) < p$, the field $\mathit{Color}(x) = \lfloor \mathit{Rank}(T,\ x)/2^c \rfloor$, and the field $\mathit{Class}(x) = k$. For any balanced records x and y,

$$\begin{aligned} \mathit{SourceRow}(x) = \mathit{SourceRow}(y) &\ \Rightarrow\ \mathit{IntColumn}(x) \neq \mathit{IntColumn}(y), \text{ and} \\ \mathit{Color}(x) = \mathit{Color}(y) &\ \Rightarrow\ \mathit{IntColumn}(x) \neq \mathit{IntColumn}(y). \end{aligned}$$

The final (and arguably the most important) output condition is that when the 2^{r+c+b} processors simulate an $(r,\ c+1,\ b)$–cube, two monotone routes that can be implemented entirely within the rows of level 0 are sufficient to move the balanced records from their source columns to their intermediate columns.

Running time: $O(bc/(b-c))$.

Example: The input is a cube of n processors arranged in $n^{1/4}$ rows, $n^{1/4}$ columns, and $n^{1/2}$ levels with $n^{5/12}$ records in the first $n^{5/12}$ processors 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$(r,\ c,\ b,\ k,\ p)$

1. For each record x in T, set the $\mathit{SourceRow}(x)$ and $\mathit{SourceColumn}(x)$ fields to the row and column positions of x, respectively. Running time: $O(1)$.

2. Assign colors to the records in T. This is accomplished by the following sequence of steps.

   (a) Sort the records in T according to their key values. Implementation: sparse enumeration sort. Running time: $O(c)$.

   (b) For each record x in T, calculate $\mathit{Rank}(T,\ x)$. Implementation: prefix operation. Running time: $O(c)$.

   (c) Calculate $|T|$, and broadcast this value to every processor in level 0. Implementation: suffix operation, broadcast. Running time: $O(c)$.

   (d) For each record x in T, set the field $\mathit{Color}(x)$ to $\lfloor \mathit{Rank}(T,\ x)/2^c \rfloor$. Running time: $O(1)$.

(e) Return the records in T to their original positions by sorting them according to their source row and source column fields. Implementation: sparse enumeration sort. Running time: $O(c)$.

Total running time: $O(c)$.

3. If $|T| \leq 2^r$ then every record in T has been assigned a different color. In this case, let $U = T$, and for each record x in T set $IntColumn(x) = SourceColumn(x)$, $Class(x) = k$, and return. Otherwise, go to Step 4. Running time: $O(1)$.

4. Copy each record in row i, $0 \leq i < 2^r$, and column j, $0 \leq j < 2^c$, of level 0 to row i and column j of each of the first p levels. Let $T_{i,j}$ denote the pile of p records in row i and column j. Implementation: broadcast. Running time: $O(c)$.

5. Simulating an $(r,\ c+1,\ b)$–cube, permute the copies of the records in the first p levels of the cube as follows. In level h, $0 \leq h < p$, move the record (if any) in row i and column j to row i and column $(j + ih) \bmod p$, $0 \leq i < 2^r$, $0 \leq j < 2^c$. Implementation: two monotone routes (one for the records going to higher-numbered processors, and one for records going to lower-numbered processors). Running time: $O(c)$.

6. Define a *collision* to be the mapping of two records in the same level and with the same color to the same column. Note that two records that begin in the same level and row cannot collide with one another. Hence, Lemma 6.1 implies that for any pair of piles of the same color, $T_{i,j}$ and $T_{i',j'}$, there is a total of at most one collision between the records in $T_{i,j}$ and the records in $T_{i',j'}$. Because there are at most 2^{c-s} piles with each color, the total number of collisions is less than $2^{2c-2s-1}2^r = 2^{r+2c-2s-1}$, and one of the first p levels must have fewer than $2^{r+2c-2s-1}/p < 2^{r+c-2s-1}$ collisions. Simulating an $(r,\ c+1,\ b)$–cube, the following steps determine which of the the first p levels contains the smallest number of collisions.

   (a) Sort the records in each column of each level according to their colors. Implementation: sparse enumeration sort. Running time: $O(bc/(b-c))$.

   (b) Calculate the position of each record within the set of records in the same level and column that have the same color. Implementation: segmented prefix operation. Running time: $O(r)$.

   (c) In each set of records in the same level and column that have the same color, designate the last record as "leader". Implementation: segmented suffix operation. Running time: $O(r)$.

   (d) For each leader, locally calculate the number of collisions involving records of its level, column, and color. Then sum these values to calculate the total number of collisions within each level. Implementation: prefix operation. Running time: $O(c)$.

   (e) Determine the level h, $0 \leq h < p$, that contains the smallest number of collisions (break ties arbitrarily) and broadcast the result to every processor in the first p levels. Implementation: prefix operation, broadcast. Running time: $O(c)$.

(f) Return each record x to the position that it had following Step 4 by sorting the records in each level according to their source row and source column fields and then routing them to the row and column given by their source row and source column fields. Implementation: sparse enumeration sort, monotone route. Running time: $O(bc/(b-c))$.

Total running time: $O(bc/(b-c))$.

7. Let level h, $0 \leq h < p$, be the level containing the smallest number of collisions. For each record x in level h that was designated a leader in Step 6c, set

$$IntColumn(x) = (SourceColumn(x) + SourceRow(x) \cdot h) \bmod p$$

and set $Class(x) = k$. These leaders form the set of balanced records U. Note that each collision prevents at most one record from becoming balanced, so that at most $2^{r+c-2s-1}$ records in level h remain unbalanced. Finally, send each record x in level h back to level 0. Implementation: monotone route. Running time: $O(c)$.

7 Improved Shared Key Sorting

This section defines a more efficient shared key sorting algorithm, SharedKeySort$''(a, b)$, which runs in $O(a \log a)$ time. This algorithm is similar to the $O(a \log^2 a)$ time algorithm SharedKeySort(a, b) given in Section 6, but it contains several modifications that yield the improved performance. Before examining these modifications, it is helpful to identify the inefficiencies in the slower algorithm, SharedKeySort(). The algorithm SharedKeySort() performs a pair of recursive calls with parameters approximately half as large as in the original call. Of the 9 major steps in SharedKeySort(), only Steps 1, 3, 4, and 9 require more than $O(a)$ time at any single level of the recursion. Step 1, the base case, requires $O(a^2)$ time only at the deepest level of the recursion. As a result, Step 1 contributes just $O(a \log a)$ time to the overall running time. However, Step 3, which computes the balanced positions, Step 4, which separates the classes, and Step 9, which merges the classes, each require $O(a \log a)$ time at every level of the recursion other than the deepest. As a result, each of these steps contributes $O(a \log^2 a)$ time to the overall running time.

Let us first consider how Steps 4 and 9, which separate and merge the classes, can be improved. In SharedKeySort(), the records are divided into $O(\log a)$ different classes by the subroutine Balance(). Step 4 separates these classes by performing a monotone route (consisting of a concentrate and an inverse concentrate) for each of the $O(\log a)$ classes. Each monotone route takes $O(a)$ time and the $O(\log a)$ monotone routes are performed sequentially, so Step 4 requires $O(a \log a)$ time at each level of the recursion. In the improved algorithm, SharedKeySort$''$(), the records are again divided into $O(\log a)$ different classes by balancing subroutines. However, a new balancing subroutine, Balance$'$(), is used that balances all but a $\Theta(1/\log a)$ fraction of the records the first time that it is called. Step 4 of SharedKeySort$''$() separates the records in this large, first class from the remaining records. It turns out that the remaining classes can then be separated from one another in parallel, because these classes contain so few records. The new balancing routine Balance$'$() is capable of balancing almost all of the records in a single call, but it guarantees a weaker form of balancing than is provided by Balance(). As a result, the recursive calls in SharedKeySort″() must be slightly larger than those used in SharedKeySort(). Fortunately, this increase in the size of the recursive calls does not affect the asymptotic performance by more than a constant factor.

A very different approach is used to improve the performance of Step 3, which calculates the balanced positions. Recall that the algorithm SharedKeySort() can be divided into two parts, a planning stage and a routing stage. The planning stage determines the sorted position of each record, and calculates a path from each record to its sorted position. The routing stage then sends each record to its sorted position along the path calculated in the planning stage. In SharedKeySort(), each recursive call consists of both planning and routing operations. In contrast, SharedKeySort″() separates the planning operations from the routing operations. The algorithm SharedKeySort″() first calls the subroutine PlanRoute() to perform the planning operations for all levels of the recursion, and it then calls the subroutine DoRoute() to perform the routing operations for all levels of the recursion. Thus the planning is completed for all levels of the recursion before any records are moved. The planning information needed by DoRoute() is provided by records, called *routing records*, that are created by PlanRoute().

The separation of the planning and routing stages permits a more efficient algorithm. Specifically, the fact that the input to PlanRoute() is so sparse allows its recursive calls to be performed in parallel in different subcubes. The calculation of the balanced positions occurs only in the subroutine PlanRoute(). Thus, although it takes $O(a \log a)$ time to calculate the balanced positions at the highest level of the recursion, the calls at each of the remaining levels of the recursion are performed in parallel and contribute only a total of $O(a \log a)$ time to the overall running time.

The remainder of this section is devoted to a top-down presentation of the improved shared key sorting algorithm. Section 7.1 presents the top-level routine, SharedKeySort″(), which calls subroutines PlanRoute() and DoRoute(). The subroutines PlanRoute() and DoRoute() are described in Sections 7.2 and 7.3, respectively. The recursive PlanRoute() routine calls subroutines CalcPrime(), Balance(), and Balance′(), which are defined in Sections 6.2, 6.3, and 7.4, respectively. The recursive DoRoute() routine calls subroutines CalcPrime(), SeparateClasses(), and MergeClasses(), defined in Sections 6.2, 7.5, and 7.6, respectively.

7.1 The Top-Level Routine

This subsection provides a faster implementation of the SharedKeySort() operation defined in Section 6.1. The running time of the following algorithm is $O(a \log a)$, and it requires only constant storage at each processor.

Algorithm SharedKeySort(a, b)

1. Let S denote the set of input records. Call PlanRoute(a, b) on the set of all level 0 records in S. Running time: $O(a \log a)$.

2. Call $\mathsf{DoRoute}(a,\ b)$ on the set S and the routing records created in the previous step. Running time: $O(a \log a)$.

7.2 Plan Route

This subsection defines the algorithm $\mathsf{PlanRoute}(a,\ b)$.

Input/Processors: A subcube of 2^{a+b} processors holding a set T of 2^a records, called *data records*, where a and b are positive integers with $b > \lceil a/2 \rceil + 1$ and $b - \lceil a/2 \rceil - 1 = \Theta(a)$. The records in T are stored, one per processor, in the first 2^a processors. Each record x in T has eight fields, namely: $Key(x)$, $SourceRow(x)$, $SourceColumn(x)$, $Class(x)$, $IntColumn(x)$, $Color(x)$, $Count(x)$, and $TempA(x)$. These fields will be called the key, source row, source column, class, intermediate column, color, count, and temporary A value of x, respectively. Only the key fields contain relevant data at the time $\mathsf{PlanRoute}(a,\ b)$ is called.

Output: A set of routing records corresponding to the records in T. Each routing record has the same fields as a data record. The source row, source column, class, intermediate column, color, and count fields of the routing records are set by calls to $\mathsf{Balance}'()$ and $\mathsf{Balance}()$. At most one routing record is output at each processor, and the output is arranged in such a way that a subsequent call to $\mathsf{DoRoute}(a,\ b)$ can efficiently obtain the routing records that it requires at every point in the recursion. The algorithm $\mathsf{DoRoute}()$, which has precisely the same recursive structure as $\mathsf{PlanRoute}()$, will be defined in Section 7.3.

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

Example: Given $n^{1/2}$ data records stored in n processors, this algorithm creates $O(\log\log n)$ routing records for each of the $n^{1/2}$ data records in $O(\log n \log\log n)$ time.

The operation $\mathsf{PlanRoute}(a,\ b)$ is performed by calling $\mathsf{PlanRoute}'(a,\ b,\ 1)$, defined below. The third parameter identifies which recursive call is being performed.

Algorithm $\mathsf{PlanRoute}'(a,\ b,\ num)$

1. **Base Case.** If $\lfloor a/2 \rfloor \le \log num$, return without creating any routing records. Running time: $O(1)$.

2. **Compute dependent parameters.** Each processor locally calculates $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$, $d = \lceil \log a \rceil + 1$, and $t = \lceil \log d \rceil$. The processors will be viewed as forming an $(r,\ c,\ b)$–cube, in which the ith record in T, $0 \le i < 2^a$, is located in row $\lfloor i/2^c \rfloor$ and column $i \bmod 2^c$ of level 0. Call $\mathsf{CalcPrime}(a)$ to determine the value of p, the smallest prime greater than 2^c. Running time: $O(a)$.

3. **Compute balanced positions.** For each record x in T, set $TempA(x)$ to the number of the processor currently holding x. Call $\mathsf{Balance}'(r,\ c,\ b,\ t,\ 0,\ p)$ on the records in T. This balances all but 2^{a-t-1} records in T. For $i = 1$ to d, do the following.

   (a) In level 0 of the cube, concentrate the records in T that have not yet been balanced to the first set of processors in level 0. That is, if x is the jth record in level 0 that has not yet been balanced, $0 \leq j < 2^a$, then x is sent to processor $(a : j,\ b : 0)$. Implementation: concentrate. Running time: $O(a)$ per iteration.

(b) Call Balance$(r,\ c,\ b,\ i,\ p)$ on the set of unbalanced records in T. Note that at most $2^{a-2^i t-2^i-1}$ records in T remain unbalanced after calling Balance$(r,\ c,\ b,\ i,\ p)$. Running time: $O(a)$ per iteration.

(c) Set the count field of each record that was balanced in the previous step to 0. Send each record that was balanced in the previous step to its location at the start of Step 3. That is, if record x was balanced in the previous step, then x is sent to processor $TempA(x)$. Implementation: monotone route. Running time: $O(a)$ per iteration.

From each record x in T, values have now been assigned to the fields $SourceRow(x)$, $SourceColumn(x)$, $IntColumn(x)$, $Color(x)$, $Count(x)$, and $Class(x)$. Create a copy of each record in T at its current location and with the same source row, source column, intermediate column, color, count, and class values. These copies will form the routing records for the current recursive call. Send each routing record x to the processor in level $num - 1$ of the pile containing x. Let U_i denote the set of records $\{x \in T \mid Class(x) = i\}$, $0 \leq i \leq d$. Implementation: monotone route. Running time: $O(a)$.

Total running time: $O(a \log a)$.

4. **Separate classes.** The records in T are currently stored in level 0 of an $(r,\ c,\ b)$–cube. View this $(r,\ c,\ b)$–cube as occupying rows 0 through $2^r - 1$ of an $(r+1,\ c,\ b)$–cube. Simulating the 2^{a+b+1} processors in this $(r+1,\ c,\ b)$–cube, send each class of records, U_i, to its own block of 2^{a-i} consecutive processors within level 0 (note that $|U_i| \leq 2^{a-i}$). To accomplish this, for $i = 0$ to d, do the following.

   (a) Concentrate the records in U_i to the first set of processors in level 0. That is, if x is the jth record in U_i, $0 \leq j < 2^a$, then x is sent to processor $(a+1 : j,\ b : 0)$. Implementation: concentrate. Running time: $O(a)$ per iteration.

   (b) Within level 0, the records in U_i are routed to a block of 2^{a-i} consecutive processors as follows. First, each processor locally calculates $e(i) = \sum_{j=0}^{i-1} 2^{a-j}$. Then, for each record x in U_i, if x is the jth record in U_i, $0 \leq j < 2^a$, x is sent to processor $(a+1 : e(i)+j,\ b : 0)$. Implementation: inverse concentrate. Running time: $O(a)$ per iteration.

   Total running time: $O(a \log a)$.

5. **Move to balanced positions.** The records in T are currently stored in level 0 of an $(r+1,\ c,\ b)$–cube. View this $(r+1,\ c,\ b)$–cube as occupying columns 0 through $2^c - 1$ of an $(r+1,\ c+1,\ b)$–cube. For $0 \leq i \leq d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j}$. Simulating the 2^{a+b+2} processors in the $(r+1,\ c+1,\ b)$–cube, order the records within level 0 according to (in decreasing order of importance) intermediate column, class, and source row as follows.

(a) In parallel, send each record to its pre-balanced position. That is, send each record x in T to processor $(r{+}1 : f(\mathit{Class}(x)){+}\mathit{SourceRow}(x),\ c{+}1 : \mathit{SourceColumn}(x),\ b : 0)$. Implementation: monotone route. Running time: $O(a)$.

(b) In parallel, send each record to its balanced position. That is, send each record x in T to processor $(r{+}1 : f(\mathit{Class}(x)){+}\mathit{SourceRow}(x),\ c{+}1 : \mathit{IntColumn}(x),\ b : 0)$. Implementation: two monotone routes (one for the records going to higher-numbered processors, and one for the records going to lower-numbered processors). Running time: $O(a)$.

(c) Map the array of balanced records from row-major to column-major storage. This moves each record that was in a processor of the form $(r+1 : X,\ c+1 : Y,\ b : 0)$ to processor $(c+1 : Y,\ r+1 : X,\ b : 0)$. The records are still viewed as occupying an $(r+1,\ c+1,\ b)$–cube, but now that cube is stored in column-major order. Implementation: BPC route. Running time: $O(a)$.

(d) Concentrate (in the new column-major order) the records in T to the first set of processors in level 0. That is, if x is the jth record (in column-major order) in T, $0 \leq j < 2^a$, then x is sent to processor $(a+2 : j,\ b : 0)$. At this point, the 2^a records are stored within the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) intermediate column, class, and source row. Implementation: concentrate. Running time: $O(a)$.

Total running time: $O(a)$.

6. **Sort within columns.** The subroutine DoRoute() will rearrange the records and perform a recursive sort at this step. A similar recursive call will be used by PlanRoute(), but the recursive call will be delayed until the last step. However, in order to prepare for the other recursive call that must be made by PlanRoute() (the one that performs the sorts within the rows), it is necessary to use a sparse enumeration sort to first sort within the columns.

   (a) Permute the data so that each record that was in a processor of the form $(c : W,\ r : X,\ b-r : 0,\ r : 0)$ is sent to processor $(c : W,\ b-r : 0,\ r : X,\ r : 0)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. Implementation: BPC route. Running time: $O(a)$.

   (b) Create a copy of each record in T at its current location. Let T' be the set of 2^a records that are created. The records in T' will form the input to one of the recursive calls in the last step of PlanRoute(). Running time: $O(1)$.

   (c) Within each subcube of 2^{2r} processors of the form $(c : W,\ b-r : 0,\ 2r : *)$, assign new key values to the records in T' that order them according to (in decreasing order of importance) intermediate column, class, color, and count. The following steps create the new keys.

      i. Save the current processor number of each record in T' in its temporary A value field. Then, within each subcube of 2^{2r} processors of the form $(c :$

$W,\ b-r:0,\ 2r:*)$, sort the records in T', resolving comparisons by (in decreasing order of importance) intermediate column, class, color, and count. Replace the key field of each record in T' by the rank that it achieves in this sort. Implementation: sparse enumeration sort. Running time: $O(a)$.

ii. Within each subcube of 2^{2r} processors of the form $(c:W,\ b-r:0,\ 2r:*)$, undo the sort that was performed in the previous step. That is, sort the records in T' according to their temporary A values. Then send each record x in T' to processor $\mathit{TempA}(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

(d) Concentrate the records in T to the first 2^a processors. Sort the records in T according to (in decreasing order of importance) intermediate column, class, color, and count. Implementation: concentrate, sparse enumeration sort. Running time: $O(a)$.

Total running time: $O(a)$.

7. **Move to correct rows.** View the 2^a records in T as occupying columns 0 through $2^{c-3}-1$ of level 0 of an $(r+3,\ c+1,\ b)$–cube. The records are stored in column-major order, so if x is the jth record in T, then x is stored in processor $(c+1:\lfloor j/2^{r+3}\rfloor,\ r+3:j \bmod 2^{r+3},\ b:0)$. Simulating the 2^{a+b+4} processors of this $(r+3,\ c+1,\ b)$–cube, order the records according to (in decreasing order of importance) class, color, count, and intermediate column.

   (a) Send each record to its intermediate column and to the row given by its class, color, and count, as follows. For $0 \le i \le d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j+2}$. For $0 \le i < 2^{r-t}$, let $g(i) = i2^{t+2}$. Send each record x to processor $(c+1:\mathit{IntColumn}(x),\ r+3:f(\mathit{Class}(x)) + g(\mathit{Color}(x)) + \mathit{Count}(x),\ b:0)$. Implementation: inverse concentrate. Running time: $O(a)$.

   (b) Map the array of balanced records from column-major to row-major storage. This moves each record that was in a processor of the form $(c+1:X,\ r+3:Y,\ b:0)$ to processor $(r+3:Y,\ c+1:X,\ b:0)$. Implementation: BPC route. Running time: $O(a)$.

   (c) Concentrate (in the new row-major order) the records in T to the first set of processors in level 0. That is, if x is the jth record (in row-major order) in T, $0 \le j < 2^a$, then x is sent to processor $(a+4:j,\ b:0)$. At this point, the 2^a records are stored within the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) class, color, count, and intermediate column. Implementation: concentrate. Running time: $O(a)$.

   Total running time: $O(a)$.

8. **Sort within rows.** Rearrange the records to prepare for the sort within rows. Then perform both recursive calls in parallel.

(a) Permute the data so that each record that was in a processor of the form $(r-t-3 : W,\ c+t+3 : X,\ b-c-t-3 : 0,\ c+t+3 : 0)$ is sent to processor $(r-t-3 : W,\ b-c-t-3 : 1,\ c+t+3 : X,\ c+t+3 : 0)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. Implementation: BPC route. Running time: $O(a)$.

(b) Create a copy of each record in T at its current location. Let T'' be the set of 2^a records that are created. The records in T'' will form the input to one of the recursive calls. Running time: $O(1)$.

(c) Within each subcube of $2^{2c+2t+6}$ processors of the form $(r-t-3 : W,\ b-c-t-3 : 1,\ 2c+2t+6 : *)$, assign new key values to the records in T'' which order them according to (in decreasing order of importance) class and key value. The following steps create the new keys.

   i. Save the current processor number of each record in T'' in its temporary A value field. Then within each subcube of $2^{2c+2t+6}$ processors of the form $(r-t-3 : W,\ b-c-t-3 : 1,\ 2c+2t+6 : *)$, sort the records in T'', resolving comparisons by (in decreasing order of importance) class and key value. Replace the key field of each record in T'' by the rank that it achieves in this sort. Implementation: sparse enumeration sort. Running time: $O(a)$.

   ii. Within each subcube of $2^{2c+2t+6}$ processors the form $(r-t-3 : W,\ b-c-t-3 : 1,\ 2c+2t+6 : *)$, undo the sort that was performed in the previous step. That is, sort the records in T'' according to their temporary A values. Then send each record x in T'' to processor $TempA(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

(d) In parallel, in subcubes of 2^{2r} processors of the form $(c : W,\ b-r : 0,\ 2r : *)$ call $\mathsf{PlanRoute}'(r,\ r,\ 2num)$ on the set T', and in subcubes of $2^{2c+2t+6}$ processors of the form $(r-t-3 : W,\ b-c-t-3 : 1,\ 2c+2t+6 : *)$ call $\mathsf{PlanRoute}'(c+t+3,\ c+t+3,\ 2num+1)$ on the set T''.

Total running time: $O(a)$ (plus parallel recursive calls).

Space analysis: Each record contains a constant number of fields, each of which consists of a constant number of words. During the course of the algorithm, only a constant number of data records are stored at a processor. Furthermore, the routing records are output at distinct levels. Finally, because tail recursion is used, the data records in the set T can be discarded before performing the recursive calls. As a result, the algorithm requires only constant storage per processor.

Time analysis: Let $PR(a,b,n)$ denote the running time of $\mathsf{PlanRoute}'(a,\ b,\ n)$. If $\lfloor a/2 \rfloor \le \log n$ then $PR(a,b,n) = O(1)$, and if $a > \log n$ then

$$PR(a,b,n) = \max(PR(r,r,2n), PR(c+t+3, c+t+3, 2n+1)) + O(a \log a),$$

where $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$, and $t = \lceil \log \lceil 1 + \log a \rceil \rceil$. This recurrence solves to give $PR(a,b,1) = O(a \log a)$.

7.3 Do Route

This subsection defines the algorithm $\mathsf{DoRoute}(a,\ b)$.

Input: A set S of 2^{a+b} data records and the set of routing records created by a call to $\mathsf{PlanRoute}(a,\ b)$. The data records are organized as 2^a lists of 2^b records each, where a and b are positive integers with $b > \lceil a/2 \rceil + 1$ and $b - \lceil a/2 \rceil - 1 = \Theta(a)$, and the 2^b records in each list have the same key field. Let S_i denote the ith list of 2^b data records, $0 \le i < 2^a$, and let T denote the set of 2^a records obtained by taking the first record from each list S_i. The records of S are stored in a subcube of dimension $a + b$; the jth element of list S_i is stored in processor $(a : i,\ b : j)$, $0 \le i < 2^a$, $0 \le j < 2^b$. Each record x in S has fifteen fields, namely: $Key(x)$, $Level(x)$, $SourceRow(x)$, $SourceColumn(x)$, $Class(x)$, $IntColumn(x)$, $Color(x)$, $Count(x)$, $DestRow(x)$, $DestColumn(x)$, $TempA(x)$, $TempB(x)$, $TempC(x)$, $TempD(x)$, and $TempE(x)$. These fields will be called the key, level, source row, source column, class, intermediate column, color, count, destination row, destination column, temporary A value, temporary B value, temporary C value, temporary D value, and temporary E value of x, respectively. The routing records have key, source row, source column, class, intermediate column, color, count, and temporary A value fields.

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 sent to processor $(a : Rank(T,\ x),\ b : j)$, $0 \le i < 2^a$, $0 \le j < 2^b$.

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

Example: Given n data records organized as $n^{1/2}$ lists of length $n^{1/2}$, in which the records belonging to the same list have the same key value, and given the routing records created by a corresponding call to $\mathsf{PlanRoute}()$, this algorithm uses n processors to sort the n data records in $O(\log n \log\log n)$ time.

The operation $\mathsf{DoRoute}(a,\ b)$ is performed by calling $\mathsf{DoRoute}'(a,\ b,\ 1)$, defined below. The third parameter identifies which recursive call is being performed and is used to locate the routing records provided by $\mathsf{PlanRoute}()$.

Algorithm $\mathsf{DoRoute}'(a,\ b,\ num)$

1. **Base case.** If $\lfloor a/2 \rfloor \le \log num$, then perform a bitonic sort of the 2^{a+b} records in S, and return. Running time: $O(a^2)$ if $\lfloor a/2 \rfloor \le \log num$, $O(1)$ otherwise.

2. **Compute dependent parameters.** Each processor locally calculates $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$, $d = \lceil \log a \rceil + 1$, and $t = \lceil \log d \rceil$. The processors will be viewed as forming an $(r,\ c,\ b)$–cube in which the jth record in list S_i, $0 \le j < 2^b$, $0 \le i < 2^a$, is located in row $\lfloor i/2^c \rfloor$ and column $i \bmod 2^c$ of level j. Call $\mathsf{CalcPrime}(a)$ to determine the value of p, the smallest prime greater than 2^c. Running time: $O(a)$.

3. **Compute balanced positions.** For each record x in S, set $Level(x)$ to the level of the $(r,\ c,\ b)$–cube in which x is currently located. Within each pile, copy the routing record stored in the processor at level num to every other processor in the pile. Set the source row, source column, class, intermediate column, color, and count fields of each

data record to match those of the routing record in the same processor. Let U_i denote the set of records $\{x \in S \mid \mathit{Class}(x) = i\}$, $0 \le i \le d$. Implementation: broadcast. Running time: $O(a)$.

4. **Separate classes.** View the $(r,\ c,\ b)$–cube as occupying rows 0 through $2^r - 1$ of an $(r+1,\ c,\ b)$–cube. Simulating the 2^{a+b+1} processors in this $(r+1,\ c,\ b)$–cube, send each class of records, U_i, to its own block of 2^{a+b-i} consecutive processors (note that $|U_i| \le 2^{a+b-i}$). To accomplish this, do the following.

   (a) In each of the levels in parallel, concentrate the records in U_0 to the first set of processors in that level. That is, if x is the ith record of U_0 in level $\mathit{Level}(x)$, $0 \le i < 2^a$, then x is sent to processor $(a+1 : i,\ b : \mathit{Level}(x))$. Implementation: concentrate. Running time: $O(a)$.

   (b) Separate the classes $U_1, \ldots, U_d$ from one another by calling SeparateClasses$(a + b + 1,\ t)$ on the records in $S \setminus U_0$. Running time: $O(a)$.

   (c) For each record x in $S \setminus U_0$, calculate the rank of x in the set of records with the same level and class. Implementation: segmented prefix operation. Running time: $O(a)$.

   (d) In each of the levels in parallel, route the records in each class U_i to a block of 2^{a+b-i} consecutive processors as follows. For $1 \le i \le d$, let $e(i) = \sum_{j=0}^{i-1} 2^{a-j}$. Then, for each record x in $S \setminus U_0$, if x is the jth record of its level and class, $0 \le j < 2^a$, route x to processor $(a+1 : e(\mathit{Class}(x))+j,\ b : \mathit{Level}(x))$. Implementation: inverse concentrate. Running time: $O(a)$.

   Total running time: $O(a)$.

5. **Move to balanced positions.** The records are currently stored in an $(r+1,\ c,\ b)$–cube. View this $(r+1,\ c,\ b)$–cube as occupying columns 0 through $2^c - 1$ of an $(r+1,\ c+1,\ b)$–cube. For $0 \le i \le d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j}$. Simulating the 2^{a+b+2} processors in the $(r+1,\ c+1,\ b)$–cube, order the records in S according to (in decreasing order of importance) intermediate column, class, source row, and level as follows.

   (a) In parallel, send each record to its pre-balanced position. That is, send each record x in S to processor $(r+1 : f(\mathit{Class}(x)) + \mathit{SourceRow}(x),\ c+1 : \mathit{SourceColumn}(x),\ b : \mathit{Level}(x))$. Implementation: monotone route. Running time: $O(a)$.

   (b) In parallel, send each record to its balanced position. That is, send each record x to processor $(r+1 : f(\mathit{Class}(x)) + \mathit{SourceRow}(x),\ c+1 : \mathit{IntColumn}(x),\ b : \mathit{Level}(x))$. Implementation: two monotone routes (one for the records going to higher-numbered processors, and one for the records going to lower-numbered processors). Running time: $O(a)$.

   (c) Map the array of balanced records from row-major to column-major storage. This moves each record that was in a processor of the form $(r+1 : X,\ c+1 : Y,\ b : Z)$ to processor $(c+1 : Y,\ r+1 : X,\ b : Z)$. The records are still viewed as occupying an $(r+1,\ c+1,\ b)$–cube, but now that cube is stored in column-major order. Implementation: BPC route. Running time: $O(a)$.

(d) In each of the levels in parallel, concentrate (in the new column-major order) the records to the first set of processors in that level. In other words, if x is the jth record (in column-major order) within level $Level(x)$, $0 \le j < 2^a$, then x is sent to processor $(a+2 : j,\ b : Level(x))$. At this point, the 2^{a+b} records are stored in the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) intermediate column, class, source row, and level. Note that each list S_i is stored in 2^b consecutive processors and any set of (at most 2^r) lists that share the same class and intermediate column numbers is stored in a set of (at most 2^{r+b}) consecutive processors. Implementation: concentrate. Running time: $O(a)$.

Total running time: $O(a)$.

6. **Sort within columns.** A recursive call will be used to sort the records in S that are in the same class and have the same intermediate column. Before and after this sort, the records must be rearranged. The following steps implement the sort within columns.

(a) Permute the data so that each record that was in a processor of the form $(c : W,\ r : X,\ b-r : Y,\ r : Z)$ is sent to processor $(c : W,\ b-r : Y,\ r : X,\ r : Z)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. 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 class, color, count, and intermediate column values. In each such subcube, save these four values in the temporary B value, temporary C value, temporary D value, and temporary E value fields, respectively, of a new record that is placed in processor $(a+b-r : X,\ r : num)$. Running time: $O(1)$.

(c) Within each subcube of 2^{2r} consecutive processors, call $\mathsf{DoRoute}'(r,\ r,\ 2num)$ to (stably) sort each group according to the routing records created by $\mathsf{PlanRoute}()$. This sorts the records in each subcube of 2^{2r} consecutive processors according to (in decreasing order of importance) intermediate column, class, color, count, and level.

(d) The class, color, count, and intermediate column fields corresponding to this level of the recursion have been overwritten by the recursive call. Restore these fields from the copies saved in Step 6b. The level fields corresponding to this level of the recursion have also been overwritten by the recursive call. For each record x in S that is currently stored in a processor of the form $(c : W,\ b-r : Y,\ r : X,\ r : Z)$, set $Level(x) = (b-r : Y,\ r : Z)$. Implementation: broadcast. Running time: $O(a)$.

(e) Perform odd-even bitonic merges of sorted lists of length 2^{2r}, resolving comparisons according to (in decreasing order of importance) intermediate column, class, color, count, and level. Running time: $O(a)$.

(f) Permute the data so that each record that is stored in a processor of the form $(c : W,\ b-r : Y,\ r : X,\ r : Z)$ is sent to processor $(c : W,\ r : X,\ b-r : Y,\ r : Z)$. At this point, the 2^{a+b} records are ordered according to (in decreasing order of importance) intermediate column, class, color, count, and level. Implementation: BPC route. Running time: $O(a)$.

Total running time: $O(a)$ (plus a recursive call).

7. **Move to correct rows.** View the 2^{a+b} records in S as occupying columns 0 through $2^{c-3}-1$ of an $(r+3,\ c+1,\ b)$–cube. The records are stored in column-major order, so if x is the jth record in S within level $Level(x)$, then x is stored in processor $(c+1 : \lfloor j/2^{r+3} \rfloor,\ r+3 : j \bmod 2^{r+3},\ b : Level(x))$. Simulating the 2^{a+b+4} processors of this $(r+3,\ c+1,\ b)$–cube, order the records in S according to (in decreasing order of importance) class, color, count, and intermediate column.

   (a) Send each record to its intermediate column and to the row given by its class, color, and count as follows. For $0 \le i \le d$, let $f(i) = \sum_{j=0}^{i-1} 2^{r-j+2}$. For $0 \le i < 2^{r-t}$, let $g(i) = i2^{t+2}$. Send each record x to processor $(c+1 : IntColumn(x),\ r+3 : f(Class(x)) + g(Color(x)) + Count(x),\ b : Level(x))$. Implementation: inverse concentrate. Running time: $O(a)$.

   (b) Map the array of balanced records from column-major to row-major storage. This moves each record that was in a processor of the form $(c+1 : X,\ r+3 : Y,\ b : Z)$ to processor $(r+3 : Y,\ c+1 : X,\ b : Z)$. Implementation: BPC route. Running time: $O(a)$.

   (c) In each of the levels in parallel, concentrate (in the new row-major order) the records in S to the first set of processors in that level. That is, if x is the jth record (in row-major order) in S within level $Level(x)$, $0 \le j < 2^a$, then x is sent to processor $(a+4 : j,\ b : Level(x))$. At this point, the 2^{a+b} records are stored within the first 2^{a+b} processors. They are ordered according to (in decreasing order of importance) class, color, count, intermediate column, and level. Implementation: concentrate. Running time: $O(a)$.

   Total running time: $O(a)$.

8. **Sort within rows.** A recursive call will be used to sort the records that are in the same class and have the same color. Before and after this sort, the records must be rearranged. The following steps implement the sort within rows.

   (a) Permute the data so that each record that was in a processor of the form $(r-t-3 : W,\ c+t+3 : X,\ b-c-t-3 : Y,\ c+t+3 : Z)$ is sent to processor $(r-t-3 : W,\ b-c-t-3 : Y,\ c+t+3 : X,\ c+t+3 : Z)$. This step is actually not required on the hypercube. On the shuffle-exchange or cube-connected cycles, its purpose is to bound the overhead associated with the recursive call. Implementation: BPC route. Running time: $O(a)$.

(b) Note that the 2^{c+t+3} records residing in any subcube of the form $(a+b-c-t-3 : X,\ c+t+3 : *)$ share the same class values. In each such subcube, save the class value in the temporary B value field of the record in processor $(a+b-c-t-3 : X,\ c+t+3 : num)$. Running time: $O(1)$.

(c) Within each subcube of $2^{2c+2t+6}$ consecutive processors, call $\mathsf{DoRoute}'(c+t+3,\ c+t+3,\ 2num+1)$ to (stably) sort each group according to the routing records created by $\mathsf{PlanRoute}()$. This sorts the records in each subcube of 2^{2r} consecutive processors according to (in decreasing order of importance) class and key value.

(d) The class fields corresponding to this level of the recursion have been overwritten by the recursive call. Restore these fields from the copies saved in Step 8b. The level fields corresponding to this level of the recursion have also been overwritten by the recursive call. For each record x in S that is currently stored in a processor of the form $(r-t-3 : W,\ b-c-t-3 : Y,\ c+t+3 : X,\ c+t+3 : Z)$, set $Level(x) = (b-c-t-3 : Y,\ c+t+3 : Z)$. Implementation: broadcast. Running time: $O(a)$.

(e) Perform odd-even bitonic merges of sorted lists of length $2^{2c+2t+6}$, resolving comparisons according to (in decreasing order of importance) class, key value, and level. Running time: $O(a)$.

(f) Permute the data so that each record that is stored in a processor of the form $(r-t-3 : W,\ b-c-t-3 : Y,\ c+t+3 : X,\ c+t+3 : Z)$ is sent to processor $(r-t-3 : W,\ c+t+3 : X,\ b-c-t-3 : Y,\ c+t+3 : Z)$. The 2^{a+b} records in S are now sorted according to (in decreasing order of importance) class, key value, and level. Implementation: BPC route. Running time: $O(a)$.

Total running time: $O(a)$ (plus a recursive call).

9. **Merge classes.** At this point, each list S_i is stored in 2^b consecutive processors. Furthermore, the lists are ordered with respect to one another according to (in decreasing order of importance) class and key value. Thus, all that remains is to merge the sorted classes. This merging task may be accomplished as follows.

   (a) View the records as forming an $(r,\ c,\ b)$–cube. Save the current processor number of each record in level 0 in its temporary A value field. Sort the records in $S \setminus U_0$ in level 0 according to their key values. Set the destination row and destination column fields of the records in $S \setminus U_0$ in level 0 to match their current positions. Implementation: sparse enumeration sort. Running time: $O(a)$.

   (b) Undo the sort that was performed in the previous step. That is, sort the records in $S \setminus U_0$ in level 0 according to their temporary A values. Then send each record x in $S \setminus U_0$ in level 0 to processor $TempA(x)$. Implementation: sparse enumeration sort, inverse concentrate. Running time: $O(a)$.

   (c) Copy the destination row and destination column fields from each record in $S \setminus U_0$ in level 0 to the destination row and destination column fields of the $2^b - 1$ other records in its pile. Implementation: broadcast. Running time: $O(a)$.

(d) For each record x in $S \backslash U_0$, set $TempA(x) = (r : DestRow(x),\ c : DestColumn(x),\ b : Level(x))$. Merge the sorted classes $U_1, \ldots, U_d$ with one another by calling MergeClasses($a+b,\ t$) on the records in $S \setminus U_0$. At this point the records in $S \setminus U_0$ are completely sorted according to their key values. Running time: $O(a)$.

(e) Merge the sorted records in U_0 with the sorted records in $S \setminus U_0$. Implementation: bitonic merge. Running time: $O(a)$.

Total running time: $O(a)$.

Analysis: Let $DR(a,b,n)$ denote the running time of DoRoute$'(a,\ b,\ n)$. If $\lfloor a/2 \rfloor \leq \log n$ then $DR(a,b,n) = O(a^2)$, and if $\lfloor a/2 \rfloor > \log n$ then

$$DR(a,b,n) = DR(r,r,2n) + DR(c+t+3, c+t+3, 2n+1) + O(a),$$

where $r = \lfloor a/2 \rfloor$, $c = \lceil a/2 \rceil$ and $t = \lceil \log \lceil 2 + \log c \rceil \rceil$. Using an analysis similar to that given in Section 6.1, this recurrence solves to give $DR(a,b,1) = O(a \log a)$. The algorithm requires only constant storage at each processor.

7.4 Modified Color Balancing

This subsection defines the subroutine Balance$'()$, which is similar to the Balance$()$ routine of Section 6.3, but uses fewer colors and allows a number of balanced records with the same color to be mapped to the same column. As a result of this weaker form of balancing, Balance$'()$ is able to balance a very large fraction of the records in a single call.

The following lemma will be useful in proving the correctness of Balance$'()$.

Lemma 7.1 *Let g and h be positive integers where $g \leq h$ and $h \bmod g = 0$, and let S be a set of h records. Associate with each record in S an integer in the range 0 through $h/g - 1$, which will be called the "color" of the record. Then there are at least $h(g-1)/2$ distinct pairs of records in S with matching colors.*

Proof: Assume that the colors are assigned to the records of S in a manner that minimizes the number of distinct pairs of records with matching colors. Now assume for the sake of contradiction that there are not exactly g records with each color. Hence, there must exist colors α and β such that α was assigned to at most $g-1$ records and β was assigned to at least $g+1$ records. But the number of distinct pairs of records with matching colors could then be reduced by changing the color of one of the records with color β to color α, which is a contradiction. Therefore, there are exactly g records with each color. Because there are $g(g-1)/2$ distinct pairs of records with each color, and there are h/g different colors, there are $h(g-1)/2$ distinct pairs of records with matching colors. $\square$

A detailed description of Balance$'(r,\ c,\ b,\ t,\ k,\ p)$ is given next.

Input/Processors: A subcube of 2^{r+c+b} processors, which will be viewed as an $(r,\ c,\ b)$–cube, and a set T of 2^{r+c} records distributed one per processor over the 2^{r+c} processors in level 0 of the cube. The parameters r, c, b, t, k, and p are all nonnegative integers, where $r \le c$, $b > c + 1$, $t < r$, and p is a prime, $2^c < p < 2^{c+1}$.

Output: A subset U of T such that $|T \setminus U| \le 2^{r+c-t-1}$, called the "balanced records", and for each record x in U an assignment to the fields $\mathit{SourceRow}(x)$, $\mathit{SourceColumn}(x)$, $\mathit{IntColumn}(x)$, $\mathit{Color}(x)$, $\mathit{Count}(x)$, and $\mathit{Class}(x)$. These fields will be referred to as the source row, source column, intermediate column, color, count, and class, respectively, of record x. The fields $\mathit{SourceRow}(x)$ and $\mathit{SourceColumn}(x)$ are set to the row and column positions, respectively, which x had when Balance() was called. The field $\mathit{IntColumn}(x)$ is in the range $0 \le \mathit{IntColumn}(x) < p$, the field $\mathit{Color}(x) = \lfloor \mathit{Rank}(T,\ x)/2^{c+t} \rfloor$, and the field $\mathit{Class}(x) = k$. For each record x in U, the field $\mathit{Count}(x)$ denotes the number of records y in U such that $\mathit{Color}(y) = \mathit{Color}(x)$, $\mathit{IntColumn}(y) = \mathit{IntColumn}(x)$, and $\mathit{SourceRow}(y) < \mathit{SourceRow}(x)$. For any balanced records x and y,

$$\mathit{SourceRow}(x) = \mathit{SourceRow}(y) \Rightarrow \mathit{IntColumn}(x) \ne \mathit{IntColumn}(y).$$

The final (and most important) output condition is that when the 2^{r+c+b} processors simulate an $(r,\ c+1,\ b)$–cube, two monotone routes that can be implemented entirely within the rows of level 0 are sufficient to move the balanced records from their source columns to their intermediate columns.

Running time: $O(c + bc/(b-c))$.

Example: The input is a cube of n processors arranged in $n^{1/4}$ rows, $n^{1/4}$ columns, and $n^{1/2}$ levels with $n^{1/2}$ records in level 0. The records are assigned colors in the range 0 through $n^{1/4}/\log\log n - 1$. In $O(\log n)$ time the algorithm balances all but $n^{1/2}/2\log\log n$ records so that there are at most $4\log\log n$ records with any given color and intermediate column value.

Algorithm Balance$'(r,\ c,\ b,\ t,\ k,\ p)$

1. For each record x in T, set the $\mathit{SourceRow}(x)$ and $\mathit{SourceColumn}(x)$ fields to the row and column positions of x, respectively. Running time: $O(1)$.
2. Assign colors to the records in T. This is accomplished by the following sequence of steps.
   (a) Sort the records in T according to their key values. Implementation: sparse enumeration sort. Running time: $O(c)$.
   (b) For each record x in T, calculate $\mathit{Rank}(T,\ x)$. Implementation: prefix operation. Running time: $O(c)$.
   (c) Calculate $|T|$, and broadcast this value to every processor in level 0. Implementation: suffix operation, broadcast. Running time: $O(c)$.
   (d) For each record x in T, set the field $\mathit{Color}(x)$ to $\lfloor \mathit{Rank}(T,\ x)/2^{c+t} \rfloor$. Running time: $O(1)$.
   (e) Return the records in T to their original positions by sorting them according to their source row and source column fields. Implementation: sparse enumeration sort. Running time: $O(c)$.

Total running time: $O(c)$.

3. View the 2^{r+c} records as occupying columns 0 through $2^c - 1$ in level 0 of an $(r,\ c+1,\ b)$–cube. Simulating the $2^{r+c+b+1}$ processors of this $(r,\ c+1,\ b)$–cube, create a *dummy record* in each processor in columns 2^c through $p-1$ of level 0. Assign color $\lfloor i/2^t \rfloor$ to each dummy record in row i, $0 \le i < 2^r$. Running time: $O(1)$.

4. Simulating an $(r,\ c+1,\ b)$–cube, copy each (input or dummy) record in row i, $0 \le i < 2^r$, and column j, $0 \le j < p$, of level 0 to row i and column j of each of the first p levels. Let $T_{i,j}$ denoted the pile of p records in row i and column j. Implementation: broadcast. Running time: $O(c)$.

5. Simulating an $(r,\ c+1,\ b)$–cube, permute the copies of the records in the first p levels of the cube as follows. In each level h, $0 \le h < p$, move the record (if any) in row i and column j to row i and column $j + ih \bmod p$, $0 \le i < 2^r$, $0 \le j < 2^c$. Implementation: two monotone routes (one for the records going to higher-numbered processors, and one for the records going to lower-numbered processors). Running time: $O(c)$.

6. Define a *collision* to be the mapping of two records in the same level and with the same color to the same column. For each collision, denote the record which participates in the collision and is in the higher-numbered row as the "owner" of the collision. Note that two records that begin in the same level and row cannot collide with one another. Hence, Lemma 6.1 implies that for any pair of piles of the same color, $T_{i,j}$ and $T_{i',j'}$, there is a total of at most one collision between the records in $T_{i,j}$ and the records in $T_{i',j'}$. Because there are $p2^t$ piles of each color, the total number of collisions is at most $p2^t(p2^t-1)(1/2)2^{r-t} = p^2 2^{r+t-1} - p2^{r-1}$, and one of the first p levels must contain at most $p2^{r+t-1} - 2^{r-1}$ collisions. Simulating an $(r,\ c+1,\ b)$–cube, the following steps determine which of the the first p levels contains the smallest number of collisions.

   (a) (Stably) sort the records in each column of each level according to their colors. Implementation: sparse enumeration sort. Running time: $O(bc/(b-c))$.

   (b) Set the count field of each record to its rank within the set of records having the same level, column, and color. Each record x for which $Count(x) < 2^{t+2}$ is marked as being "acceptable". Implementation: segmented prefix operation. Running time: $O(r)$.

   (c) In each set of records having the same level, column, and color, designate the last record as "leader". Implementation: segmented suffix operation. Running time: $O(r)$.

   (d) For each leader, locally calculate the number of collisions involving records of its level, column, and color. Then sum these values to calculate the total number of collisions within each level. Implementation: prefix operation. Running time: $O(c)$.

   (e) Determine the level h, $0 \le h < p$, that contains the smallest number of collisions (break ties arbitrarily), and broadcast the result to every processor in the first p levels. Implementation: prefix operation, broadcast. Running time: $O(c)$.

(f) Return each record x to the position that it had following Step 4 by sorting the records in each level according to their source row and source column fields and then routing them to the row and column given by their source row and source column fields. Implementation: sparse enumeration sort, monotone route. Running time: $O(bc/(b-c))$.

Total running time: $O(bc/(b-c))$.

7. Let level h, $0 \leq h < p$, be the level with the fewest collisions. For each record x in level h that was designated as being acceptable in Step 6b, set

$$\mathit{IntColumn}(x) = (\mathit{SourceColumn}(x) + \mathit{SourceRow}(x) \cdot h) \bmod p$$

and set $\mathit{Class}(x) = k$. Let V denote this set of acceptable records in level h and let $\overline{V}$ denote the set of records in level h that are not in V. The non-dummy records in V form the set of balanced records, U. Note that each record in $\overline{V}$ owns at least 2^{t+2} collisions. Furthermore, note that it would be possible to re-color the records in $\overline{V}$ so that each record in $\overline{V}$ owns at most 2^t collisions and the number of collisions owned by each record in V does not increase. (To obtain this re-coloring, simply give each record in $\overline{V}$ a color that appears at most 2^t times in its column.) This re-coloring would reduce the number of collisions by at least $3(2^t)|\overline{V}|$. But Lemma 7.1 implies that even after such a re-coloring, each column would have at least $2^r(2^t-1)/2$ collisions and there would be a total of at least $p2^r(2^t-1)/2 = p2^{r+t-1} - p2^{r-1}$ collisions in level h. Thus re-coloring would eliminate at most $(p-1)2^{r-1}$ collisions. Therefore, $3(2^t)|\overline{V}| \leq (p-1)2^{r-1}$ and $|T \setminus U| \leq |\overline{V}| \leq (p-1)2^{r-t-1}/3 < 2^{r+c-t-1}$. Finally, send each record in U back to level 0. Implementation: monotone route. Running time: $O(c)$.

7.5 Separate Classes

This subsection describes the algorithm SeparateClasses(s, t), which is used as a subroutine by DoRoute().

Input/Processors: A subcube of 2^s processors holding a set S of at most 2^{s-t} records, where s and t are nonnegative integers and $t \leq s$. At most one record is stored in each processor. Each record x in S has a field $\mathit{Class}(x)$ which will be called the class of x. For each record x in S, $1 \leq \mathit{Class}(x) \leq 2^t$. Let U_i denote the set of records $\{x \in S \mid \mathit{Class}(x) = i\}$, $1 \leq i \leq 2^t$.

Output: The records in S (stably) separated according to class. That is, if upon input x is the jth record in class k, $0 \leq j < 2^{s-t}$, $1 \leq k \leq 2^t$, then x is output at processor $j + \sum_{i=1}^{k-1} |U_i|$.

Running time: $O(s)$.

Example: The input is a set of n processors holding $n/\log n$ records, each of which is assigned a class in the range 1 through $\log n$. The algorithm separates the records according to class in $O(\log n)$ time.

Algorithm SeparateClasses(s, t)

1. Concentrate the records in S to processors 0 through $|S|-1$. That is, if x is the jth record in S, $0 \le j < 2^{s-t}$, then x is sent to processor j. Implementation: concentrate. Running time: $O(s)$.

2. View the 2^s processors as forming a two-dimensional array with 2^t rows and 2^{s-t} columns, numbered in row-major order. Thus the records in S are currently stored in row 0 of this array. Map this array from row-major to column-major storage. This moves each record that was in a processor of the form $(t:0,\ s-t:Y)$ to processor $(s-t:Y,\ t:0)$. Implementation: BPC route. Running time: $O(s)$.

3. Send each record to the row specified by its class, without leaving its current column. That is, if x is in processor $(s-t:Y,\ t:0)$, then x is sent to processor $(s-t:Y,\ t:$ $\mathit{Class}(x)-1)$. Implementation: monotone route. Running time: $O(t)$.

4. Map the array from column-major to row-major storage. This moves each record that was in a processor of the form $(s-t:Y,\ t:Z)$ to processor $(t:Z,\ s-t:Y)$. Implementation: BPC route. Running time: $O(s)$.

5. Concentrate (in the new row-major order) the records in S to processors 0 through $|S|-1$. That is, if x is the jth record (in row-major order) in S, $0 \le j < 2^{s-t}$, then x is sent to processor j. Implementation: concentrate. Running time: $O(s)$.

7.6 Merge Classes

This subsection describes the algorithm MergeClasses(), which is used as a subroutine by DoRoute(). The algorithm MergeClasses() is essentially the inverse of the algorithm SeparateClasses().

Input/Processors: A subcube of 2^s processors holding a set S of at most 2^{s-t} records, where s and t are nonnegative integers and $t \le s$. At most one record is stored in each processor. Each record x in S has fields $\mathit{Class}(x)$ and $\mathit{TempA}(x)$. These fields will be referred to as the class and temporary A value, respectively, of x. For each record x in S, $1 \le \mathit{Class}(x) \le 2^t$ and $0 \le \mathit{TempA}(x) < |S|$. No two records in S have the same temporary A value. The records in S are arranged in sorted order according to (in decreasing order of importance) class and temporary A value. Let U_i denote the set of records $\{x \in S \mid \mathit{Class}(x) = i\}$, $1 \le i \le 2^t$.

Output: The records in S routed to the processors specified by their temporary A values. That is, for each record x in S, x is output at processor $\mathit{TempA}(x)$.

Running time: $O(s)$.

Example: The input is a set of n processors holding $n/\log n$ records, each of which is assigned a class in the range 1 through $\log n$ and a unique temporary A value in the range 0 through $n/\log n - 1$. The records are sorted by class, and within each class, by temporary A value. The algorithm sends each record to the processor specified by its temporary A value in $O(\log n)$ time.

Algorithm MergeClasses(s, t)

1. View the 2^s processors as forming a two-dimensional array with 2^t rows and 2^{s-t} columns, numbered in row-major order. Send each record x in S to the processor in row $Class(x) - 1$ and column $TempA(x)$. Implementation: monotone route. Running time: $O(s)$.

2. Map the array from row-major to column-major storage. This moves each record x in S from processor $(t : Class(x)-1,\ s-t : TempA(x))$ to processor $(s-t : TempA(x),\ t : Class(x) - 1)$. Implementation: BPC route. Running time: $O(s)$.

3. Concentrate (in the new column-major order) the records in S to processors 0 through $|S| - 1$. That is, if x is the jth record (in column-major order) in S, $0 \leq j < |S|$, then x is sent to processor j. Implementation: concentrate. Running time: $O(s)$.

8 Concluding Remarks

This paper has provided a deterministic sorting algorithm that runs in $O(\log n\,(\log\log n)^2)$ time on an n-processor hypercube, shuffle-exchange, or cube-connected cycles. As a result, the gap between the known upper and lower bounds for this problem has been exponentially reduced. However, the intriguing question of existence of an $O(\log n)$ deterministic sorting algorithm for the hypercube and related computers remains open.

The conference version of this paper listed a number of extensions and applications of the Sharesort algorithm, and stated that the details would appear in the full version of the paper [7]. Instead, these additional results will appear in a separate paper.

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. Cole and C. K. Yap. A parallel median algorithm. *IPL*, 20:137–139, 1985.

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

[6] R. E. Cypher. Theoretical aspects of VLSI pin limitations. *SIAM Journal on Computing*, 22:58–63, 1993.

[7] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47:501–548, 1993.

[8] J. P. Fishburn and R. A. Finkel. Quotient networks. *IEEE Transactions on Computers*, C-31:288–295, 1982.

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

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

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

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

[13] C. G. Plaxton. *Efficient Computation on Sparse Interconnection Networks*. PhD thesis, Department of Computer Science, Stanford University, September 1989.

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

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

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

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