Algorithmica (2000) 26: 237–254
DOI: 10.1007/s004539910011

Algorithmica
© 2000 Springer-Verlag New York Inc.

Sorting-Based Selection Algorithms for Hypercubic Networks[1]

P. Berthomé,[2,3] A. Ferreira,[2] B. M. Maggs,[4] S. Perennes,[2,5] and C. G. Plaxton[6]

Abstract. This paper presents several deterministic algorithms for selecting the kth largest record from a set of n records on any n-node hypercubic network. All of the algorithms are based on the selection algorithm of Cole and Yap, as well as on various sorting algorithms for hypercubic networks. Our fastest algorithm runs in $O(\lg n \lg^* n)$ time, very nearly matching the trivial $\Omega(\lg n)$ lower bound. Previously, the best upper bound known for selection was $O(\lg n \lg\lg n)$. A key subroutine in our $O(\lg n \lg^* n)$ time selection algorithm is a sparse version of the Sharesort algorithm that sorts n records using p processors, $p \geq n$, in $O(\lg n(\lg\lg p - \lg\lg(p/n))^2)$ time.

Key Words. Selection, Hypercube, Parallel algorithms.

1. Introduction.

This paper presents several algorithms for solving the selection problem on hypercubic networks. The input to the selection problem is a set S of n records and an integer k. The goal is to find the kth smallest record in S. This problem is also called the order statistics problem. The algorithms in this paper run on the hypercube or on any of its bounded-degree derivatives including the butterfly, cube-connected cycles, and shuffle-exchange network. The fastest runs in $O(\lg n \lg^* n)$ time on an n-node network. (Throughout the paper, we use $\lg n$ to denote $\log_2 n$, and we use $\lg^* n$ to denote the "log-star" function defined by $\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$, where $\lg^{(i)} n$ denotes the ith iterated logarithm of n.) The fastest previously known algorithm ran in $O(\lg n \lg\lg n)$ time [9]. The algorithms use a technique called successive sampling, which was previously used by Cole and Yap [5] to solve the selection problem in an idealized model of computation called the parallel comparison model. The algorithms

[1] Work by the first author was supported by the French CNRS Coordinated Research Program on Parallelism C^3 and the French PRC/GDR MathInfo. Work by the fifth author was supported by NSF Research Initiation Award CCR-9111591, the Texas Advanced Research Program (TARP) under Grant Nos. 93-003658-461 and 91-003658-480, and the NEC Research Institute.

[2] Laboratoire de l'Informatique du Parallélisme, CNRS, Ecole Normale Supérieure de Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France. Afonso.Ferreira@ens-lyon.fr, www.ens-lyon.fr/˜ferreira.
[3] Current address: LRI, Bât 490, Université Paris-Sud, 91405 Orsay Cedex, France. berthome@lri.fr, www.lri.fr/people/berthome.html.
[4] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA. bmm@cs.cmu.edu, www.cs.cmu.edu/˜bmm. This work was performed while the author was at NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, USA.
[5] I3S, CNRS, rue A. Einstein, Sophia Antipolis, 06560 Valbonne, France. sp@essi.fr, www.i3s.unice.fr/˜sp.
[6] Department of Computer Science, University of Texas at Austin, Austin, TX 78712, USA. plaxton@cs.utexas.edu, www.cs.utexas.edu/users/plaxton.

Received March 23, 1994; revised October 30, 1997. Communicated by Chee-Keng Yap.

also use as subroutines sorting algorithms for hypercubic networks due to Nassimi and Sahni [8] and Cypher and Plaxton [6].

1.1. *Hypercubic Networks.* A *hypercube* contains $n = 2^d$ nodes, each of which has a distinct d-bit label (d must be a nonnegative integer). A node labeled $b_0 \cdots b_{d-1}$ is connected by edges to those nodes whose labels differ from $b_0 \cdots b_{d-1}$ in exactly one bit position. An edge connecting two nodes whose labels differ in bit i is called a dimension i edge. Each node has d neighbors, one for each dimension. A *subcube* of the hypercube is formed by fixing the bit values of the labels in some subset of the d dimensions of the hypercube, and allowing the bit values in the other dimensions to vary. In particular, for each subset $j_0, \ldots, j_{k-1}$ of the set of dimensions $\{0, \ldots, d-1\}$, and each set of bit values $v_0, \ldots, v_{k-1}$, there is a dimension-k subcube of the hypercube consisting of the $n/2^k$ nodes whose labels have value v_i in dimension j_i, $0 \leq i < k$, and the edges connecting those nodes.

The nodes in a hypercube represent processors, and the edges represent wires. Each processor has some local memory organized in $O(d)$-bit words. At each time step, a processor can send a word of data to one of its neighbors, receive a word of data from one of its neighbors, and perform a local operation on word-sized operands. In sorting and selection problems, the input consists of a number of $O(1)$-word *records*. Each record has an associated *key* that determines its rank in the entire set of records. We assume throughout that all keys are unique. This may be done without loss of generality, since ties can always be broken in a consistent manner by appending the initial address (processor and memory location) of each record to its key.

All of the algorithms described in this paper use the edges of the hypercube in a very restricted way. At each time step, only the edges associated with a single dimension are used, and consecutive dimensions are used on consecutive steps. Such algorithms are called *normal* [7, Section 3.1.4]. The bounded-degree variants of the hypercube, including the butterfly, cube-connected cycles, and shuffle-exchange network, can all simulate any normal hypercube algorithm with constant slowdown [7, Sections 3.2.3 and 3.3.3]. For simplicity, we describe all of the algorithms in terms of the hypercube.

1.2. *Selection Refinement.* Like most selection algorithms, the algorithms in this paper use a technique called selection refinement. Given a set S of n records and an integer k, $0 \leq k < n$, a *selection refinement* algorithm finds the key with rank k as follows. First, the algorithm computes lower and upper approximations to the desired record. A *lower approximation* to the record of rank k is a record with rank less than or equal to k. An *upper approximation* to the record of rank k is a record with rank greater than or equal to k. Second, the algorithm extracts the subset S' of S consisting of all records between the lower and upper approximations. (The lower and upper approximations are considered to be "good" if $|S'|$ is much smaller than $|S|$.) Third, the algorithm computes an integer k' such that the record with rank k' in S' has rank k in S. Finally, the algorithm recursively finds the element of rank k' in S'.

1.3. *Successive Sampling.* Selection refinement algorithms differ in the method used to find lower and upper approximations. Our algorithm uses a technique called successive sampling. This technique is also used in the algorithm of Cole and Yap [5]. Given a set S of n records and an integer k, $0 \leq k < n$, a *successive sampling* algorithm computes lower and upper approximations as follows. First, the algorithm partitions the set $S = S_0$ into n/s groups of size s (in an arbitrary fashion), and sorts each group. Second, a new set $S_1 \subseteq S$ of nt/s records is formed by taking t evenly spaced records from each group. This sampling process is repeatedly applied to obtain a subset S_2 of S_1, a subset S_3 of S_2, and so on until the set of remaining elements $S' \subseteq S$ is sufficiently small that it can be sorted efficiently. (Different values of the parameters s and t may be used at each "level" of sampling.) Finally, the lower (resp., upper) approximation is chosen to be the largest (resp., smallest) record in S' with rank that is guaranteed (by properties of the successive sampling process) to be less (resp., greater) than or equal to k in S.

1.4. *Previous Work.* The selection problem is closely related to the sorting problem. On the one hand, it is obvious that any sorting algorithm can be used for selection. On the other hand, the performance of selection refinement algorithms depends heavily on the cost of sorting "small" sets of records (i.e., sorting n records using $p \gg n$ processors). For the hypercube, the fastest n-record n-processor sorting algorithm known is the Sharesort algorithm of Cypher and Plaxton [6], which runs in $O(\lg n(\lg\lg n)^2)$ time. (A nonuniform version of the Sharesort algorithm runs in $O(\lg n \lg\lg n)$ time [6].) In addition to Sharesort, we make use of Nassimi and Sahni's sparse enumeration sort [8], which sorts n records on p processors, $p \geq n$, in $O((\lg n \lg p)/(\lg p - \lg n))$ time. (Note that sparse enumeration sort runs in optimal $O(\lg n)$ time if $p \geq n^{1+\varepsilon}$ for some positive constant ε.)

The fastest previously known algorithm for solving the selection problem on a hypercubic network is due to Plaxton and runs in $O(\lg n \lg\lg n)$ time on an n-node network [9]. Of course, the selection problem can also be solved in $O(\lg n(\lg\lg n)^2)$ time using Sharesort. Plaxton also showed that any deterministic algorithm for solving the selection problem on a p-processor hypercubic network requires $\Omega((n/p)\lg\lg p + \lg p)$ time in the worst case [9]. Since the selection problem can be solved in linear time sequentially [3], the lower bound implies that it is not possible to design a deterministic hypercubic selection algorithm with linear speedup. For $n = p$ the lower bound is $\Omega(\lg n)$, which is the diameter of the network.

In [10] Valiant proved an $\Omega(\lg\lg n)$ lower bound on the time to find the largest record in a set of n records using n processors in an idealized model called the parallel comparison model. The lower bound implies a lower bound on the time to select the kth smallest record as well. Valiant also showed how to find the largest record in $O(\lg\lg n)$ time. Cole and Yap [5] then described an $O((\lg\lg n)^2)$ selection algorithm for this model. The running time was later improved to $O(\lg\lg n)$ by Ajtai et al. [1]. The comparisons performed by the latter algorithm are specified by an expander graph, however, making it unlikely that this algorithm can be efficiently implemented on a hypercubic network.

A different set of upper and lower bounds hold in the PRAM models. Beame and Håstad [2] proved an $\Omega(\lg n/\lg\lg n)$ lower bound on the time for selection in the CRCW comparison PRAM using a polynomial number of processors. Vishkin [11] discovered an $O(\lg n \lg\lg n)$ time PRAM algorithm that uses $O(n/\lg n \lg\lg n)$ processors. The algorithm is work-efficient (i.e., exhibits optimal speedup) because the processor–time product is equal to the time, $O(n)$, of the fastest sequential algorithm for this problem. Cole [4] later found an $O(\lg n \lg^* n)$ time work-efficient PRAM algorithm.

1.5. *Outline.* A "basic" selection algorithm that runs in $O(\lg n \lg\lg n)$ time is presented in Section 2. Several faster selection algorithms are presented in the remainder of the paper. Pseudocode for these faster selection algorithms is provided in Section 3. Running times of $O(\lg n \lg^{(3)} n)$ and $O(\lg n \lg^{(4)} n)$ are established in Sections 4 and 5, respectively. An $O(\lg n \lg^* n)$ algorithm is presented in Section 6. This time bound is obtained at the expense of using a nonuniform variant of the Sharesort algorithm [6] that requires a certain amount of preprocessing. Finally, in Section 7 we show how to avoid the nonuniformity introduced in Section 6.

2. An $O(\lg n \lg\lg n)$ Selection Algorithm.

In this section we develop an efficient subroutine for selection refinement based on the parallel comparison model algorithm of Cole and Yap [5]. There are two major differences. First, we use Nassimi and Sahni's sparse enumeration sort [8] instead of a constant time sort (as is possible in the parallel comparison model), and second we obtain a total running time that is proportional to the running time of the largest call to sparse enumeration sort, whereas in the Cole–Yap algorithm, the running time is proportional to the number of sorts, $O(\lg\lg n)$, each of which costs constant time.

As in the Cole–Yap algorithm, the selection refinement algorithm proceeds by successively sampling the given set of records. We define "sample 0" as the entire set of records. At the ith stage of the selection refinement algorithm, $i \geq 0$, a "subsample" is extracted from sample i. This subsample represents sample $i + 1$, and is a proper subset of sample i. Hence the sequence of sample sizes is monotonically decreasing. The sampling process terminates at a value of i for which the ith sample is sufficiently small that it can be sorted in logarithmic time (using sparse enumeration sort). From this final sample, we extract lower and upper approximations to the desired record. Our goal is to obtain "good" upper and lower approximations in the sense that the ranks of our approximations are close to k.

The following approach is used to extract sample $i + 1$ from sample i. First, the records of sample i are partitioned into a number of equal-sized groups, and each group is assigned an equal fraction of the processors. Second, each group of records is sorted using sparse enumeration sort. The number of groups is determined in such a way that the running time of sparse enumeration sort is logarithmic in the group size. This is the case, for example, if sparse enumeration sort is used to sort m^2 records in a subcube with m^3 processors. Letting s denote the group size, the third step is to extract approximately $\sqrt{s}$ uniformly spaced records (i.e., every $\sqrt{s}$th record) from each group. The union of these extracted sets of size $\sqrt{s}$ forms sample $i + 1$. Note that the ratio of the size of sample i to that of sample $i + 1$ is $\sqrt{s}$.

Before proceeding, we introduce a couple of definitions.

DEFINITION 2.1. The rank of a record α in a set S, $\text{rank}(\alpha, S)$, is equal to the number of records in S that are strictly smaller than α. (Note that the record α may or may not belong to the set S.)

DEFINITION 2.2. An r-sample of a set of records S is the subset R of S consisting of those records with ranks in S congruent to 0 modulo r, i.e., $R = \{\alpha \in S \mid \text{rank}(\alpha, S) = ir, 0 \le i < |S|/r\}$.

2.1. *Pseudocode for the Sampling Procedure.* The input to procedure Sample_ℓ below is a set S of records concentrated in a subcube of p processors, $p \ge |S|$. (A set of records X is said to be *concentrated* in a subcube C if each of the $|X|$ lowest-numbered processors of C contains a unique element of X.) The output is a sample (i.e., subset) S' of S. The sample S' is chosen in such a manner that: (i) $|S'| \ll |S|$ and (ii) the record of rank k in the sample has rank approximately $k|S|/|S'|$ in S. (Lemma 2.1 provides precise bounds on the rank properties of S' with respect to S.) The subscript ℓ is drawn from the set $\{0, 1, 2, 3, 4\}$. (In effect, we are defining five slightly different sampling procedures, one corresponding to each subscript value.) For the purposes of Section 2 the reader may assume that $\ell = 0$.

```
Procedure Sample_ℓ(S, p)

1. Partition S into g groups (the parameter g will be defined momentarily)
   of size $s = |S|/g$, assign $p/g$ processors to each group, and sort each
   of the groups in parallel. If $\ell = 0$ or 1, then use sparse enumeration
   sort to accomplish the sorting. If $\ell = 2$, then use Sharesort. (Note that
   Sharesort assumes an input consisting of one record at each processor; if
   $p > |S|$, then we simply use $|S|$ of the $p$ processors.) If $\ell = 3$, then use
   the nonuniform sparse Sharesort algorithm defined in Section 6. If $\ell = 4$,
   then use the uniform sparse Sharesort algorithm defined in Section 7.
   The parameter g is chosen so that the running time of this step (which
   dominates the overall running time of the procedure) is $\Theta(\lg s)$ if $\ell = 0$,
   and $\Theta(\lg p)$ otherwise.
2. Extract a $\sqrt{s}$-sample from each of the g groups.
3. Return the union of these $\sqrt{s}$-samples.
```

2.2. *Analysis of the Sampling Procedure.* Given the rank of a record in the sample returned by a call to $\mathsf{Sample}_\ell(S, p)$, the following lemma provides upper and lower bounds on the rank of that record in S.

LEMMA 2.1. *Let δ, δ', and δ'' denote integers satisfying $0 \le \delta'' \le \delta' \le \delta$. Let X denote a set of 2^δ records, and assume that X is partitioned into $2^{\delta-\delta'}$ sets X_k, $0 \le k < 2^{\delta-\delta'}$, of size $2^{\delta'}$. Let X' denote the union of the $2^{\delta''}$-samples of each of the X_k's. If record α has rank j in set X', then the rank of α in set X lies in the interval*

$$\left(j2^{\delta''} - 2^{\delta-\delta'+\delta''}, j2^{\delta''}\right].$$

PROOF. Let r_k denote the rank of α in the $2^{\delta''}$-sample extracted from set X_k, $0 \le k < 2^{\delta-\delta'}$. Then the rank of α in set X_k lies in the interval $((r_k - 1)2^{\delta''}, r_k 2^{\delta''}]$, and so the rank of α in X belongs to

$$\left(\sum_{0\le k<2^{\delta-\delta'}} (r_k - 1)2^{\delta''}, \sum_{0\le k<2^{\delta-\delta'}} r_k 2^{\delta''}\right],$$

which proves the claim since $j = \sum_{0\le k<2^{\delta-\delta'}} r_k$. □

2.3. *Pseudocode for the Basic Approximate Selection Procedure.* The input to procedure BasicBounds below is a set S of 2^δ records concentrated in a subcube of $p = 2^{\delta+x}$ processors, and integers k_L and k_U in the range 0 to $|S| - 1$. The output is a pair of records (R_L, R_U) such that R_L (resp., R_U) is in S and the rank of R_L (resp., R_U) in S is at most k_L (resp., greater than k_U). Furthermore, R_L (resp., R_U) is chosen so that its rank in S is "close" to k_L (resp., k_U). (To obtain precise bounds on the ranks of R_L and R_U in S, we make use of Lemma 2.3 as in the proof of Theorem 1.) We remark that all calls to BasicBounds in Section 2 satisfy $k_L = k_U$. The reason we have not replaced the two parameters k_L and k_U with a single parameter will become apparent in Section 3.

```
Procedure BasicBounds($S$, $p$, $k_L$, $k_U$)
1. Set $i$ to 0 and set $S_0$ to $S$.
2. While $|S_i|^{3/2} > p$, set $S_{i+1}$ to Sample$_0(S_i, p)$ and increment $i$.
3. Sort $S_i$ in $O(\lg |S_i|)$ time using sparse enumeration sort.
4. Determine records $R_L$ and $R_U$ as in the proof of Theorem 1 below. (Set
   $k = k_L$ when determining $R_L$, and set $k = k_U$ when determining $R_U$.)
5. Return $(R_L, R_U)$.
```

2.4. *Analysis of the Basic Approximate Selection Procedure.* The input to our algorithm is a set S_0 of 2^δ elements concentrated in a subcube C of size $2^{\delta+x}$. The factor by which the size of the subcube exceeds the size of S_0, 2^x, is called the *excess processor ratio*. In the first iteration, the records in S_0 are partitioned into $2^{\delta-2x}$ groups of size 2^{2x} and 2^{3x} processors are assigned to each group. Each group is then sorted in $O(x)$ time using sparse enumeration sort, and a 2^x-sample is taken from each group. The samples from all the groups are combined to form a new set S_1 containing $2^{\delta-x}$ elements. In general, after $i - 1$ iterations, a set S_{i-1} of $2^{\delta-x(2^{i-1}-1)}$ records remain. In the ith iteration, set S_i is formed by partitioning the records of S_{i-1} into $g_{i-1} \overset{\text{def}}{=} 2^{\delta-x(3\cdot 2^{i-1}-1)}$ groups of size 2^{x2^i} and then extracting a $2^{x2^{i-1}}$-sample from each group. Since the ratio of the number of processors in C to $|S_{i-1}|$ is $2^{x2^{i-1}}$, we can assign $2^{3x2^{i-1}}$ processors to each group of size 2^{x2^i}, and each group can be sorted in $O(x2^i)$ time using sparse enumeration sort.

LEMMA 2.2. *The time required for procedure* BasicBounds *to compute sets* S_0 *through* S_i *is* $O(x2^i)$.

PROOF. The time is $\sum_{1\le j\le i} O(x2^j) = O(x2^i)$. □

LEMMA 2.3. *Let record α have rank j in set S_i, for some $i \geq 1$. Then the rank of α in set S_{i-1} lies in the interval*

$$\left(j2^{x2^{i-1}} - 2^{\delta - x(2^i - 1)}, j2^{x2^{i-1}}\right].$$

PROOF. A straightforward application of Lemma 2.1, with the variables δ, δ', and δ'' of the lemma replaced by the expressions $\delta - x(2^{i-1} - 1)$, $x(3 \cdot 2^{i-1} - 1)$, and $x2^{i-1}$, respectively. □

LEMMA 2.4. *Let record α belong to S_i and let j denote the rank of α in S_i, for some $i \geq 1$. Then the rank of α in S_0 lies in the range*

$$\left(j2^{x(2^i-1)} - \sum_{0 \leq k < i} 2^{\delta - x2^k}, j2^{x(2^i-1)}\right].$$

PROOF. The proof is by induction on i. The base case, $i = 1$, is a special case of Lemma 2.3.

Now we assume that the claim holds inductively. Suppose that record α has rank j in S_i. Then, by Lemma 2.3, the rank of α in the set S_{i-1} lies in the interval

$$\left(j2^{x2^{i-1}} - 2^{\delta - x(2^i - 1)}, j2^{x2^{i-1}}\right].$$

Applying the induction hypothesis, the rank of record α in the set S_0 is strictly greater than

$$\left(j2^{x2^{i-1}} - 2^{\delta - x(2^i-1)}\right) 2^{x(2^{i-1}-1)} - \sum_{0 \leq k < i-1} 2^{\delta - x2^k},$$

which is equal to $j2^{x(2^i-1)} - \sum_{0 \leq k < i} 2^{\delta - x2^k}$, and at most

$$j2^{x2^{i-1}} 2^{x(2^{i-1}-1)} = j2^{x(2^i-1)},$$

as required. □

THEOREM 1. *Let S denote a set of 2^δ records concentrated in a subcube C of size $2^{\delta+x}$ (x integer, $4 \leq x \leq \delta/2$), and let k be an integer, $0 \leq k < 2^\delta$. Then in $O(\delta)$ time it is possible to compute a subset S' of S and an integer k' such that the following conditions are satisfied*:

(i) $|S'| = 2^{\delta - x + 3}$,
(ii) $0 \leq k' < |S'|$,
(iii) *the record of rank k' in S' has rank k in S, and*
(iv) *S' is concentrated in C.*

PROOF. A call to BasicBounds (S, p, k, k) produces a sequence of sets S_0 through S_i where $|S_i| = 2^{\delta - x(2^i-1)}$ and, as we shall see, $(\delta + x)/3 \leq x2^i \leq \delta + 1$. By Lemma 2.2, the time required to compute S_i is $O(x2^i) = O(\delta)$.

Next, the records in S_i are sorted using sparse enumeration sort. There are $2^{\delta - x(2^i - 1)}$ records and $2^{\delta + x}$ processors; hence, the excess processor ratio is 2^{x2^i}. We choose i to be the smallest value such that the excess processor ratio is at least the square root of the number of records, $x2^i \geq \frac{1}{2}(\delta - x(2^i - 1))$. Solving for i yields $2^i \geq (\delta + x)/3x$ and $i = \lceil \lg((\delta + x)/3x) \rceil$. The time for sparse enumeration sort is $O(\delta + x) = O(\delta)$.

We would now like to find two records, R_L and R_U in S_i, with ranks r_L and r_U in S_0, such that k belongs to the interval $[r_L, r_U)$ and $r_U - r_L$ is small. In the following, let $A = 2^{x(2^i - 1)}$ and $B = 2^{\delta - x + 1}$. By Lemma 2.4, the key with rank j in S_i lies in the interval $(jA - B, jA]$ in S_0. Let $j_L = \lfloor k/A \rfloor$, $j_U = \lceil (k + B)/A \rceil$, and R_L and R_U be the records in S_i with ranks j_L and j_U in S_i, respectively. (If k is sufficiently close to $|S_0|$, we can have $j_U \geq |S_i|$ and hence the record R_U is not well defined; in this case, we can set R_U to a dummy record that is greater than any record in S_i, and set r_U to $|S_i|$.) Then $j_L A - B < r_L \leq j_L A \leq k$, and $k \leq j_U A - B < r_U \leq j_U A$. Note that $B = bA$, with b an integer, and there exist integers α and β, $0 \leq \beta < A$, such that $k = \alpha A + \beta$. Hence $j_U = \alpha + b + \lceil \beta/A \rceil \leq \alpha + b + 1$, $j_L = \alpha$, and

$$\begin{aligned} r_U - r_L &\leq (j_U - j_L)A + B \\ &\leq (\alpha + 1 + b - \alpha)A + B \\ &= A + 2B. \end{aligned}$$

We set S' to be the set of at most $A + 2B$ records in S_0 with ranks in $[r_L, r_U)$. Note that, given records R_L and R_U, it is straightforward to identify and concentrate the set S' in $O(\delta)$ time. For $A \leq B$, we have $|S'| \leq 3B < 2^{\delta - x + 3}$. For $i = \lceil \lg((\delta + x)/3x) \rceil$, the inequality $A \leq B$ is satisfied since

$$\begin{aligned} x(2^i - 1) &\leq x\left[2 \cdot \left(\frac{\delta + x}{3x}\right) - 1\right] \\ &= (\delta - x) - \frac{\delta - 2x}{3} \\ &\leq \delta - x, \end{aligned}$$

where the last inequality follows from the assumption that $x \leq \delta/2$. The value of k' is determined by finding the rank of k in S', which can easily be done in $O(\delta)$ time. □

2.5. *Pseudocode for the Basic Selection Algorithm.* The input to procedure BasicSelect below is a set S of records concentrated in a subcube of $p \geq |S|$ processors, and an integer k in the range 0 to $|S| - 1$. The output is the record of rank k in S.

```
Procedure BasicSelect($S$, $p$, $k$)

1. If $|S|^{3/2} \leq p$, then sort $S$ in $O(\lg |S|)$ time using sparse enumeration sort
   and return the record of rank $k$ in $S$.
2. Set $(R_L, R_U)$ to BasicBounds($S$, $p$, $k$, $k$).
3. Set $r_L$ (resp., $r_U$) to the rank of $R_L$ (resp., $R_U$) in $S$.
4. Let $S'$ denote the $r_U - r_L$ records in $S$ with ranks (in $S$) in the interval
   $[r_L, r_U)$, and set $k'$ to $k - r_L$.
5. Return BasicSelect($S'$, $p$, $k'$).
```

2.6. *Analysis of the Basic Selection Algorithm.* We define $T(\delta, x)$ as the worst-case (over all k) running time of a call to BasicSelect(S, p, k) with $|S| = 2^\delta$ and $p = 2^{\delta+x}$. Note that, for $\delta' \le \delta$ and $x' \ge x$, we have $T(\delta', x') \le T(\delta, x)$; in what follows, we occasionally make implicit use of this trivial inequality.

Theorem 1 implies that

$$T(\delta, x) \le T(\delta - x + 3, 2x - 3) + O(\delta) \tag{1}$$

for $4 \le x \le \delta/2$. For $x > \varepsilon\delta$, where ε denotes an arbitrarily small positive constant, sparse enumeration sort implies that $T(\delta, x) = O(\delta)$. We are interested in obtaining an upper bound for $T(\delta, 0)$. Note that $T(\delta, 0) = \Theta(T(\delta, 4))$, since we can simulate a $2^{\delta+4}$-processor hypercube on a 2^δ-processor hypercube with only constant factor slowdown. By iterating the recurrence of (1), we can obtain an upper bound for $T(\delta, 4)$. For $\delta \ge 8$, one application of the recurrence gives $T(\delta, 4) \le T(\delta - 1, 5) + c\delta$ for some constant $c > 0$. For $\delta \ge 11$ we can apply the recurrence again to obtain $T(\delta, 4) \le T(\delta - 3, 7) + 2c\delta$. In general, for $\delta \ge 2^i + 2^{i-1} + 5$, we can apply the recurrence i times to obtain $T(\delta, 4) \le T(\delta - 2^i + 1, 2^i + 3) + ic\delta$ (this claim is easily verified by induction on i). For $\delta \ge 20$, we can set $i = \lfloor \lg \delta \rfloor - 1$ to obtain $T(\delta, 4) \le T(\lfloor 3\delta/4 \rfloor + 1, \lceil \delta/4 \rceil + 3) + O(\delta \lg \delta) = O(\delta \lg \delta)$. Hence $T(d, 0) = O(d \lg d) = O(\lg n \lg\lg n)$, and we have proved the following theorem.

THEOREM 2. *Any call to* BasicSelect(S, p, k) *with* $n = |S| = p$ *runs in* $O(\lg n \lg\lg n)$ *time.*

Procedure BasicSelect is essentially equivalent to a selection algorithm described by Plaxton in [9]. Prior to this algorithm, the best bounds known for selection on the hypercube were given by sorting algorithms.

3. Pseudocode for Several Faster Selection Algorithms.

In this section we present the procedures Bounds$_\ell$ and Select$_\ell$, which will be used to establish improved time bounds for selection in Sections 4–7.

The input to procedure Bounds$_\ell$ below is a set S of 2^δ records concentrated in a subcube of $p = 2^{\delta+x}$ processors, and an integer k in the range 0 to $|S| - 1$. The output is a pair of records (R_L, R_U) such that R_L (resp., R_U) is in S and the rank of R_L (resp., R_U) in S is at most k (resp., greater than k). Furthermore, R_L (resp., R_U) is chosen so that its rank in S is "close" to k. (See the analysis of Sections 4–7 for precise bounds.) The subscript ℓ is drawn from the set $\{0, 1, 2, 3, 4\}$. (In effect, we are defining five slightly different procedures, one corresponding to each subscript value.) Our interest lies primarily with subscript values other than 0 since the procedure Bounds$_0$ is essentially equivalent to the procedure BasicBounds of Section 2.3.

```
Procedure Bounds$_\ell(S, p, k)$

1. Set $S'$ to Sample$_\ell(S, p)$.
2. Set $k_L$ (resp., $k_U$) to the largest (resp., smallest) integer such that the record
```

of rank k_L (resp., k_U) in S' is guaranteed by Lemma 2.1 to have rank less than or equal to k in S.

3. Return BasicBounds(S', p, k_L, k_U).

The procedure Select$_\ell(S, p, k)$ is identical to the procedure BasicSelect(S, p, k) of Section 2.5 except that the call to BasicBounds(S, p, k, k) in Step 2 is replaced by a call to Bounds$_\ell(S, p, k)$.

4. An $O(\lg n \lg^{(3)} n)$ Selection Algorithm.

Throughout this section, we refer to the $O(\lg n \lg\lg n)$ selection algorithm of Section 2 as the "basic" algorithm. Our present goal is to improve the running time of the basic algorithm to $O(d \lg\lg d) = O(\lg n \lg^{(3)} n)$ by a simple modification. The basic algorithm consists of $O(\lg d)$ applications of the $O(d)$ selection refinement subroutine corresponding to Theorem 1. We view each application of the selection refinement subroutine as a "phase" of the basic algorithm. In order to improve the performance of the basic algorithm, we augment each phase in the following manner: Before applying the selection refinement subroutine, we partition the remaining data into subcubes of dimension δ', sort these subcubes completely, and extract a $2^{\delta''}$-sample from each subcube, where $\delta'' = \lceil \delta'/2 \rceil$. These samples are then passed on to the selection refinement subroutine for successive sampling.

The parameter δ' is chosen in such a way that, given the excess processor ratio available at that particular phase, the sort can be completed in $O(d)$ time. The motivation for defining δ' in this manner is to balance the time spent on the initial sort with the $O(d)$ running time of the selection refinement subroutine. Sparse enumeration sort is used to perform the initial sort in each phase.

The modified basic selection algorithm described above corresponds to procedure Select$_1$ of Section 3. The goal of the present section is to prove that any call to Select$_1(S, p, k)$ with $n = |S| = p$ runs in $O(\lg n \lg^{(3)} n)$ time (see Theorem 3 below).

We now analyze the performance of each phase in greater detail. Before the phase, let S denote the set of remaining records, assume that $|S| = 2^\delta$, and assume that the excess processor ratio is 2^x, $x \geq 0$. At the beginning of the phase, we partition S into $2^{\delta-\delta'}$ sets of size $2^{\delta'}$, and sort each set in a subcube of dimension $\delta' + x$. We then extract a $2^{\delta''}$-sample from each sorted set, where $\delta'' = \lceil \delta'/2 \rceil$. Let S' denote the set of $2^{\delta-\delta''}$ records in the union of all of these samples. By Lemma 2.1, the key of rank j in S' has rank in the interval

$$(j2^{\delta''} - 2^a, j2^{\delta''}]$$

in S, where $a = \delta - \delta' + \delta''$. Accordingly, we can obtain a lower approximation for the kth record in S by computing a lower approximation (via Theorem 1) for the $\lfloor k2^{-\delta''} \rfloor$th record in S'. Similarly, we can obtain an upper approximation for the kth record in S by computing an upper approximation for the $(\lceil k2^{-\delta''} \rceil + 2^{\delta-\delta'})$th record in S'.

By Theorem 1, in $O(\delta)$ time we can determine a set of at most 2^b records with contiguous ranks in S' that contains any desired rank, where $b = \delta - x - 2\delta'' + 3$. (To see this, apply Theorem 1 with the variables δ and x of the theorem replaced by the expressions $\delta - \delta''$ and $x + \delta''$, respectively.) In particular, we can obtain a lower approximation for the record of rank k' in S' that has rank strictly greater than $k' - 2^b$, and we can obtain an upper approximation with rank strictly less than $k' + 2^b$. Thus, in $O(\delta)$ time, we can determine:

(i) a lower approximation to the record of rank $\lfloor k2^{-\delta''} \rfloor$ in S' with rank strictly greater than $\lfloor k2^{-\delta''} \rfloor - 2^b$ in S', and
(ii) an upper approximation to the record of rank $\lceil k2^{-\delta''} \rceil + 2^{\delta-\delta'}$ in S' with rank strictly less than $\lceil k2^{-\delta''} \rceil + 2^{\delta-\delta'} + 2^b$ in S'.

With the bounds of the preceding paragraph, the aforementioned lower and upper approximations represent, respectively:

(i) a lower approximation to the record of rank k in S with rank strictly greater than $k - 2^{\delta''} - 2^a - 2^{b+\delta''}$ in S, and
(ii) an upper approximation to the record of rank k in S with rank strictly less than $k + 2^{\delta''} + 2^a + 2^{b+\delta''}$ in S.

Hence, within the same time bound we can identify a set of at most

$$z \overset{\text{def}}{=} 2 \cdot \left(2^{\delta''} + 2^a + 2^{b+\delta''}\right)$$

records with contiguous ranks in S and which contains the record of rank k in S. Observe that $a \geq \delta''$ and $a + 3 \geq b + \delta''$ (recall that $a = \delta - \delta' + \delta''$). Hence $z \leq 2^{a+5}$.

Note that the initial application of sparse enumeration sort runs in $O(\delta)$ time if $\delta' = O(\sqrt{\delta x})$, since the running time of sparse enumeration sort is $O(\delta'(\delta' + x)/x) = O((\delta')^2/x)$. Accordingly, we set $\delta' = \left\lceil c\sqrt{\delta} \right\rceil$ for some positive constant c. Note that, for $x \geq 1$, $c > 1$, and δ sufficiently large, $2^{\delta-\delta'+\delta''+5} \leq 2^{\delta - \lceil \sqrt{\delta x} \rceil}$, and hence $z \leq 2^{\delta - \lceil \sqrt{\delta x} \rceil}$. As in Section 2.6 (where in fact we assumed that $x \geq 4$), we may assume that $x \geq 1$ without loss of generality.

Hence, the foregoing discussion has established the recurrence

$$\begin{aligned} T(\delta, x) &\leq T\left(\delta - \left\lceil \sqrt{\delta x} \right\rceil, x + \left\lceil \sqrt{\delta x} \right\rceil\right) + c'\delta \\ &\leq T\left(\delta, \left\lceil \sqrt{\delta x} \right\rceil\right) + c'\delta \end{aligned}$$

for $1 \leq x \leq \delta/2$ and some constant $c' > 0$. For $x \geq 1$ and $\left\lceil \sqrt{\delta x} \right\rceil \leq \delta/2$ we can iterate this recurrence to obtain

$$\begin{aligned} T(\delta, x) &\leq T\left(\delta, \left\lceil \sqrt{\delta \left\lceil \sqrt{\delta x} \right\rceil} \right\rceil\right) + 2c'\delta \\ &\leq T\left(\delta, \left\lceil \delta^{3/4} x^{1/4} \right\rceil\right) + 2c'\delta. \end{aligned}$$

More generally, for $x \geq 1$ and $\left\lceil \delta^{1-2^{-i}} x^{2^{-i}} \right\rceil \leq \delta/2$, $i \geq 0$, we can apply the recurrence

i times to obtain

$$\begin{aligned} T(\delta, x) &\le T\left(\delta, \left\lceil \delta^{1-2^{-i}} x^{2^{-i}} \right\rceil\right) + ic'\delta \\ &\le T\left(\delta, \left\lceil \delta^{1-2^{-i}} \right\rceil\right) + ic'\delta. \end{aligned}$$

It is straightforward to verify that the recurrence can be applied $\lg\lg\delta + O(1)$ times, at which point we have

$$T(\delta, x) \le T(\delta, y) + O(\delta \lg\lg\delta)$$

for some y with $\delta/2 < y < \delta$. Sparse enumeration sort implies that $T(\delta, y) = O(\delta)$ and hence $T(d, 0) = O(d \lg\lg d) = O(\lg n \lg^{(3)} n)$. We have thus proved the following theorem.

THEOREM 3. *Any call to* Select$_1(S, p, k)$ *with* $n = |S| = p$ *runs in* $O(\lg n \lg^{(3)} n)$ *time.*

5. An $O(\lg n \lg^{(4)} n)$ Algorithm. We can improve the time bound achieved in Section 4 by making use of the Sharesort algorithm of Cypher and Plaxton [6]. Several variants of that algorithm exist; in particular, detailed descriptions of two versions of Sharesort may be found in [6]. Both of these variants are designed to sort n records on an n-processor hypercubic network. The first algorithm runs in $O(\lg n(\lg\lg n)^3)$ time and the second algorithm, which is somewhat more complicated, runs in $O(\lg n(\lg\lg n)^2)$ time. The selection algorithm of this section makes use of Sharesort as a subroutine. For this purpose, either of the aforementioned variants of Sharesort may be used; this choice does not affect the overall running time by more than a constant factor. For the sake of concreteness, in the calculations that follow we assume that the simpler $O(\lg n(\lg\lg n)^3)$ algorithm is used.

The only change to the algorithm of Section 4 is that, in the initial phase, Sharesort is used instead of sparse enumeration sort to perform the initial $O(d)$-time sort. With Sharesort, we can afford to set $\delta' = \Theta(d/(\lg d)^3)$, which is substantially larger than the $\Theta(\sqrt{d})$ bound achievable with sparse enumeration sort. For all phases subsequent to the first phase, however, we make use of sparse enumeration sort. The reason is that, in the absence of a suitable processor–time tradeoff for the Sharesort algorithm, sparse enumeration sort is actually faster than Sharesort after the first phase (due to the large excess processor ratio created by the first phase). In Section 6 we obtain an even faster selection algorithm by developing and applying an effective processor–time tradeoff for the Sharesort algorithm.

The selection algorithm described above corresponds to a hybrid of the procedures Select$_1$ and Select$_2$ of Section 3. (The top-level selection call is to Select$_2$, but the recursive selection calls are to Select$_1$.) We refer to this hybrid selection procedure as Select$'_2$.

In order to analyze the running time of procedure Select$'_2$, we repeat the analysis of Section 4, but with δ' set to $\Theta(d/(\lg d)^3)$ in the first phase. The first phase establishes the inequality

$$T(d, 0) \le T\left(d, \left\lceil d/(\lg d)^3 \right\rceil\right) + O(d).$$

Now $d/(\lg d)^3 = d^{1-2^{-i}}$ with $i = \lg\lg d - \lg^{(3)} d - O(1)$. Hence, the recurrence of Section 4 implies that $T(d, \lceil d/(\lg d)^3 \rceil) = O(d \lg^{(3)} d)$. Thus $T(d, 0) = O(d \lg^{(3)} d) = O(\lg n \lg^{(4)} n)$, and we have proved the following theorem.

THEOREM 4. *Any call to* $\mathsf{Select}_2'(S, p, k)$ *with* $n = |S| = p$ *runs in* $O(\lg n \lg^{(4)} n)$ *time.*

6. A Nonuniform $O(\lg n \lg^* n)$ Algorithm.

The improvement described in Section 5 resulted from applying Sharesort instead of sparse enumeration sort at the beginning of the first phase. Note, however, that all of the phases (including the first) continue to make extensive use of sparse enumeration sort. The calls to sparse enumeration sort made by each of the algorithms defined thus far may be partitioned into two classes: (i) those calls made within applications of Theorem 1, and (ii) those calls used to perform an $O(d)$-time sort (actually, a set of parallel $O(d)$-time sorts) before applying Theorem 1. The algorithm of Section 2 contains only calls of Type (i), since each phase consists solely of an application of Theorem 1. The algorithm of Section 4 contains both Type (i) and Type (ii) calls, since each phase consists of an $O(d)$-time sort followed by an application of Theorem 1. The algorithm of Section 5 is the same as the algorithm of Section 4, except that the Type (ii) call of the first phase is replaced with a call to Sharesort (causing the number of phases to be substantially reduced).

Could we obtain an even faster selection algorithm than that of Section 5 by replacing some or all of the remaining calls to sparse enumeration sort with calls to Sharesort? With regard to the Type (i) calls, the answer is no. Even if the Type (i) sorts were performed in optimal logarithmic time, the reduction in data (i.e., relevant records) between successive phases would not be improved significantly. The reason is that the amount of data that "survives" to the next phase is predominantly determined by the size of the subcubes sorted in the Type (ii) sorts. Thus, in all of the algorithms described in this paper, we continue to make use of sparse enumeration sort to perform all of the sorts within applications of Theorem 1.

Now we consider the Type (ii) calls. All of these calls to sparse enumeration sort will in fact be replaced with calls to a more efficient sorting algorithm in order to obtain the $O(\lg n \lg^* n)$ time bound. Unfortunately, we cannot obtain such a bound by merely replacing all of the Type (ii) calls to sparse enumeration sort with calls to one of the single-item-per-processor variants of Sharesort. Instead, we proceed by developing a time–processor tradeoff for Sharesort, and then using the resulting algorithm, a sparse Sharesort, to perform all of the Type (ii) sorts.

THEOREM 5. *Let n records be concentrated in a subcube of a p-processor hypercubic network, $p \geq n$. There exists a nonuniform deterministic algorithm for sorting these records in time*

$$O(\lg n(\lg\lg p - \lg\lg(p/n))). \tag{2}$$

PROOF. As indicated in Section 5, there are a number of variants of the Sharesort algorithm of Cypher and Plaxton [6]. These algorithms differ solely in the way that the so-called *shared key sorting* subroutine is implemented. The shared key sorting problem represents a restricted version of the sorting problem; a formal definition of the shared key sorting problem is not needed in this paper, and so is not given. All variants of Sharesort make use of precisely the same recursive framework to reduce the problem of sorting to that of shared key sorting.

Perhaps the simplest variant of Sharesort runs in $O(\lg n \lg\lg n)$ time and relies upon an optimal logarithmic time shared key sorting subroutine. This particular result is mentioned in the original Sharesort paper [6] and more fully described by Leighton [7, Section 3.5.3]. Although it is the fastest of the Sharesort variants, this sorting algorithm suffers from the disadvantage that it is nonuniform. From the point of view of a user who would like to run this sorting algorithm on a particular hypercube of dimension d, what this nonuniformity implies is that a "setup" routine must be executed when the machine is first configured in order to generate a version of the algorithm that is capable of efficiently sorting any subcube of dimension less than or equal to d. Note that the setup routine need only be executed once in the lifetime of the machine (and not once per sort) and so this deficiency may not be considered overly severe. Unfortunately, the most efficient deterministic algorithms currently known for performing the setup task run in time that is exponential in n.

We will establish the validity of (2) by developing a time–processor tradeoff for the $O(\lg n \lg\lg n)$ time, nonuniform variant of Sharesort.

As mentioned above, all variants of Sharesort are based on a particular system of recurrences. At the highest level, sorting is performed recursively via *high-order merging* (i.e., merging n^{ε} sorted lists of length $n^{1-\varepsilon}$ for some constant ε, $0 < \varepsilon < 1$). The running time of Sharesort is dominated by the time required for high-order merging, which is itself performed recursively. Let $M(x)$ denote the task of merging x sorted lists of length x^4. One possible recurrence for performing the merge is (minor technical details related to integrality constraints are dealt with in [6] and are not addressed here)

$$M(n^{1/5}) \le M(n^{4/45}) + M(n^{1/9}) + O(\lg n) + SKS(n), \tag{3}$$

where $SKS(n)$ denotes the time required to solve the shared key sorting problem. (To justify the preceding recurrence, apply (1) of [6] with $a = \frac{1}{5}\lg n$, $b = 4a$, $a' = \frac{4}{45}\lg n$, $b' = 4a'$, $a'' = \frac{1}{9}\lg n$, $b'' = 4a''$, and observe that the additive $O(a \lg a)$ term corresponds to the sum of $O(a)$ and the cost of a call to the shared key sorting subroutine.) For the $O(\lg n \lg\lg n)$ time, nonuniform variant of Sharesort, $SKS(n) = O(\lg n)$, and so the $SKS(n)$ term essentially disappears from the recurrence of (3). In order to obtain the time bound of (2), we make use of additional processors in the following simple way: whenever a merging problem of the form $M(x)$ arises and x^5 is less than the excess process ratio, we apply x^{10} processors to solve that merging subproblem in optimal $O(\lg x)$ time using sparse enumeration sort. A straightforward analysis shows that this modification to the $O(\lg n \lg\lg n)$ algorithm of Sharesort yields the sorting time bound of (2). □

We modify the $O(\lg n \lg^{(3)} n)$ algorithm of Section 4 by replacing all of the Type (ii) calls to sparse enumeration sort with calls to the sorting algorithm of Theorem 5. The resulting selection algorithm corresponds to procedure Select$_3$ of Section 3.

In order to analyze the performance of procedure Select$_3$, one may simply repeat the analysis of Section 4, setting δ' to $\lceil c\delta/(\lg\delta - \lg x)\rceil$ for some sufficiently large positive constant c. Doing this, we obtain the recurrence

$$T(\delta, x) \le T\left(\delta, \left\lceil \frac{\delta}{\lg\delta - \lg x} \right\rceil\right) + c'\delta$$

for $1 \le x \le \delta/2$ and some constant $c' > 0$. For $x \ge 1$ and $\left\lceil \delta/\lg^{(i)}(\lg\delta - \lg x)\right\rceil \le \delta/2$, $i \ge 0$, we can apply the recurrence i times to obtain

$$T(\delta, x) \le T\left(\delta, \left\lceil \frac{\delta}{\lg^{(i)}\delta} \right\rceil\right) + ic'\delta.$$

In general, the recurrence can be applied $\lg^* d + O(1)$ times, and we find that $T(d, 0) = O(d\lg^* d) = O(\lg n \lg^* n)$. We have thus proved the following theorem.

THEOREM 6. *Any call to* Select$_3(S, p, k)$ *with* $n = |S| = p$ *runs in* $O(\lg n \lg^* n)$ *time.*

7. A Uniform $O(\lg n \lg^* n)$ Selection Algorithm.

In Section 6, we proved the existence of an algorithm for selection that runs in $O(\lg n \lg^* n)$ time. However, as indicated in Section 6, that algorithm is nonuniform because it makes use of a nonuniform version of the Sharesort algorithm. In the present section we establish the existence of a uniform selection algorithm with the same asymptotic complexity as the algorithm of Section 6.

The version of the Sharesort algorithm employed in Section 6 makes use of a nonuniform shared key sorting subroutine. The running time of the nonuniform shared key sorting subroutine is $O(\lg n)$, which is easily seen to be optimal. The fastest known uniform version of the shared key sorting subroutine takes $O(\lg n \lg\lg n)$ time, which leads to an $O(\lg n(\lg\lg n)^2)$ running time for the corresponding uniform variant of Sharesort [6]. In the following we show how to adapt this uniform version of Sharesort to obtain a uniform version of a "sparse" Sharesort, that is, an algorithm for sorting n records on $p \ge n$ processors. We express the running time of SparseSharedKeySort in terms of the two parameters n and p.

THEOREM 7. *Let n records be concentrated in a subcube of a p-processor hypercubic network, $p \ge n$. There exists a uniform deterministic algorithm for sorting these records in time*

$$O\big(\lg n(\lg\lg p - \lg\lg(p/n))^2\big)\,. \tag{4}$$

PROOF. To obtain the result, we show how to improve the time complexity of the shared key sorting procedure when we have a significant number of "extra" processors (i.e., $p \gg n$). Let $SSKS(n, p)$ be the time needed to solve the following *sparse shared key sorting problem*: Perform 2^b identical sorts of lists of size 2^a, with a and b such that $a + b = n$ and $b - a/2 = \Theta(a)$, on a hypercube of size p, $p \ge n$. By a similar analysis as that provided in Section 6, the theorem follows if we can prove that

$$SSKS(n, p) = O(\lg n(\lg\lg p - \lg\lg(p/n)))\,. \tag{5}$$

We modify the SharedKeySort algorithm of [6] to obtain a "SparseSharedKeySort" routine satisfying (5). Because the complete description of the SharedKeySort algorithm is quite lengthy, we content ourselves with a description of the differences between SparseSharedKeySort and SharedKeySort. Fortunately these differences are minor.

We begin by observing that SharedKeySort consists of a call to subroutine PlanRoute followed by a call to subroutine DoRoute [6, Section 7.1]. For the case $n = p$ considered in [6], each of these two subroutines runs in $O(\lg n \lg\lg n)$ time. We now argue that each of these subroutines can be generalized to run in $O(\lg n(\lg\lg p - \lg\lg(p/n)))$ time for $p \geq n$, which implies the desired bound of (5).

We consider subroutine DoRoute first, since it is simpler to deal with than PlanRoute. Looking at the three-parameter recurrence for the running time of DoRoute appearing in Section 7.3 of [6], we observe that: (i) the third parameter does not affect the running time and hence can be ignored, and (ii) we can assume without loss of generality that the first two parameters are equal since they are equal in every recursive call. With the preceding observations, we find that the running time of DoRoute is upper-bounded by a recurrence of the form

$$T(n) \leq 2T(O(\sqrt{n}\lg n)) + O(\lg n), \tag{6}$$

where the parameter n above corresponds to 2^{a+b} (which can be assumed to be equal to 2^{2a}, by observation (ii) above) in the recurrence of Section 7.3 of [6]. Note that (6) is very similar to (3); in each case the additive term is $O(\lg n)$ and the sum of the exponents associated with the "recursive" terms on the right-hand side is equal to the exponent appearing on the left-hand side (i.e., $\frac{1}{5} = \frac{4}{45} + \frac{1}{9}$ in (3) and $1 = \frac{1}{2} + \frac{1}{2}$ in (6)). (The reader may wonder whether it is significant that the argument of T on the right-hand side of (6) is $O(\sqrt{n}\lg n)$ and not simply $\sqrt{n}$; it is easy to argue that the O-bound implied by the recurrence is the same in either case.) In fact, the same technique that we used in conjunction with (3) in the proof of Theorem 5 can now be used to modify DoRoute to obtain the desired $O(\lg n(\lg\lg p - \lg\lg(p/n)))$ time bound; namely, we cut off the recurrence and apply sparse enumeration sort as soon as the excess processor ratio is polynomial in the input size (so that sparse enumeration sort runs in logarithmic time). (We are free to replace any call to DoRoute with a call to a sorting routine since DoRoute implements a restricted type of permutation route, and a sorting routine can be use to route an arbitrary permutation.)

It remains to consider the subroutine PlanRoute. Looking at the three-parameter recurrence for the running time of PlanRoute appearing in Section 7.2 of [6], we observe that: (i) the third parameter does not affect the running time and hence can be ignored, and (ii) we can assume without loss of generality that the first two parameters are equal since they are equal in every recursive call. With the preceding observations, we find that the running time of PlanRoute is upper-bounded by a recurrence of the form

$$T(n) \leq T(O(\sqrt{n}\lg n)) + O(\lg n \lg\lg n), \tag{7}$$

where the parameter n above corresponds to 2^{a+b} (which can be assumed to be equal to 2^{2a}, by observation (ii) above) in the recurrence of Section 7.2 of [6]. This recurrence solves to give $T(n) = O(\lg n \lg\lg n)$. Because the overhead term associated with the first level of the recurrence is also $O(\lg n \lg\lg n)$, the technique of cutting off the recurrence at some depth (i.e., the technique used in conjunction with the DoRoute recurrence above) cannot give more than a constant factor improvement in the upper bound. To obtain the desired $O(\lg n(\lg\lg p - \lg\lg(p/n)))$ time bound, we instead argue that PlanRoute can be generalized to the case of $p \geq n$ processors in such a manner that the $\lg\lg n$ factor appearing within the additive term of (7) becomes $\lg\lg p - \lg\lg(p/n)$ (the resulting recurrence is easily solved and yields the desired time bound).

In order to see that this $\lg\lg n$ factor can in fact be replaced by $\lg\lg p - \lg\lg(p/n)$, we need to understand how the $\lg\lg n$ factor arises. A cursory examination of the PlanRoute algorithm [6, Section 7.2] reveals that the $\lg\lg n$ factor corresponds to the number of "classes" into which the set of n input records is partitioned. The partitioning into classes is performed through $\lg\lg n$ calls to the subroutine Balance. Each successive call to Balance runs in $O(\lg n)$ time and acts only on those records that have yet to be assigned to a class. The ith call to Balance determines a class of size $\Theta(n2^{-2^i})$ and leaves $\Theta(n2^{-2^{i+1}})$ records unassigned. In other words, the ratio of the number of processors to the number of unassigned records is squared with each successive call to Balance. Thus, if we start with n unassigned records and $p \geq n$ processors (instead of n processors), every record is assigned to one of $\max\{1, \lg\lg n - \lg\lg(p/n)\} \leq \lceil \lg\lg p - \lg\lg(p/n)\rceil$ classes using the same number of calls to the subroutine Balance. □

We can now make use of Theorem 7 to obtain a uniform selection algorithm in the same way that Theorem 5 was used to define a nonuniform selection algorithm in Section 6. The resulting selection algorithm corresponds to procedure Select$_4$ of Section 3.

To determine the time complexity of Select$_4$, we can apply the analysis of Section 6 with δ' set to $\lceil c\delta/(\lg\delta - \lg x)^2\rceil$. We obtain the recurrence

$$T(\delta, x) \leq T\left(\delta, \left\lceil \frac{c\delta}{(\lg\delta - \lg x)^2} \right\rceil\right) + c'\delta$$

for $1 \leq x \leq \delta/2$, and some constant $c' > 0$. Applying this recurrence $\lg^*((\lg d)^2) + O(1) = \lg^* d + O(1)$ times, we find that $T(d, 0) = O(d \lg^* d) = O(\lg n \lg^* n)$. We have thus proved the following theorem.

THEOREM 8. *Any call to* Select$_4(S, p, k)$ *with* $n = |S| = p$ *runs in* $O(\lg n \lg^* n)$ *time.*

8. Concluding Remarks.

We have developed a number of asymptotically fast selection algorithms for hypercubic networks. Our analysis of these algorithms has focused on determining their running times to within a constant factor. In order to simplify the analysis, we have occasionally employed rather loose bounds, and so the multiplicative constants implicit in our O-bounds are correspondingly pessimistic.

The selection algorithms described in Sections 2–5 of this paper can be easily expressed in terms of: (i) local operations, (ii) standard normal hypercube primitives (e.g., prefix sum and monotone routing operations), and (iii) calls to previously known normal sorting algorithms (i.e., sparse enumeration sort, Sharesort). Hence, these algorithms are also normal. Furthermore, it is not difficult to argue that both the nonuniform and uniform sparse Sharesort subroutines developed in Sections 6 and 7 are, like Sharesort, normal. Hence all of the selection algorithms discussed in this paper are normal, and we can conclude that our asymptotic time bounds hold not only for the hypercube but also for bounded-degree variants of the hypercube such as the butterfly, cube-connected cycles, and shuffle-exchange.

It is noteworthy that the algorithm devised by Cole and Yap for the powerful and abstract parallel comparison model has essentially pointed the way to the best known algorithms for realistic models of parallel computation.

Acknowledgments. We are grateful to the anonymous referee for insightful remarks that helped to improve the presentation considerably.

References

[1] M. Ajtai, J. Komlós, W. L. Steiger, and E. Szemerédi. Optimal Parallel Selection Has Complexity $O(\log \log N)$. *Journal of Computer and System Sciences*, 38:125–133, 1989.

[2] P. Beame and J. Håstad. Optimal Bounds for Decision Problems on the CRCW PRAM. *Journal of the ACM*, 36:643–670, 1989.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[4] R. Cole. An Optimally Efficient Parallel Selection Algorithm. *Information Processing Letters*, 26:295–299, 1988.

[5] R. Cole and C. K. Yap. A Parallel Median Algorithm. *Information Processing Letters*, 20:137–139, 1985.

[6] 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.

[7] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*: *Arrays*, *Trees*, *and Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[8] D. Nassimi and S. Sahni. Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network. *Journal of the ACM*, 29:642–667, 1982.

[9] C. G. Plaxton. Efficient Computation on Sparse Interconnection Networks. Ph.D. thesis, Department of Computer Science, Stanford University, September 1989.

[10] L. G. Valiant. Parallelism in Comparison Problems. *SIAM Journal on Computing*, 4:348–355, 1975.

[11] U. Vishkin. An Optimal Parallel Algorithm for Selection. In *Parallel and Distributed Computing*, pages 79–86, Volume 4 of Advances in Computing Research. JAI Press, Greenwich, CT, 1987.