On the Network Complexity of Selection

C. Greg Plaxton*
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

The selection problem is to determine the kth largest out of a given set of n keys, and its sequential complexity is well known to be linear. Thus, given a p processor parallel machine, it is natural to ask whether or not an $O(n/p)$ selection algorithm can be devised for that machine. For the EREW PRAM, Vishkin has exhibited a straightforward selection algorithm that achieves optimal speedup for $n = \Omega(p \log p \log\log p)$ [18]. For the network model, the sorting result of Leighton [12] and the token distribution result of Peleg and Upfal [13] together imply that Vishkin's algorithm can be adapted to run in the same asymptotic time bound on a certain class of bounded degree expander networks. On the other hand, none of the network families currently of practical interest have sufficient expansion to permit an efficient implementation of Vishkin's algorithm.

The main result of this paper is an $\Omega((n/p)\log\log p + \log p)$ lower bound for selection on any network that satisfies a particular low expansion property. The class of networks satisfying this property includes all of the common network families such as the tree, multi-dimensional mesh, hypercube, butterfly and shuffle exchange. When n/p is sufficiently large (for example, greater than $\log^2 p$ on the butterfly, hypercube and shuffle exchange), this result is matched by the upper bound presented in [14].

*This work was supported in part by a grant from the AT&T Foundation, NSF grant DCR-8351757 and ONR grant N00014-88-K-0166. The author is primarily supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

This paper is concerned with deriving a lower bound on the complexity of the selection problem for a certain large class of networks. Given a set S of n keys and an integer k, $0 \leq k < n$, the selection problem is to determine a key with rank k in S. If all of the keys are distinct, there will be a unique key with rank k. The lower bound will apply to the special case of the selection problem where all keys are distinct and the key being sought is the median of S, that is, $k = \lfloor n/2 \rfloor$. The only operations allowed on keys are copy and comparison.

The sequential complexity of selection is well known to be linear [5], [16]. The parallel complexity of selection has been studied extensively under a variety of different models of computation; see for example [2], [9], [15] and [18]. For parallel models, the complexity of selection is a function of both p, the number of processors available, as well as n, the size of the set S. As with the related problem of sorting n keys, most research has focused on obtaining tight asymptotic bounds for the case $n = p$. For sorting, the existence of an $O(\log p)$ time comparison-exchange based algorithm for the case $n = p$ implies optimal speedup for $n \geq p$ [4]. For selection, however, a tight bound for the case $n = p$ can be expected to be non-constant. Hence, simulating a fast algorithm for the case $n = p$ will not yield optimal speedup for $n > p$.

On the other hand, optimal speedup for selection is often possible when n/p is sufficiently large. For the EREW PRAM, Vishkin has exhibited a straightforward selection algorithm that achieves optimal speedup for $n = \Omega(p \log p \log\log p)$ [18]. This result has been improved by Cole, who obtained optimal speedup for $n = \Omega(p \log p \log^* p)$ [7]. Vishkin's algorithm is based on two ideas. First, if the set S is partitioned into p groups of size n/p, with a single processor assigned to each group, then the median of each group can be computed in $O(n/p)$ time sequentially, and the median of the resulting set of p medians (which can be obtained by using the fastest known selection algorithm for the case $n = p$) is guaranteed to be a constant fraction splitter for the set S. In other words, it is bigger than a constant fraction of the keys in S, and smaller than a constant fraction of the keys in S (the fraction is $\frac{1}{4}$). Hence, by computing the exact rank in S of this median of medians, a constant fraction of the set S can be discarded from further consideration. The second idea is that the keys which have not been discarded can be partitioned into p equal-sized groups in $O(n/p)$ time. Iterating this process of elimination and redistribution, one finds that the number of keys remaining decreases geometrically and the complexity of Vishkin's algorithm is $O(n/p + \log p \log\log p)$, where Cole's parallel merge sort has been used to make the additive term small [8].

Given that optimal speedup of selection is attainable on the EREW PRAM, for n/p sufficiently large, one is led to ask whether a similar result can be achieved under a more realistic model of computation such as the network model to be defined in Section 2. In fact, networks exist for which optimal speedup of selection is attainable. The sorting result of Leighton [12] and the token distribution result of Peleg and Upfal [13] together imply that Vishkin's algorithm can be implemented to run in $O(n/p + \log p \log\log p)$ time on a certain class of bounded degree expander networks. Given a network corresponding to the graph $G = (V, E)$, the *expansion* of any subset U of the vertices (processors) is defined as

$|\Gamma(U)|/|U|$, where $\Gamma(U)$ denotes the set of vertices in $V \setminus U$ that are adjacent (connected by a communication channel) to at least one vertex in U. The *expansion of a network* is the minimum over all $U \subseteq V$ such that $1 \leq |U| \leq |V|/2$ of the expansion of U. An *expander network* is a network with expansion $\Omega(1)$.

Now consider implementing Vishkin's algorithm on a network such as the hypercube (1-port communication). Sorting $n = p$ keys can be performed in $O(\log^2 p)$ time [3], which only causes an increase in the additive term above, but the redistribution step cannot be implemented efficiently due to insufficient expansion. The expansion of the hypercube is $\Theta(\log^{-1/2} p)$, which leads to a lower bound of $\Omega((n/p)\log^{1/2} p)$ time in the worst case for the redistribution step. Even stronger lower bounds can be proven for other common network families such as the tree, multi-dimensional mesh, butterfly and shuffle-exchange.

The main result of this paper is an $\Omega((n/p)\log\log p + \log p)$ lower bound for selection on any network that satisfies a particular low expansion property defined in Section 5. The class of networks satisfying this property includes all of the common network families such as the tree, multi-dimensional mesh, hypercube, butterfly and shuffle-exchange. The lower bound is proven in Sections 4 and 5. Note that this lower bound disproves a claim of Aggarwal and Huang stating that optimal speedup is possible for selection on the hypercube and shuffle-exchange [1]. When n/p is sufficiently large (for example, greater than $\log^2 p$ on the hypercube and shuffle-exchange), the lower bound is tight to within a multiplicative constant. The matching upper bound is provided by the algorithm Select presented in [14]. Section 3 contains a brief description of this algorithm, along with a few additional remarks.

2 The Network Model

A p processor fixed interconnection network may be viewed as an undirected graph, where vertices correspond to processors and edges correspond to communication channels. Each processor has an infinite local memory. There is no global memory; processors communicate with one another by sending and receiving data over the channels provided by the network. In order to discuss the time complexity of an algorithm it is necessary to define exactly what operations can be performed in a single unit of time, or *time step*. For establishing asymptotic upper bounds, it is realistic to assume that:

1. Memory is configured in $O(\log p)$ bit words.
2. In a single time step, a processor can send and/or receive $O(1)$ words of data and perform $O(1)$ CPU operations on word-sized operands.

For the particular problem of selection it is necessary to make additional assumptions regarding the size of the keys and the operations that are allowed to be performed on them. It will be assumed that a key can be stored in a constant number of words and that the only operations allowed on keys are copy and comparison. It will also be assumed that $n = O(p^c)$ for some positive constant c. Having bounded n in this manner, it may be assumed without loss of generality that all of the keys are distinct [14].

The selection algorithm presented in [14] obeys all of the above restrictions. Section 3 contains a brief outline of that algorithm along with a summary of its asymptotic performance on a variety of networks. The lower bound for selection proven in this paper applies under a strictly more powerful model of network computation than that defined above. In particular, the only restrictions enforced by this model are the following:

1. Each processor can send and/or receive at most one key per time step.
2. The only operations allowed on keys are copy and comparison, and each processor can perform at most one such operation per time step.

Note that an unlimited amount of computation and communication involving data other than keys can be performed in each time step. Under this model, it will be proven that any selection algorithm running on a network satisfying a particular low expansion property requires $\Omega((n/p)\log\log p + \log p)$ time steps in the worst case.

The two models of network computation defined in this section will be referred to as the *upper bound model* and *lower bound model*, respectively.

3 Upper Bounds

As observed in Section 1, non-expander networks such as the hypercube cannot perform the redistribution step of Vishkin's algorithm efficiently. For such networks, the algorithm Select presented in [14] achieves nearly optimal speedup when the ratio n/p is sufficiently large. To be precise, assume that a particular network is capable of sorting $n = p$ keys located one per processor in time T_1 and can perform broadcasting and summing operations in time T_2. Then the running time of algorithm Select may be written as

$$O((n/p)\log\log p + (T_1 + T_2 \log p)\log(n/p)). \qquad (1)$$

Briefly, algorithm Select operates as follows. Initially, there are n/p keys at each processor, all of which could potentially be the key x being sought, that is, the key of rank k in the entire set S of n keys. Like Vishkin's algorithm, Select operates in $\log(n/p)$ stages, and at each stage the number of "live" keys (keys that could be x) is reduced by a constant fraction. The difference is that Select achieves this reduction at every processor without redistributing the keys.

The elimination of keys at each stage is based upon determining a particular key y with rank in S "close" to that of x, as follows. First, each processor computes the median of the set of live values that it holds using an efficient sequential algorithm. The resulting set of p medians is then sorted, and a binary search is performed over the set of sorted medians to determine that median with rank in the set of all live items that is closest to the rank of x. This median is the desired key y and, as proven in [14], it can be used to eliminate half of the live items at every processor. In order to facilitate the $\log p$ rank computations performed by the binary search over the set of p medians, the ith stage begins by spending $O(2^{-i}(n/p)\log\log p)$ time performing local preprocessing of the set of live items at each processor. The corresponding sequential tradeoff between preprocessing time and search time is well understood, see [6] and [11]. The running time of the ith stage is readily shown to be $O(2^{-i}(n/p)\log\log p + T_1 + T_2 \log p)$, which implies the overall time bound for Select stated in Equation (1).

Consider the performance of algorithm Select on a number of common network families. For the butterfly, hypercube and shuffle exchange, $T_1 = O(\log^2 p)$ and $T_2 = \Theta(\log p)$, so the second term in Equation (1) is $\log^2 p \log(n/p)$, and for $n = \Omega(p \log^2 p)$ the running time of Select is $O((n/p)\log\log p)$. For the d-dimensional mesh (d constant), $T_1 = T_2 = \Theta(p^{1/d})$, so the first term dominates for $n = \Omega(p^{1+1/d}\log^2 p/\log\log p)$. Let T_3 denote the time required for a given network to perform selection over $n = p$ keys located one at each processor. If $T_3 \log p < T_1$ then the second term in Equation (1) can be reduced to $(T_2 + T_3)\log p \log(n/p)$ by implementing each of the $\log(n/p)$ binary searches over p medians with $\log p$ selection operations rather than a single sort.

4 A Restricted Lower Bound

This section provides an $\Omega((n/p)\log\log p - \log p/\log\log p)$ time lower bound for selection on a complete network with restricted capability. In order to simplify the exposition, it will be assumed that n and p are both powers of 2, $n \geq p$. Recall that S denotes the set of n keys, and that there are initially n/p keys located at each of the p processors. The lower bound established in this section applies under the model of computation defined below.

Definition 4.1 The *restricted lower bound model* is equivalent to the lower bound model defined in Section 2, with the following additional restriction. For every $R \subseteq S$, if R is initially assigned to a set of processors X, then at most $|X|$ comparisons per time step can involve keys belonging to R.

The motivation for the restricted lower bound model is that networks with poor expansion properties suffer from a similar, albeit less severe, inability to spread out a concentrated set of data in order to apply more processors to it. In particular, if the size of the neighborhood of a given set of processors X is $\alpha|X|$ where $\alpha = o(1/\log\log p)$, then in $O((n/p)\log\log p)$ time not even a constant fraction of the $|X|n/p$ keys initially located in the set X can have been moved or copied to processors outside of X. Of course, the restricted lower bound model is, by itself, quite unrealistic. Furthermore, it is unclear whether or not selection can be performed in $O((n/p)\log\log p)$ time on this model, even assuming the complete network. However, the preceding observation indicates that it may be possible to transfer a lower bound for the complete network operating under the restricted lower bound model to a realistic network operating under the lower bound model.

The remainder of Section 4 is concerned with proving an $\Omega((n/p)\log\log p - \log p/\log\log p)$ lower bound on the running time of any algorithm for computing the median on the complete network operating under the restricted lower bound model. The proof makes use of an adversary argument. At each time step, the algorithm indicates which set of at most p comparisons it would like to make, and the adversary resolves these comparisons sequentially.

Of course, the algorithm must respect the rules of the restricted lower bound model, and the adversary must resolve comparisons in a manner that is consistent with at least one total ordering of the keys. Sometimes the adversary will give away the outcome of a comparison that has not been performed by the algorithm. This is done in order to simplify the lower bound argument. Note that giving away such additional information can only help the algorithm.

It is useful to keep in mind that the restricted lower bound model does not limit the amount of computation or communication involving non-key data that can be performed in each time step. Hence, it may be assumed that at all times, every processor is aware of all of the comparison information that has been gathered thus far. In other words, one may envision a global controller that receives the outcome of every comparison query made in a given time step, and then performs an unbounded amount of computation in order to determine the next set of comparison queries. This is essentially Valiant's parallel comparison model [17], except for the added restriction imposed by Definition 4.1.

The description and analysis of the adversary argument has been divided into a number of parts. Section 4.1 describes the information that the adversary gives away at the outset of the computation. Section 4.2 provides some useful definitions. Section 4.3 states, without proof, the invariants that will be satisfied by the adversary. Section 4.4 gives the procedure by which the adversary resolves comparison queries made by the algorithm. Section 4.5 completes the construction of the adversary by describing the additional information given away at certain points during the computation. Section 4.6 proves that the adversary resolves comparison queries in a consistent manner. Sections 4.4 to 4.6 assume that the invariants of Section 4.3 are satisfied. Section 4.7 proves that the construction of the adversary actually does ensure that these invariants are satisfied. Finally, Section 4.8 gives a precise statement of the lower bound established by the adversary argument.

4.1 The Initial Setup

Before the computation begins, certain information is given to the algorithm for free. Several definitions are needed in order to describe this information. Let a *block* of processors be defined as a set of processors B such that the $|B|n/p$ keys initially located in the set B have contiguous ranks in the set of all keys S. Let λ denote a positive integer to be defined later (it turns out that $\lambda = \Theta(\log p / \log\log p)$). It will be convenient to define the concept of a *block at level* i, $0 \leq i < \lambda$, which is a block with the following additional properties.

1. All blocks at level i contain the same number of processors, s_i, and they are pairwise disjoint. Furthermore, $s_0 = p$.

2. A block at level i contains $b_i = 2^{\lceil \log(i+1) \rceil}$ pairwise disjoint blocks at level $i+1$, $0 \leq i < \lambda - 1$.

3. The size of the blocks at level $i+1$ is such that the union of the blocks at level $i+1$ within a given block B at level i contains $\gamma|B|$ processors, $0 \leq i < \lambda - 1$, where γ is a real constant between 0 and 1 (it turns out that $\frac{1}{8}$ is an appropriate choice for γ, see below and the proof of Lemma 4.4).

Thus, $s_{i+1} = \gamma s_i / b_i$ and hence

$$s_i = \frac{\gamma^i p}{b_0 \cdots b_{i-1}},$$

$0 \le i < \lambda$. Setting γ to the reciprocal of some power of 2 will ensure that the s_i's are integer-valued as long as $p \ge c^i i!$ for some constant c. Taking logarithms, this requirement becomes $\log p \ge i \log i + O(i)$, which is asymptotically satisfied for certain values of λ in the range $\Theta(\log p / \log\log p)$. Note that every block has a unique level that can be determined from its size.

The following information is given away by the adversary before the computation begins. First, the input permutation of the keys is such that the set of all p processors forms a block at level 0. This implies the existence of a tree of blocks of depth λ. The algorithm is given both the ids of the processors that make up each of the blocks in this tree as well as the ordering of the blocks within each level.

4.2 Useful Definitions

The adversary argument proceeds in *stages* consisting of a number of consecutive time steps. The number of stages is given by the positive integer λ defined in the preceding section. The ith stage begins at time $t_i = \max_{0 \le j \le i} \lfloor (n/p) d_j \rfloor - j$ and ends at time t_{i+1}, where $d_i = \frac{1}{2}\log(i+1)$, $0 \le i < \lambda$. Note that $t_0 = 0$ and $t_{i+1} \ge t_i$, $0 \le i < \lambda$. The following pair of technical lemmas will be useful for bounding the amount of work that can be performed in a single stage, and up to a particular stage.

Lemma 4.1 There are at most $\frac{1}{2\ln 2}\frac{n}{p(i+1)}$ time steps in the ith stage.

Proof: Observe that whenever $t_{i+1} - t_i$ is nonzero, it satisfies the following inequality: $t_{i+1} - t_i \le \left(\lfloor \frac{1}{2}(n/p)\log(i+2)\rfloor - i - 1\right) - \left(\lfloor \frac{1}{2}(n/p)\log(i+1)\rfloor - i\right) \le \frac{1}{2\ln 2}(n/p)\ln\left(1 + \frac{1}{i+1}\right) \le \frac{1}{2\ln 2}\frac{n}{p(i+1)}$. □

Lemma 4.2 The starting time of the ith stage, t_i, satisfies the inequality $t_i \le \frac{1}{2}(n/p)\log(i+1)$, $0 \le i < \lambda$.

Proof: Immediate from the definition of t_i. □

Like blocks, processors and keys are assigned unique level numbers. The *level of a processor* is i if and only if the highest level block that it is contained in is at level i. The *level of a key* is given by the level of the processor that it is initially contained in. Note that a processor or key at level i is contained in exactly one block at level j, $0 \le j \le i$.

At any given time during the computation, some subset of the $n!$ possible total orderings of the keys remain consistent with all of the information that the algorithm has learned. Consider an arbitrary pair of distinct keys x and y. If $x < y$ in every one of the possible total orderings, then the outcome of the comparison between x and y is said to be *forced.*

Definition 4.2 Once the outcomes of all $n-1$ comparisons involving a particular key are forced, that key is said to be *dead*. Keys that are not dead are *live*.

Before the beginning of each stage, the adversary will give certain information away (described in Section 4.1 for stage 0, and in Section 4.5 for subsequent stages). After that information has been given away, and before the beginning of the ith stage, $0 \leq i < \lambda$, let D_i and $L_i = S \setminus D_i$ denote the sets of dead and live keys, respectively. Note that the ranks in S of the keys in D_i have all been determined.

Let U_i denote the set of all level i keys in L_i. Let $V_i = L_i \setminus U_i$. It will turn out that all keys of level less than i are dead by the beginning of stage i, so V_i denotes the set of all keys in L_i with level strictly greater than i.

4.3 Invariants

This section states, without proof, certain useful invariants that will be satisfied by the adversary. Section 4.7 proves that the construction of the adversary actually ensures that these invariants are satisfied.

As mentioned earlier, the adversary gives away certain information at the beginning of the ith stage, $0 \leq i < \lambda$. After this information has been given away, and before the ith stage begins, the construction of the adversary will guarantee that the following invariants hold:

1. There are integers $j < \lfloor n/2 \rfloor$ and $k > \lfloor n/2 \rfloor$ such that the set of ranks in S of the keys in D_i is exactly $\{0, \ldots, j-1\} \cup \{k, \ldots, n-1\}$. Thus, the set of ranks in S of the keys in L_i is $\{j, \ldots, k-1\}$.

2. The set L_i is a subset of a single block at level i. Thus, every key in L_i is of level i or higher.

3. The key being sought, namely the median of S, is also the median of L_i.

4. All pairs of keys in U_i are incomparable. Furthermore, $|U_i| \geq (1-\gamma)(n/p)s_i/(i+1)$.

5. The set V_i is exactly the set of keys in a single block at level $i+1$.

6. Every key in U_i is incomparable to every key in V_i.

7. Two or more keys in L_i could still be the median of S.

In particular, note that Invariant 7 implies that the algorithm cannot yet have terminated successfully.

4.4 Resolving Comparison Queries

This section gives the procedure by which the adversary resolves comparison queries made by the algorithm.

Consider a single comparison made by the algorithm, between keys x and y of levels j and k, respectively. Without loss of generality, assume that $j \leq k$. If either x or y is a dead key (this will be referred to as a type A comparison) then the outcome of the comparison is forced, and the adversary responds accordingly. Similarly, if y does not belong to the same block at level j as x (type B comparison), the response is forced. Otherwise, y belongs to the same block at level j as x, and the two cases $i < j$ and $i = j$ will be considered separately.

If $i = j$ (type C comparison) then x belongs to U_i and y belongs to either U_i or V_i. The adversary alternately resolves queries of this type by saying that x is smaller than (larger than), not only y, but the entire set of remaining live keys. The key x becomes a dead key.

In order to determine how to resolve a comparison query in the case $i < j$ (type D comparison), the adversary consults a *comparison tree* that it has been maintaining for the block at level j containing x. The adversary maintains such a comparison tree for every block. A comparison tree of the same sort was used by Borodin et al. [6] to obtain an easy (though not their strongest) sequential tradeoff between preprocessing time and search time in a partial order. A comparison tree is a binary tree with *tokens* placed at certain nodes. The comparison tree for a block B at level j contains $(1-\gamma)(n/p)s_j$ tokens corresponding to the keys of level j in B, and b_j tokens corresponding to the blocks of level $j+1$ in B. When it is important to distinguish these two types of tokens, they will be referred to as *key tokens* and *block tokens*, respectively. At time 0, the key tokens are all placed at the root and the block tokens are placed, one per node, on the b_j nodes at depth $\log b_j = \lceil \log(j+1) \rceil$.

Note that every key corresponds to a key token in the comparison tree of exactly one block. Similarly, every block (except those at the highest level, $\lambda - 1$) corresponds to a block token in the comparison tree of exactly one block, namely that of its parent.

To resolve the comparison query between keys x and y in the case $i < j$ (type D comparison), the adversary locates the key token x' corresponding to x in the comparison tree T associated with the block at level j containing x. The adversary also locates the token y' corresponding to y in the same comparison tree. If $j = k$, this will be a key token; otherwise, it will be the block token corresponding to the unique block of level $j+1$ that contains y. Having located tokens x' and y' in the comparison tree T, the adversary resolves the comparison between keys x and y as follows. Let x' and y' reside in nodes u and v of the tree T, respectively, and let w be the least common ancestor of u and v. If w is not equal to either u or v, then the adversary does not move any tokens.

If w is equal to either u or v, the adversary must move at least one token downwards in the tree. If w is equal to both u and v, then x' is moved to the left child of w, and y' is moved to the right child of w. If w is equal to u but not v, then if v is located in the left subtree of w, x' is moved to the right child of w. The remaining cases are treated similarly.

The preceding algorithm describes how the adversary manipulates the tokens in each comparison tree, but does not indicate how comparison queries are resolved. To resolve queries, the adversary interprets each comparison tree as the partial order given by applying the following rules:

1. The keys corresponding to two tokens in the same comparison tree are incomparable if and only if they lie on a single downward path from the root.

2. If two tokens in the same comparison tree do not lie on a single downward path from the root, then the key corresponding to the token lying to the "left" (that is, the token residing in the left subtree of the least common ancestor) is deemed to be the smaller key.

Note that the preceding rules apply to block tokens as well, where the key corresponding to the block token is actually the entire set of keys in the corresponding block. This is appropriate since the set of keys in any block have contiguous ranks in S, and hence they all compare in the same way to any key outside of the block (that is, if B is a block and keys x, y and z are chosen such that $x, y \in B$ and $z \notin B$, then $x < z$ if and only if $y < z$).

At the beginning of the ith stage, consider the set of comparison trees corresponding to the unique block at level i given by Invariant 2 and all of the blocks that it contains. At any time during the ith stage, the inequalities between live keys to which the adversary has committed itself are exactly those encoded by this set of comparison trees. Note that the initial configuration of the tokens in the comparison trees encodes precisely the information given away by the adversary at the beginning of the computation, as described in Section 4.1. Furthermore, one may check that the token movements (if any) performed in processing a particular comparison query are always sufficient to resolve the query. In Section 4.6, it will be proven that this method of resolving queries can never lead to an inconsistent response by the adversary.

4.5 Additional Information

This section completes the construction of the adversary by describing the additional information given away before the beginning of each stage.

After stage i and before stage $i+1$, let T denote the comparison tree associated with the block V_i given by Invariant 5. The adversary counts the number of key tokens residing on each of the b_i paths of length $\lceil \log(i+1) \rceil$ descending from the root of T. Letting q denote the path containing the most key tokens, the adversary kills off certain keys in such a way that U_{i+1} becomes the set of key tokens on the path q, and V_{i+1} becomes the block of keys at level $i+2$ corresponding to the path q.

Before the beginning of stage $i+1$, the remaining live keys in $X = S \setminus L_{i+1}$ (recall that $L_{i+1} = U_{i+1} \cup V_{i+1}$) are killed off in such a way that the median of S is also the median of L_{i+1}. Every key that is killed off will either be said to be smaller than every key in L_{i+1}, or it will be said to be larger than every key in L_{i+1}. An arbitrary consistent order is maintained between the dead keys. Each key in X of level greater than or equal to $i+1$ corresponds to a key or block token in the tree T that does not reside on the path q. If the token resides to the left (right) of the path q, then the corresponding key is forced to be smaller (larger) than every key in L_{i+1}, in order to ensure consistency. On the other hand, the live level i keys of X remain incomparable to every key in L_{i+1}. Hence, in killing off each of these keys, the adversary has the freedom to decide whether to make it smaller or larger than every key in L_{i+1}. In Section 4.7, it will be proven that X contains sufficiently many live level i keys to force the median of S to be the median of L_{i+1}.

4.6 Consistency of the Adversary

This section proves that the adversary resolves comparison queries in a manner that is consistent with at least one total ordering of the keys.

Section 4.4 gave the adversary's procedure for resolving comparison queries, partitioning the queries into types A, B, C and D. For type A and B comparisons, consistency is immediate.

At any given time during the ith stage, let U_i^L denote the set of keys in U_i that are known to the algorithm to be less than every key in V_i, and let U_i^H denote the set of keys in U_i that are known to be greater than every key in V_i.

Now consider the type C comparisons. As argued in Section 4.4, the live keys in U_i remain incomparable to one another and to V_i throughout the ith stage. Therefore, at all times during the ith stage, any live key in U_i could be the minimum (maximum) key among all of the remaining live keys. Hence, the adversary can consistently kill off any key in U_i and assign it to either U_i^L or U_i^H. A similar argument can be used to prove that the comparison information given away by the adversary at the beginning of each stage (this information was described in Section 4.5) does not lead to an inconsistency.

The consistency of the procedure for resolving type D comparison queries follows immediately from the following lemma.

Lemma 4.3 Moving a token downward in a comparison tree can never result in an inconsistent response by the adversary.

Proof: Assume the lemma is false, and consider the first downward movement of a token that results in an inconsistent response by the adversary. Assume the token was moved downward in comparison tree T corresponding to block B at level i. First note that while a token movement in T can affect the partial order represented by T, it cannot affect the partial order represented by any other comparison tree. This "decoupling" of the partial orders represented by the various comparison trees follows from the use of block tokens to represent all of the keys in block B with level strictly greater than i. Since all of the keys within a block B' at level $i+1$ are treated as a single key in tree T, no comparison that is resolved within T can provide any ordering information regarding keys within block B'. By a similar argument, a token movement in tree T gives no information about keys at levels strictly less than i.

Thus, the inconsistency must arise within the partial order represented by comparison tree T alone. This is impossible since moving a token downward in a comparison tree can only augment the partial order that it represents, and a total order over the tokens in T that is consistent with this partial order is trivial to construct (by an inorder traversal of the tree, for example). □

The preceding discussion establishes that, up to any particular point in the computation, the behavior of the adversary is consistent with at least one total ordering of the keys.

4.7 Correctness of the Adversary

Sections 4.4 to 4.6 assume that the invariants of Section 4.3 are satisfied. This section proves that the construction of the adversary actually does ensure that these invariants are satisfied.

The proof is by induction on the number of stages. It is easy to check that all of the invariants are satisfied at the beginning of stage 0, with L_0 being S, U_0 being the set of $(1-\gamma)n$ level 0 keys, and $V_0 = S \setminus U_0$ being the set of keys in the lone block at level 1 (note that $b_0 = 2^{\lceil \log 1 \rceil} = 1$). The induction hypothesis is that the invariants hold at the beginning of the ith stage, $0 \leq i < \lambda - 1$. It remains to be proven that the invariants hold at the beginning of stage $i+1$. Of these, only Invariants 3 and 4 do not follow immediately from the construction of the adversary. The task of establishing this remaining pair of invariants will now be addressed.

The set of keys U_i are initially contained in a set of at most $(1-\gamma)s_i$ processors (recall that all the keys in U_i are of level i), so Definition 4.1 and Lemma 4.1 imply that only $\frac{1}{2\ln 2}(n/p)(1-\gamma)s_i/(i+1)$ comparisons made during the ith stage can involve keys from U_i. Now each such comparison (even if it involves two keys from U_i) kills off only one key in U_i, and leaves the remaining live keys in U_i incomparable to one another and to V_i. By Invariant 4 (which holds at the beginning of stage i by the induction hypothesis), the preceding comparison bound implies that only a fraction $\frac{1}{2\ln 2}$ of the keys in U_i could have been killed off during the ith stage. Recall that the adversary kills off keys in U_i by alternately assigning them to U_i^L and U_i^H. Hence, at the end of stage i, the algorithm could at best have determined that $\frac{1}{4\ln 2}|U_i|$ of the keys in U_i belong to U_i^L, and that a similar number belong to U_i^H. At least $\left(1 - \frac{1}{2\ln 2}\right)|U_i|$ of the keys in U_i are still incomparable to one another and to V_i. Using the inequality of Invariant 4, and the fact that $|V_i| = (n/p)s_{i+1} = \gamma(n/p)s_i/b_i$, one finds that the ratio $|U_i|/|V_i|$ is at least $(1-\gamma)/\gamma$. Now consider the proof of the following lemma.

Lemma 4.4 For sufficiently small choices of γ, any of the keys in V_i could still be the median of L_i at the end of stage i.

Proof: The median of S is also the median of L_i by Invariant 3 (which holds at the beginning of stage i by the induction hypothesis). Hence, it is sufficient to prove that the size of the set of keys known to reside in U_i^L (U_i^H) plus the size of the set V_i is less than $|L_i|/2$. Using the inequalities $|U_i^L| \leq \frac{1}{4\ln 2}|U_i|$ and $|U_i|/|V_i| \geq (1-\gamma)/\gamma$ proven above, this sum can be bounded by $\frac{1-\gamma}{4\ln 2}|L_i| + \gamma|L_i|$ at the end of the ith stage. Thus, the lemma holds for $\gamma < \frac{2\ln 2 - 1}{4\ln 2 - 1} \approx 0.218$. As mentioned earlier, it is convenient to set γ to the reciprocal of a power of 2; $\frac{1}{8}$ is an appropriate choice. □

Lemma 4.4 proves that the adversary can kill off keys at the beginning of stage $i+1$ in such a way that the median of S is also the median of L_i, as required in Section 4.5. Thus, Invariant 3 holds at the beginning of stage $i+1$.

It remains to prove Invariant 4. The construction of the adversary ensures that the keys of U_{i+1} are incomparable at the beginning of stage $i+1$, but the lower bound on $|U_{i+1}|$ requires proof. Let T denote the comparison tree from which the adversary extracts the set U_{i+1}. The number of key tokens in this tree is equal to $(1-\gamma)(n/p)s_{i+1}$, and these tokens reside initially in a set of $(1-\gamma)s_{i+1}$ processors. Let Λ denote the average depth of the key tokens in T. Observe that every comparison made by the algorithm increments the depth of at most two tokens. Hence, Definition 4.1 and Lemma 4.2 imply that $\Lambda \leq \frac{1}{2}\log(i+2)$ at time t_{i+1}. Let P denote the set of b_{i+1} paths in T from the root to the initial position of each of the b_{i+1} block tokens in T, that is, all paths of depth $\lceil \log(i+2) \rceil$. Let a_j denote the number of key tokens at depth j in T at time t_{i+1}. By a simple averaging argument, some path in P must contain at least

$$\sum_{0 \leq j \leq \lceil \log(i+2) \rceil} a_j 2^{-j}$$

key tokens. This sum is minimized by moving the tokens downward in a uniform fashion. Hence, the bound on Λ implies that some path q in P contains at least a fraction $2^{-\log(i+2)} = \frac{1}{i+2}$ of the key tokens. Therefore $|U_{i+1}| \geq (1-\gamma)(n/p)s_{i+1}/(i+2)$, and Invariant 4 holds.

Thus, the construction of the adversary ensures that Invariants 1 to 7 all hold at the beginning of stage $i+1$, $0 \leq i < \lambda - 1$, and the proof by induction is complete.

4.8 The Lower Bound

The following theorem summarizes the main result of Section 4.

Theorem 4.1 Any selection algorithm for the complete network running under the restricted lower bound model requires $\frac{1-o(1)}{2}(n/p)\log\log p - O(\log p/\log\log p)$ time steps.

Proof: This bound follows from Invariant 7 and the definition of t_i, with $i = \lambda - 1 = \Theta(\log p/\log\log p)$. $\square$

The next section proves that the argument used to establish this lower bound can be adapted to a large class of realistic networks running under the lower bound model. Note that for $n/p = O(\log p/(\log\log p)^2)$, Theorem 4.1 does not provide any useful information; alternative lower bound arguments need to be applied. If every processor is required to receive a copy of the median, then a trivial $\Omega(\log p)$ lower bound holds, even for the complete network running under the lower bound model. If this requirement is not made, the task of proving an $\Omega(\log p)$ lower bound for such a powerful network may not be entirely trivial. For the complete network running under the restricted lower bound model, it is easy to prove an $\Omega(\log p)$ lower bound for computing the maximum (and hence for selection). However, this result is not particularly useful since the proof does not carry over to realistic networks running under the lower bound model. In any event, such considerations may be avoided in the special case of $\Omega(\log p)$ diameter networks, since a simple fooling argument implies that at least $\lceil d/2 \rceil$ time steps are necessary for any selection algorithm running (under the lower bound model) on a network with diameter d. Note that all bounded degree networks, and all of the networks considered in Section 5, have $\Omega(\log p)$ diameter.

5 The Network Lower Bound

The purpose of this section is to prove an $\Omega((n/p)\log\log p + \log p)$ time lower bound for selection on certain realistic networks. The lower bound will apply under the powerful lower bound model defined in Section 2, and will be obtained by making suitable modifications to the proof of Theorem 4.1. Consider the following definition.

Definition 5.1 Let $\mathcal{N}(\alpha, \beta)$ denote the class of all network families $\mathcal{F}$ for which, given any p processor network in $\mathcal{F}$, it is possible to construct all of the blocks (as defined in Section 4.1) at levels less than β in such a way that every block has expansion at most α, where α and β may depend on p.

A careful examination of the proof of Theorem 4.1 reveals that there are only two points at which the definition of the restricted lower bound model is invoked; the rest of the proof applies to the unrestricted lower bound model. The first use of Definition 4.1 is in the proof of Lemma 4.4, and the second use leads to the existence of a suitable set U_{i+1} in the induction step. In both cases, Definition 4.1 provides an upper bound of $(1-\gamma)s_i$ on the number of comparisons per time step involving the set of level i keys of a particular block at level i.

Theorem 5.1 Let $\mathcal{F}$ be a network family belonging to the class $\mathcal{N}(o(1/\log\log p), \beta)$. Then any selection algorithm for $\mathcal{F}$ has a running time of at least $\frac{1}{2}(n/p)\log\beta - \beta$ under the lower bound model.

Proof: In the following argument, let X denote a generic block at level i, $0 \le i < \beta - 1$. Let Y denote the union of the blocks at level $i+1$ in X, and let $Z = X \setminus Y$. Let Y' and Z' denote the sets of keys initially residing in Y and Z, respectively. By the remarks preceding the statement of the theorem, it is sufficient to prove that the adversary construction of Section 4 can be revised in such a way that the two applications of Definition 4.1 can be avoided.

The construction of the adversary will be augmented in the following manner. Let T denote the comparison tree associated with block X. At any given time step, each of the $|Z'|$ key tokens in T is said to be either *bad* or *good*. A good key token is one for which no copy of the corresponding key has ever left the set of processors Z. Thus, all key tokens are initially good, and bad key tokens never become good again. How many good key tokens in tree T can become bad in a single time step? There are only two ways for a good key token to become bad. One way is for a copy of the corresponding key to be transmitted to a processor outside of X. Since block X has expansion $o(1/\log\log p)$, the number of good key tokens that can become bad in this manner is $o(|X|/\log\log p)$ per time step. The other way for a key token to become bad is for a copy of the corresponding key to be transmitted to a processor in Y. Since Y is the union of a number of disjoint sets with expansion $o(1/\log\log p)$, the number of these events is $o(|Y|/\log\log p)$ per time step. By construction, $|Z|$ is a constant fraction of $|X|$, so the total number of key tokens that can become bad in a single time step is $o(|Z|/\log\log p)$. Now the lower bound argument only runs for $O((n/p)\log\log p)$ time steps, during which time the number of bad tokens that can be generated is $o(|Z|n/p) = o(|Z'|)$. In other words, the number of good key tokens in T is $(1-o(1))|Z'|$ at all times in the range of interest.

The final modification to the adversary construction of Section 4 is as follows. In the induction step, the adversary now extracts the set U_{i+1} from the set of good key tokens only. In a single time step, at most $|Z|$ comparisons can be made involving keys in the subset of Z' corresponding to the good key tokens in T, since processors outside of Z do not have copies of any of these keys. Thus, the argument of Section 4 goes through with the inequality of Invariant 4 weakened by a factor of $1-o(1)$, the fraction of all key tokens that are guaranteed to be good. Now consider the proof of Lemma 4.4. At the beginning of the ith stage, all of the key tokens corresponding to U_i are good, and they reside in a set of $|Z|$ processors. Using an expansion argument as above, Lemma 4.1 implies that at most a $o(1)$ fraction of the key tokens corresponding to U_i can become bad during the ith stage. This minor effect is of no consequence since γ has already been set to a value that is bounded away from its maximum acceptable value. A similar comment applies to the effect of the weakened inequality in Invariant 4. □

Corollary 5.1.1 Let $\mathcal{F}$ be an $\Omega(\log p)$ diameter network family belonging to the class $\mathcal{N}(o(1/\log\log p), \Theta(\beta))$. Then any selection algorithm for $\mathcal{F}$ has a running time of at least $\frac{1}{2}(n/p)\log\beta + \Omega(\log p)$ under the lower bound model.

Proof: This bound follows immediately from Theorem 5.1 and the additional observation that any selection algorithm for a network of diameter d has a running time of at least $\lfloor d/2 \rfloor$ under the lower bound model. □

5.1 The Hypercube

Throughout this section, the quantity ϵ will be used to denote an arbitrarily small positive constant. A decomposition of the hypercube will now be defined that proves the hypercube network family belongs to $\mathcal{N}(o(1/\log\log p), \Theta(\log^{1/3-\epsilon} p))$. Given a hypercube with $p = 2^d$ processors, let $q = \lfloor d^\delta \rfloor$ or $\lfloor d^\delta \rfloor - 1$, whichever is odd. The exponent δ is a parameter between 0 and 1 to be determined later. Let $r = \lfloor d/q \rfloor$, and divide the first qr bits of each processor id into r *fields* of q contiguous bits. The ith field determines the ith bit of an r-bit *condensed id* according to the following rule, $0 \le i < r$. If the majority of the q bits in the ith field are 0, then the ith bit of the condensed id is 0; otherwise, it is a 1. Note that since q is odd there will always be a strict majority of either 0's or 1's. By symmetry, 2^{d-l} processors will belong to any *condensed subcube* of dimension $r-l$ obtained by fixing the values of l bits in the condensed id, $0 \le l < r$.

Lemma 5.1 The expansion of a condensed subcube of dimension l is $O(lq^{-1/2})$.

Proof: Let U denote the set of processors belonging to a particular condensed subcube of dimension l. By symmetry, it will suffice to consider the condensed subcube corresponding to the condensed id with first l bits fixed to 0, and with the remaining $r-l$ unspecified. Let V denote the set of processors in $\Gamma(U) \setminus U$ that are adjacent to some processor in V across some dimension in field 0. It is sufficient to prove that $|V|/|U| = O(q^{-1/2})$. But this ratio is readily seen to be precisely $\binom{q}{(q+1)/2}$ over 2^{q-1}, which is $O(q^{-1/2})$ by Stirling's approximation. $\square$

Since $l \leq r$, $q = \Theta(d^{\delta})$ and $r = \Theta(d^{1-\delta})$, Lemma 5.1 implies that the expansion of every condensed subcube is $O(rq^{-1/2}) = O(d^{1-3\delta/2})$. If δ is chosen to be any constant strictly between 2/3 and 1, then the expansion of every condensed subcube will be $O(\log^{-\epsilon} p) = o(1/\log\log p)$. Furthermore, it should be clear that the condensed subcube structure can be used to construct the tree of blocks required by the lower bound argument of Section 4, at least to a certain depth, since the size of every block is a power of 2. All that remains is to determine the maximum possible value of β as a function of p. The relevant inequality is $c^{\beta}\beta! \leq 2^r$ for some constant c, which is satisfied for $\beta = \Theta(d^{1-\delta-\epsilon/2})$. Setting $\delta = 2/3 + \epsilon/2$ gives $\beta = d^{1/3-\epsilon}$, as claimed above. Slightly finer calculations allow this ϵ to be replaced by $o(1)$. Hence, Corollary 5.1.1 implies the following result.

Theorem 5.2 Any selection algorithm for the hypercube has a running time of at least $\frac{1-o(1)}{6}(n/p)\log\log p + \Omega(\log p)$ under the lower bound model.

5.2 Other Networks

The above decomposition also works for the shuffle-exchange, since it is easy to prove that Lemma 5.1 remains valid. Hence, the lower bound of Theorem 5.2 applies to the shuffle-exchange. Similar comments apply for the butterfly network.

Low flux networks such as the tree and multi-dimensional mesh can be easily decomposed into a large number of equal-sized components with very poor expansion. In such cases, β can be increased so that the lower bound of Theorem 5.2 applies with an improved multiplicative constant of $\frac{1-o(1)}{2}$.

6 Concluding Remarks

The upper and lower bounds for network selection discussed in this paper significantly improve on previously known results when the number of keys at each processor, n/p, is sufficiently large [10]. In proving lower bounds, it was assumed that n and p are powers of 2, and that every processor begins with exactly n/p keys. The proofs can easily be extended to handle arbitrary values of n and p (losing at most a constant factor), and arbitrary initial distributions of the keys. Theorem 5.1 was proven for network families $\mathcal{F}$ belonging to $\mathcal{N}(\alpha, \beta)$ with $\alpha = o(1/\log\log p)$; for $\alpha = \Omega(1/\log\log p)$ an obvious tradeoff occurs. It is likely that the multiplicative constants appearing in the lower bounds could be improved. Finally, it should be emphasized that this work deals with the worst case complexity of selection. Under an average case analysis, and for sufficiently high values of the ratio n/p, optimal speedup of selection is possible on all of the networks discussed in this paper.

7 Acknowledgments

Thanks to Leo Guibas, Andrew Goldberg, Edith Cohen, Ernst Mayr and Alex Wang for enduring various early versions of the lower bound argument (names are listed in chronological order).

References

[1] A. Aggarwal and M.-D. A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks. In J. H. Reif, editor, *VLSI Algorithms and Architectures: Proceedings of the 3rd Aegean Workshop on Computing,* Lecture Notes in Computer Science, volume 319, pages 339–350. Springer-Verlag, 1988.

[2] M. Ajtai, J. Komlós, W. L. Steiger, and E. Szemerédi. Deterministic selection in $O(\log \log n)$ parallel time. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 188–195, May 1986.

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

[4] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C–27:84–87, 1978.

[5] M. Blum, R. W. Floyd, V. R. Pratt, and R. L. Rivest. Time bounds for selection. *J. Comput. Sys. Sci.*, 7:448–461, 1973.

[6] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[7] R. Cole. An optimal selection algorithm. Technical Report #209, Ultracomputer Research Laboratory, March 1986.

[8] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, 1988.

[9] R. Cole and C. K. Yap. A parallel median algorithm. *Information Processing Letters*, 20:137–139, 1985.

[10] A. Gottlieb and C. P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *JACM*, 31:193–209, 1984.

[11] R. M. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17:883–902, 1988.

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

[13] D. Peleg and E. Upfal. The token distribution problem. *SIAM J. Comput.*, 18:229–243, 1989.

[14] C. G. Plaxton. Load balancing, selection and sorting on the hypercube. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, June 1989.

[15] M. Rodeh. Finding the median distributively. *JCSS*, 24:162–166, 1982.

[16] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *JCSS*, 13:184–199, 1976.

[17] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[18] U. Vishkin. An optimal parallel algorithm for selection. Technical Report 106, Courant Institute of Mathematical Sciences, Department of Computer Science, December 1983.