# Techniques for Shared Key Sorting


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

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



## Abstract

This paper presents deterministic algorithms for sorting $n$ records on a $p$ processor hypercube, shuffle-exchange or cube-connected cycles computer when $n \geq p$. The fastest known deterministic sorting algorithm running on any of these computers for the case $n = p$ is Sharesort, which runs in $O(\log n(\log\log n)^2)$ time in the worst case. An important subroutine in Sharesort solves a problem called *shared key sorting*. This paper presents new techniques for shared key sorting. These techniques yield algorithms for the sorting problem that are faster than all previously known algorithms over a wide range of the ratio $n/p$.

Specifically, the techniques described in this paper yield 1) a relatively simple sorting algorithm that runs in $O((n/p)\log n(\log\log n - \log\log(2n/p)) + \log^3 n)$ time, 2) an $O((n/p)^{3/4}\log^{3/2} n)$ time sorting algorithm for the case $p \leq n \leq p\log^2 p$, 3) a nonconstructive $O(\log n \log\log n)$ time sorting algorithm for the case $n = p$, and 4) an $O(\log n \log\log n \log^* n)$ time sorting algorithm for the case $n = p$ which has certain preprocessing requirements. The preprocessing required by the fourth algorithm is the loading of a constant amount of routing information in each processor. This routing information can be calculated in $O(\log^3 p)$ time, and it is independent of the sorting problem to be solved, so it can be loaded just once prior to performing any sorts. Finally, we also give an $O(\log n \log\log n)$ time algorithm for sorting $n$ packets on a multi-butterfly with $p = n\log n$ processors. All of the above algorithms require only $O(n/p + 1)$ storage at each processor.



*Supported in part by an NSF graduate fellowship.

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

## 1 Introduction

This paper presents new algorithms for sorting on hypercube, shuffle-exchange [18] and cube-connected cycles [16] computers when there are at least as many records to sort as processors available. The input to the sorting problem is a set of $n$ records, evenly distributed over the $p$ processors of a parallel machine.[1] The problem is to rearrange the records so that they are in sorted order with respect to their key fields. No assumptions will be made about the range of values of the key fields, so only comparison-based sorts will be used. Unless stated otherwise, it will be assumed that the key values are distinct, as it is simple to create distinct keys by using a record's original position to break ties.

As is the case for sequential computers, sorting is central to the efficient use of parallel computers. In fact, sorting is particularly important in parallel message-passing computers because it can be used to implement pointer operations and to simulate shared memory models. Specifically, let a *cube-type computer of size* $p$ denote a hypercube, shuffle-exchange or cube-connected cycles computer with $p$ processors, and let $\mathsf{Sort}(n,\ p)$ be the worst case time required to sort $n$ records on a cube-type computer of size $p$. Algorithms given by Nassimi and Sahni [11] can be used to implement a single operation of a powerful shared memory model (the Priority CRCW PRAM) that has $n$ processors and $n$ words of memory using a cube-type computer of size $p$ in $O(\mathsf{Sort}(n,\ p))$ time. Thus improvements in sorting can lead to improvements in a wide range of other problems.

The problem of sorting with message-passing parallel computers has received a great deal of attention. One of the most notable results was obtained by Leighton [8], who used the $O(\log n)$ depth[2] sorting network of Ajtai, Komlós and Szemerédi [2] to obtain an $O(\log n)$ time sorting algorithm for a bounded degree message-passing parallel computer with $p = n$ processors. This result is clearly optimal, as it performs only $O(n \log n)$ comparisons. Another breakthrough was provided by the randomized Flashsort algorithm of Reif and Valiant [17], which sorts every possible input permutation with high probability in $O(\log n)$ time on a cube-connected cycles with $p = n$ processors.

Unfortunately, no algorithm with an optimal worst case complexity has been created for sorting on cube-type computers with $p = n$ processors. One of the earliest results is Batcher's bitonic sort [3], which runs in $O(\log^2 n)$ time on cube-type computers when $n = p$ [18, 16]. When $n$ is sufficiently small with respect to $p$, the fastest known algorithm for sorting on cube-type computers is Nassimi and Sahni's sparse enumeration sort, which runs in $O(\log n \log p/ \log(p/n))$ time [14]. The problem of sorting on cube-type computers when $n > p$ has been addressed by a number of researchers. Aggarwal and Huang created an algorithm that sorts in $O((n/p) \log n(\log n/ \log(n/p))^{\alpha})$ time, where $\alpha = (3 - \log 3)/(\log 3 - 1) \approx 2.419$ [1]. This was improved by Cypher and Sanz, who created an algorithm called Cubesort that runs in $O((n/p) \log^2 n/ \log(n/p))$ time, provided $p \log^{(x)} p \leq n$ for some positive integer $x$ (see Section 2.2 for the definition of $\log^{(x)} p$). A further improvement was given by Plaxton's Smoothsort algorithm [15]. If $q = \log^{3/2} p \log\log p$, then Smoothsort runs on a hypercube in $O(\log(n/p) \log^3 p)$ time when $n \leq pq$ and in $O((n/p) \log^{3/2} p/ \log^{1/2}(n/pq))$ time when $n > pq$.

Recently, Cypher and Plaxton presented an algorithm called Sharesort that runs in $O(\log n(\log\log n)^2)$ time on cube-type computers with $p = n$ processors [6]. In terms of worst case analysis, this is the first result to improve upon the $O(\log^2 n)$ time bitonic sort for cube-type computers with $p = n$ processors. For $n > p$ this algorithm can be easily generalized to obtain an $O((n/p) \log n \log\log n \log\log(n/p))$ time sorting algorithm.

Sharesort operates by reducing the sorting problem to the *shared key sorting problem*. The shared key sorting problem is a sorting problem in which the $n$ records to be sorted consist of $n^{\alpha}$ lists of $n^{1-\alpha}$ records each, for some constant $\alpha$, $0 < \alpha < 1$. The records in each of the lists all have the same key value. In other words, the input to the shared key sorting problem can be viewed as an $n^{\alpha} \times n^{1-\alpha}$ array of records in which the $n^{1-\alpha}$ records in each row have the same key value. The problem is to permute the rows of this array so that they are in sorted order

[1] It will be assumed throughout that $n \leq p^k$ for some constant $k$.

[2] Throughout this paper, "log" will denote the base 2 logarithm.

with respect to their keys.

On a cube-type computer with $p = n$ processors, the shared key sorting problem is easier than the general sorting problem for two reasons. First, in the shared key sorting problem a single record can be selected from each row and these $n^\alpha$ records can be sorted in $O(\log n)$ time by using Nassimi and Sahni's sparse enumeration sort. The result of this sort gives the permutation of the rows that is required by the shared key sort. Thus the shared key sorting problem is no longer a sorting problem, in which records must be compared, but is instead a routing problem, in which a known permutation of the rows must be performed. Second, this routing of the rows to their sorted positions can be performed efficiently by dividing it into a *planning stage* and a *routing stage*. In the planning stage, all $n$ processors are used to figure out how to route the $n^\alpha$ records in a single column to their destinations. Then in the routing stage, the plan that was calculated is applied in parallel to each of the columns. Note that during the planning stage the number of processors available greatly exceeds the number of records that are being permuted. Thus the planning stage can use the excess processors to calculate a plan very efficiently.

In the original description of Sharesort, an $O((n/p)\log n \log\log n)$ time algorithm for shared key sorting was presented [6]. The remainder of this paper is devoted to obtaining new solutions to the shared key sorting problem that have improved running times. These new algorithms for shared key sorting will in turn yield new algorithms for sorting on cube-type computers that improve upon all previous deterministic solutions to the problem for many values of the ratio $n/p$.

Two of the new algorithms use a well known technique called Benes routing [4] to perform the desired permutation of the rows. The first of these algorithms performs a shared key sort in $O((n/p)\log n + \log^3 n)$ time, while the second performs a shared key sort in $O((n/p)^{3/4}\log^{3/2} n)$ time when $p \leq n \leq p\log^2 p$. These results yield general sorting algorithms that run in $O((n/p)^{3/4}\log^{3/2} n)$ time when $p \leq n \leq p\log^2 p$ and in $O((n/p)\log n(\log\log n - \log\log(2n/p)) + \log^3 n)$ time when $n \geq p\log^2 p$. Note that these results improve upon the performance of the original Sharesort algorithm when $n = \omega(p\log^2 p/(\log\log p)^4)$.

Next, a non-constructive $O((n/p)\log n)$ time algorithm for shared key sorting is presented. This yields a non-constructive $O((n/p)\log n(\log\log n - \log\log(2n/p))$ time algorithm for the general sorting problem. Finally, an $O(\log n \log^* n)$ time shared key sorting algorithm for the case $n = p$ is given which has certain preprocessing requirements (see Section 2.2 for the definition of $\log^* n$). This in turn yields an $O(\log n \log\log n \log^* n)$ time sorting algorithm for the case $n = p$ which also has preprocessing requirements. The preprocessing that is required by these algorithms is simply the loading of a single pointer in each processor. These pointers can be calculated in $O(\log^3 p)$ time on a cube-type computer. Furthermore, these pointers are independent of the sorting problem to be solved, so they can be loaded once prior to performing any sorts. This last algorithm is also generalized to the case of $n > p$. All of the algorithms given in this paper require only $O(n/p)$ storage at each processor.

The remainder of this paper is organized as follows. The model of computation, definitions and notation, and a review of some previously known operations are presented in Section 2. Section 3 gives the shared key sorting algorithms that are based on Benes routing. The non-constructive algorithm and the algorithms which require preprocessing are presented in Sections 4 and 5, respectively. An $O(\log n \log\log n)$ sorting algorithm for a sparsely loaded multi-butterfly is described in Section 6.

## 2 Preliminaries

### 2.1 The Model of Computation

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

The records which will be sorted consist of a constant number of fields, each of which is one word long. One of the fields is a *key* that determines the sorted order of the records. The only operations that we require on keys are copy and comparison.

### 2.2 Definitions and Notation

Let $\log^{(0)} x \equiv x$ and for all integers $i \geq 1$, let $\log^{(i)} x \equiv \log(\log^{(i-1)} x)$. Let $\log^* x$ denote the least integer $i$ such that $\log^{(i)} x \leq 1$.

Given a hypercube of dimension $d$, let any string $\alpha$ of length $d$ over the alphabet $\{0, 1, *\}$ correspond to that set of processors for which the ID "matches" $\alpha$ 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, $\sum_{0 \leq i < r} d_i = d$, and $\alpha_i$ is either $*$ or a $d_i$-bit integer. Such a tuple corresponds to the string $\beta_0 \cdots \beta_{r-1}$, where $\beta_i$ is the $d_i$-bit string corresponding to the binary representation of $\alpha_i$ if $\alpha_i \neq *$, and $*^{d_i}$ otherwise.

Given a set of records $S$ and a record $x$ in $S$, let $Rank(S,\ x)$ denote the rank of $x$ in the set $S$, that is, the number of records in $S$ having a lower associated key than $x$.

### 2.3 Useful Operations

A number of previously known operations will be used in this paper. These algorithms will be described in terms of the parameter $n$, a power of 2. With the exception of sparse enumeration sort and Cubesort, all of these operations run on a cube-type computer with $n$ processors in $O(\log n)$ time.

*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 [12]. *Broadcasting* copies a record from one processor to all $n$ processors [11].

*Benes routing* is a technique for implementing arbitrary permutations efficiently [4]. Benes routing permutes $n$ records by sending each record first to an intermediate destination and then to its final destination given by the permutation. The intermediate destinations are chosen so that the records will not collide with one another. As a result, a Benes route of $n$ records requires only $O(\log n)$ time. However, the intermediate destinations depend on the permutation that is being performed, and the problem of calculating the intermediate destinations can take considerable time. In particular, Nassimi and Sahni have shown that the intermediate destinations for a Benes route of $n$ records can be calculated in $O(\log^3 p)$ time when $n \leq p^\alpha$ for some constant $\alpha$, $0 < \alpha < 1$ [13].

*Sparse enumeration sort* is a useful sorting technique for the case when the number of records to be sorted, $n$, is much smaller than the number of processors available, $p$ [14]. Sparse enumeration sort runs in $O(\log n \log p / \log(p/n))$ time.

*Cubesort* is a sorting algorithm that reduces the problem of sorting $n$ records into a number of smaller sorting problems. Specifically, let $g$ be a power of 2 where $\log^{(x)} n \leq g$ for some constant $x$. Then Cubesort consists of $O(\log^2 n / \log^2 g)$ stages. Each stage partitions the $n$ records into $n/g$ groups of $g$ records each and sorts these groups in parallel.

### 2.4 Shared Key Sorting

This subsection contains two lemmas regarding shared key sorting. Recall that the input to the shared key sorting problem is a set of $n$ records, arranged as $n^\alpha$ lists of $n^{1-\alpha}$ records each, where the records in each list all have the same key value. This shared key sorting problem will be called the $(n, \alpha)$ shared key sorting problem. The first lemma follows immediately from Cubesort.

**Lemma 2.1** *Let $\alpha$ and $\beta$ be constants where $0 < \alpha \leq \beta < 1$. Any $(n, \beta)$ shared key sorting problem can be reduced to a constant number of $(n, \alpha)$ shared key sorting problems.*

The second lemma summarizes the relationship between the sorting problem and the shared key sorting problem that is given by the algorithm Sharesort.

**Lemma 2.2** *Let $SKS(n, s)$ be the running time of an algorithm that solves the $(n, 1/2)$ shared key sorting problem on an $n/s$ processor cube-type computer. Then $n$ items can be sorted on an $n/s$ processor cube-type computer in time proportional to*

$$\sum_{i=0}^{k}(2^i) \cdot SKS(n^{1/2^i}, s)$$

*where $k = \log\log n - \log\log(n/p)$.*

### 2.5 Time Analysis

When implementing the algorithms given in this paper on a hypercube computer, the running time can be calculated by adding together the running times of the subroutines of which they are composed. When implementing them on a 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 algorithms. Finally, when implementing them 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 be managed in time proportional to the running time of the shuffle-exchange implementation [5].

## 3 Benes Shared Key Sorting

This section presents two shared key sorting algorithms that are based on Benes routing. Both algorithms use sparse enumeration sorts to find the correct sorted position of each record. The algorithms then calculate the intermediate destinations for the Benes route which sends each record from its original position to its sorted position. Finally, this Benes route is implemented to complete the sort. Both algorithms are simpler than the original shared key sorting algorithm [6], and they both provide the fastest known solution to the shared key sorting problem for certain ranges of the the ratio $n/p$.

In order to simplify the description of these algorithms, they will be defined in terms of the parameters $a$ and $v$, where $2^{2a} = n$ and $2^{2a-v} = p$. The input to both algorithms is a set $S$ of $2^{2a}$ data records organized as $2^a$ lists of $2^a$ records each, where the $2^a$ records in each list have the same key field. Let $S_i$ denote the $i$th list of $2^a$ records, $0 \leq i < 2^a$, and let $T$ be the set of $2^a$ records obtained by taking the 0th record from each list $S_i$. The records of $S$ are stored in a subcube of dimension $2a - v$, with the $j$th element of the list $S_i$ being stored in memory location $j \bmod 2^v$ of processor $(a : i,\ a - v : k)$ where $k = \lfloor j/2^v \rfloor$, $0 \leq i < 2^a$, $0 \leq j < 2^a$.

The problem is to use the $2^{2a-v}$ processors to put the lists $S_i$ into sorted order. More formally, if $x$ is the $j$th record in list $S_i$, then $x$ is sorted to memory location $j \bmod 2^v$ of processor $(a : \mathit{Rank}(T,\ x),\ a - v : k)$ where $k = \lfloor j/2^v \rfloor$, $0 \leq i < 2^a$, $0 \leq j < 2^a$.

### 3.1 Basic Benes Shared Key Sorting

This subsection defines the algorithm $\mathsf{BenesSKS}(a,\ v)$. This algorithm, which is particularly simple, is optimal when $2^v \geq a^2$.

Range of applicability: $0 \leq v < a$ and $(a - v) = \Theta(a)$.

Running time: $O(2^v a + a^3)$.

**Algorithm** $\mathsf{BenesSKS}(a,\ v)$

1. Sort the records in $T$. For each record in $T$ create a new record, called a routing record, that gives the original position and sorted position of that record. That is, for each record $x$ in $T$, if $x$ is in $S_i$ then create the routing record $(i, \mathit{Rank}(T,\ x))$. These routing records define a permutation on the $2^a$ data records in $T$. Implementation: sparse enumeration sort. Running time: $O(a)$.

2. Calculate the intermediate destinations for performing the Benes route defined by the routing records. Append each intermediate destination to the corresponding routing record. Thus the routing records define a Benes routing of $2^a$ data records, where routing record $(i, j, k)$ indicates that the data record in location $i$ is sent to location $j$ via intermediate destination $k$. Store routing record $(i, j, k)$ in processor $(a : i,\ a - v : 0)$. Running time: $O(a^3)$.

3. Send a copy of each routing record of the form $(i, j, k)$ to all of the data records in $S_i$, $0 \leq i < 2^a$. Implementation: broadcast. Running time: $O(a + 2^v)$.

4. Send each data record to the final destination given by its routing record via the intermediate destination given by its routing record. That is, a data record in processor $(a : i,\ a - v : j)$ which has the routing record $(i, k, m)$ is sent to processor $(a : k,\ a - v : j)$ via processor $(a : m,\ a - v : j)$. Running time: $O(2^v a)$.

### 3.2 Improved Benes Shared Key Sorting

This subsection defines the algorithm $\mathsf{BenesSKS}'(a,\ v)$. This algorithm improves upon the running time of $\mathsf{BenesSKS}()$ when when $2^v = o(a^2)$, and it improves upon the running time of the original shared key sorting algorithm [6] when $2^v = \omega(a^2/\log^4 a)$. The idea is to use Cubesort to reduce the problem of sorting the entire set of records into a number of smaller sorting problems. These smaller sorting problems are then solved by performing Benes routes. The advantage of this approach is that the intermediate destinations for the Benes routes can be found quickly because each Benes route is being performed over a small set of records.

Range of applicability: $0 \leq v < a$ and $2^v \leq a^2$.

Running time: $O(2^{3v/4}a^{3/2})$.

**Algorithm** $\mathsf{BenesSKS}'(a,\ v)$

1. Let $g = \lceil 2^{v/4}a^{1/2} \rceil$. Sort the records in $T$ by using Cubesort and the sparse enumeration sort. Specifically, use Cubesort to reduce the problem of sorting the $2^a$ records in $T$ to $O(a^2/g^2) = O(a/2^{v/2})$ stages of sorting groups of $2^g$ records. Let $h$ be the number of Cubesort stages that are required. During stage $i$, $0 \leq i < h$, use a sparse enumeration sort to sort the groups of $2^g$ records. Use BPC routes between stages to ensure that the spare processors are distributed evenly between the groups of $2^g$ records. The data records in $T$ will be viewed as forming an array with $2^a$ locations. Create a set of routing records to store the location in this array of each data record during each stage of Cubesort. In particular, a routing record of the form $(i, j, k)$ indicates that the data record which was in location $j$ immediately prior to stage $i$ of Cubesort was in location $k$ immediately following stage $i$ of Cubesort. Running time: $O(hg) = O(a^{3/2}/2^{v/4})$.

2. For all $h$ stages in parallel, calculate the intermediate destinations for the Benes route that implements the sorts performed in the given stage. That is, for each stage $i$, $0 \leq i < h$, and for each routing record $(i, j, k)$, calculate the intermediate destination for the Benes route that sends the data record from location $j$ to location $k$. Append each such intermediate destination to its corresponding routing record. This creates routing records of the form $(i, j, k, m)$ where the sorts performed in stage $i$ of Cubesort can be implemented by using a Benes route on an array of $2^a$ data records which sends the data record from location $j$ to location $k$ via intermediate destination $m$. Note that the Benes routes implement sorts of $2^g$ items. Running time: $O(g^3) = O(2^{3v/4}a^{3/2})$.

3. Let $d = \lceil \log h \rceil$. Copy each routing record of the form $(i, j, k, m)$ to each processor of the form $(a : j,\ a - v - d : *,\ d : i)$ for $0 \leq i < h$, $0 \leq j < 2^a$. At this point all of the routing records that will be needed by any processor are stored in a block of $2^d$ neighboring processors. Implementation: monotonic route, BPC route, broadcast, BPC route. Running time: $O(a)$.

4. For $i = 0$ through $h-1$, do the following. First, each processor of the form $(a : j,\ a-v : *)$ obtains a copy of the routing record of the form $(i, j, k, m)$ for $0 \leq j < 2^a$. Then each processor attaches a copy of the routing record that it just obtained to each of the $2^v$ data records that it holds. Next, each data record is sent to the final destination given by its routing record via the intermediate destination given by its routing record. That is, a data record in processor $(a : j,\ a - v : q)$ which has the routing record $(i, j, k, m)$ is sent to processor $(a : k,\ a - v : q)$ via processor $(a : m,\ a - v : q)$. Implementation: broadcast, Benes route. Running time: $O(h(2^v)g) = O(2^{3v/4}a^{3/2})$.

## 4 Non-Constructive Shared Key Sorting

This section presents an optimal $O(\log n)$ time non-constructive algorithm for deterministic shared key sorting on cube-type computers. This time bound for shared key sorting implies a non-constructive $O(\log n \log\log n)$ time bound for sorting [6]. The algorithm is termed "non-constructive" because no efficent (polynomial time) algorithm is known for its construction.

In the following, let $P(n)$ denote the set of all $n!$ possible permutations of $n$ objects, and for each $n$ let $X(n)$ denote some fixed set of "advice" strings. Let $\mathcal{A}$ denote a uniform permutation routing algorithm that takes as input a permutation $\pi \in P(n)$ (the value $\pi(i)$ is given at processor $i$, $0 \leq i < n$) and a string $\sigma \in X(n)$. We say that a string $\sigma$ *covers* a permutation $\pi$ (or, equivalently, that *pi is covered by* $\sigma$) if and only if algorithm $\mathcal{A}$ routes permutation $\pi$ in $O(\log n)$ time given string $\sigma$. Further assume that algorithm $\mathcal{A}$ satisfies the following properties:

1. Every string in $X(n)$ covers at least a $1 - \frac{1}{n}$ fraction of the permutations in $P(n)$.

2. Every permutation in $P(n)$ is covered by at least a $1 - \frac{1}{n}$ fraction of the strings in $X(n)$.

The well-known hypercube routing scheme of Valiant and Brebner [20] may be viewed as an example of such an algorithm, with $X(n) = P(n)$. The following lemma proves the existence of a small set of strings that together cover every permutation in $P(n)$. Such a set will be referred to as a *covering set* for $P(n)$. Unfortunately, no efficient (polynomial time) algorithm is known for constructing a small covering set for $P(n)$.

**Lemma 4.1** *There exists a subset $S$ of $X(n)$ with $|S| \leq n$, such that $S$ is a covering set for $P(n)$.*

**Proof:** Consider the following (inefficient) construction of such a set $S$. Initially, $S$ is the empty set and none of the permutations in $P(n)$ have been covered. Now add strings to $S$ in a greedy fashion, at each step adding the string in $X(n)$ that covers the maximum number of previously uncovered permutations in $P(n)$. The procedure terminates when all of the permutations in $P(n)$ have been covered. For $\pi \in P(n)$, $\sigma \in X(n)$ let $\xi(\pi, \sigma) = 1$ if $\sigma$ covers $\pi$, and 0 otherwise. Now consider the set $U$ of uncovered permutations that remain before a given step. The second property stated above implies that

$$\sum_{\pi \in U)} \sum_{\sigma \in X(n)} \xi(\pi, \sigma) \geq |U||X(n)|(1 - \frac{1}{n}).$$

Averaging over all strings in $X(n)$ implies that some string must cover at least $(1 - \frac{1}{n})|U|$ permutations in $U$. Hence, after $k$ steps of the preceding construction, at most $n!/n^k$ permutations remain uncovered, and the procedure terminates within $n$ steps. □

The following lemma demonstrates that a polynomial-sized covering set is useful for shared key sorting.

**Lemma 4.2** *For every $n$, let $Y(n)$ be a covering set for $P(n)$ with $|Y(n)| \leq n^k$, where $k$ is some positive constant. Given these covering sets, the $(n, 1/2)$ shared key sorting problem can be solved in optimal $O(\log n)$ time.*

**Proof:** If $k > 1$ then apply Cubesort as in Lemma 2.1 in order to reduce the given $(n, 1/2)$ shared key sorting problem to a constant depth of sorts of the form $(a, 1/(k'+1))$ where $k' \geq k$. Each of these subproblems may be solved in $O(\log n)$ time by executing algorithm $\mathcal{A}$ for $O(\log a)$ time in parallel over $a^k$ subcubes of $a$ processors, with a different string in $Y(a)$ assigned to each subcube. Since $Y(a)$ is a covering set, at least one of the subcubes is guaranteed to complete the permutation route. The lowest-numbered subcube that completed the route then broadcasts its string to all of the other subcubes. Finally, this string is used in all of the other subcubes to perform the route quickly. □

**Theorem 4.1** *There exists an optimal $O(\log n)$ time, $O(1)$ space algorithm for $(n, 1/2)$ shared key sorting on the hypercube, CCC and SE.*

**Proof:** These results follow easily from the preceding pair of lemmas given the existence of randomized routing algorithms for each of these networks. These algorithms route every input permutation in $O(\log n)$ with high probability. The first such algorithm is due to Valiant and Brebner, and runs on the hypercube using $O(\log n)$ storage per processor [20]. A succession of papers have led to similar results for the CCC and SE, and have reduced the per processor storage requirement to a constant in each case. For the most recent improvements to randomized routing techniques, and for pointers to previous work in this area, the reader is referred to [10]. □

Another significant improvement in randomized routing algorithms has been the reduction in the number of random bits from $O(n \log n)$ to $O(\log^2 n)$ [7]. It is interesting to note that if a randomized routing algorithm requiring only $O(\log n)$ random bits could be devised, then Lemmas 4.1 and 4.2 would provide an on-line shared key sorting algorithm with the same (optimal) asymptotic performance as the non-constructive algorithms of Theorem 4.1.

## 5 Near-Optimal Shared Key Sorting

This section contains a brief description of the shared key sorting algorithm AscendSKS(), which runs in $O(\log n \log^* n)$ time on a cube-type computer with $p = n$ processors, provided the necessary pre-processing has been performed. This in turn yields an $O(\log n \log\log n \log^* n)$ time algorithm for solving the general sorting problem when $n = p$, again given certain pre-processing. The pre-processing required by these algorithms is independent of the sorting problem being solved, and can be calculated in $O(\log^3 n)$ time.

As is the case for the other shared key sorting algorithms presented here, AscendSKS() first uses a sparse enumeration sort to determine the sorted position of each record. The problem of routing the permutation which implements the sort is then divided into a *planning stage* and an *implementation stage*. Because the planning stage calculates a route for only $n^\alpha$ records (one from each list) and $n$ processors are available, the excess processors can be used to improve the running time of the planning stage.

Both the planning stage and the implementation stage have (identical) recursive structures. Let us assume that input to the shared key sorting problem consists of $n^\alpha$ lists of $n^{1-\alpha}$ records each. The idea in AscendSKS() is to use recursive calls to implement permutations of $2^g$ records each. These recursive calls route the records by correcting bits in order from least to most significant. That is, after the $i$th recursive call, the $ig$ least significant bits of the ID of the processor holding any record $x$ match the $ig$ least significant bits of the ID of the processor to which $x$ is being sent. Let $Color(x,\ k)$ be the $k$ least significant bits of the destination to which record $x$ is being sent, and let a $k$-block be a set of $2^k$ processors with IDs that differ in only the $k$ least significant bits. Then in order for this routing algorithm to operate without collisions, for each value of $i$, no $ig$-block can contain two records $x$ and $y$ where $Color(x,\ ig) = Color(y,\ ig)$.

In order to approximate this perfect balancing of the colors, the records are first permuted. Let $P$ be the smallest prime greater than $n^{\alpha}$ and let $\pi(P,d,e,j) = jd + e \bmod P$. The permutation that is used sends the record from processor $j$ to processor $\pi(P,d,e,j)$, where the parameters $d$, $1 \leq d \leq P-1$, and $e$, $0 \leq e \leq P-1$, are set in the planning stage. The Benes routes which implement these permutations are the cause of the pre-processing requirements. The procedure described above is then used to route records to their destinations. Because the colors are not perfectly balanced, some collisions may occur, and as a result some records may not be delivered to the destinations. These records are then permuted again in order to balance the colors and the above procedure is repeated. A total of $\log^* n$ levels of recursive calls are used, with each level requiring $O(\log n)$ time. In order to obtain the given time bound, the calls at any given level of the recursion overlap according to a fairly complex formula.

## 6 Sorting on the Multi-Butterfly

The multi-butterfly family of interconnection networks was defined by Upfal [19], who proved that computers of this sort can deterministically route an arbitrary permutation in $O(\log n)$ time. The multi-butterfly was studied further by Leighton and Maggs, who simplified some of Upfal's work and also showed that the multi-butterfly possesses a high degree of fault tolerance [9].

The shared key sorting techniques described in this paper lead easily to an $O(\log n \log\log n)$ time, $O(1)$ storage sorting algorithm for sorting $n/\log n$ records on an $n$ processor multi-butterfly. The sorting algorithm uses the deterministic routing capability of the multi-butterfly to perform optimal shared key sorting. Otherwise, the structure of the algorithm is essentially the same as that presented in [6]. The difficulty that forces us to implement a sparse sort (only $n/\log n$ records on an $n$ processor multi-butterfly) is that the multi-butterfly is not recursively decomposable in the same sense as the cube-type computers that we have been considering.

## 7 Open Problems

This work leaves a number of interesting open problems. With respect to Section 4, is there an efficient algorithm for constructing covering sets of polynomial size with respect to one of the known randomized routing algorithms? Can the shared key sorting approach be modified in order to obtain an optimal sorting algorithm for the case $n = p$, or is a multiplicative penalty of $\log\log n$ inherent in the method? Can the rather complex algorithm of Section 5 be substantially simplified?

## 8 Acknowledgments

Thanks to Tom Leighton, Bruce Maggs, Tomasz Radzik and Alex Wang for helpful discussions.

## References

[1] Alok Aggarwal and Ming-Deh A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMS by interconnection networks (preliminary version). In *Proc. 3rd Aegean Workshop on Computing*, pages 339–350, 1988.

[2] Miklos Ajtai, Janos Komlos, and Endre Szemeredi. An O(n log n) sorting network. In *Proc. 15th Annual Symposium on Theory of Computing*, pages 1–9, 1983.

[3] Kenneth E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 307–314, 1968.

[4] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965.

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

[6] Robert Cypher and C. Greg Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proc. 22nd Annual Symposium on Theory of Computing*, 1990. to appear.

[7] A. R. Karlin and E. Upfal. Parallel hashing – an efficient implementation of shared memory. In *Proc. 18th Annual Symposium on Theory of Computing*, pages 160–168, 1986.

[8] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[9] F. T. Leighton and B. M. Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th Annual Symposium on Foundations of Computer Science*, pages 384–389, 1989.

[10] F. T. Leighton, B. M. Maggs, and S. Rao. Universal packet routing algorithms. In *Proc. 29th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[11] David Nassimi and Sartaj Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–107, February 1981.

[12] David Nassimi and Sartaj Sahni. A self-routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers*, C–30(5):332–340, May 1981.

[13] David Nassimi and Sartaj Sahni. Parallel algorithms to set up the Benes permutation network. *IEEE Transactions on Computers*, C-31(2):148–154, February 1982.

[14] David Nassimi and Sartaj Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29:642–667, July 1982.

[15] C. Greg Plaxton. Load balancing, selection and sorting on the hypercube. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, 1989.

[16] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[17] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[18] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, February 1971.

[19] E. Upfal. An $o(\log n)$ deterministic packet routing scheme. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 241–250, 1989.

[20] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, 1981.