Randomized Sorting and Selection
on Mesh-Connected Processor Arrays

Christos Kaklamanis\textsuperscript{1} Danny Krizanc\textsuperscript{2} Lata Narayanan\textsuperscript{3} Thanasis Tsantilas\textsuperscript{4}

Abstract

We show that sorting an input of size $N = n^2$ can be performed by an $n \times n$ mesh-connected processor array in $2.5n + o(n)$ parallel communication steps and using constant size queues, with high probability. The best previously known algorithm for this problem required $3n + o(n)$ steps. We also show that selecting the element of rank $k$ out of $N = n^2$ inputs on an $n \times n$ mesh can be performed in $1.25n + o(n)$ steps and using constant size queues, with high probability. The best previously known algorithm for this problem involved sorting, and required $3n + o(n)$ steps. Both of our algorithms can be generalized to higher dimensions, achieving bounds better than the known results.

\textsuperscript{1}Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.
\textsuperscript{2}Dept. of Computer Science, University of Rochester, Rochester, NY 14627.
\textsuperscript{3}Dept. of Computer Science, University of Rochester, Rochester, NY 14627.
\textsuperscript{4}Dept. of Computer Science, Columbia University, New York, NY 10027.
1 Introduction

The mesh-connected array has been the object of a great deal of theoretical study as well as being the basis for a number of proposed and implemented parallel computers. While its diameter is large in comparison to other well-studied networks (e.g., hypercube, butterfly, shuffle-exchange networks), the simplicity and regularity of its interconnection pattern make it ideal for VLSI implementation. Recent work by Dally [Dal87] suggests that high diameter networks such as the mesh may provide a more efficient communication medium for VLSI-based parallel computers. Furthermore, a large number of efficient algorithms have been designed to run on this architecture.

Sorting and selection (the problem of selecting the element of rank $k$ out of $N$ elements), are important and well-studied problems in computer science. Valiant [Val75] was the first to study the parallel complexity of these and other comparison-based problems and his work has been followed by that of a great many researchers working on different models of parallelism. Reischuk [Rei85] presented a randomized algorithm for sorting $N$ inputs on an $N$ processor parallel random access machine in $O(\log N)$ time, and an algorithm for selection (discovered independently by Meggido [Meg82]) on a $N$ processor parallel comparison tree running in $O(1)$ time, both results holding with high probability. Building upon these results, Reif and Valiant [RV87] gave an optimal randomized $O(\log N)$ time algorithm for sorting on the $N$-node hypercube (and related networks). Recently, Rajasekaran [Raj90] applied these ideas to achieve optimal time selection in the single-port model of the hypercube. Using new techniques, we are able to adapt these ideas to show bounds for sorting and selection on the mesh which are significant improvements over previously known results.

The problem of sorting on a mesh has a long history starting with Thompson and Kung [TK77], who gave an algorithm which sorts $N = n^2$ inputs into snake-like row major order in $6n + o(n)$ parallel communication steps on a $n \times n$ synchronous SIMD mesh; their algorithm may be adapted to run in $3n + o(n)$ time on a MIMD mesh. Schnorr and Shamir [SS86] gave a second algorithm for sorting on a MIMD mesh running in $3n + o(n)$ time, and they also provide a lower bound, discovered independently by Kunde [Kun89], of $3n - o(n)$ communication steps. The model for the lower bound puts no limit on the power of the processors but requires each processor to hold exactly one packet at all times, which forces, for all steps of an algorithm, the configuration of the packets over the mesh to be a permutation of the input.

In this paper, we consider a model in which a processor is allowed to perform only simple operations, to communicate one packet of information to its neighbors during a single time step and to store a constant number of packets between time steps. (This is the same model used when the problem of routing permutations on the mesh is studied; see for example [RT91], [LMT89] and [Kun88].) In this model we provide a randomized algorithm for sorting $N = n^2$ inputs on a $n \times n$ mesh in $2.5n + o(n)$ steps, with high probability. The inputs are sorted into block snake-like ordering (defined below). It should be noted that a lower bound of $3n - o(n)$ steps for any deterministic algorithms for sorting to this indexing scheme is easily obtained in the Schnorr-Shamir model. Furthermore, Chlebus [Chl89], provides a
lower bound of $2.5n - o(n)$ for any randomized algorithm for sorting into snake-like order in the Schnorr-Shamir model which also extends to the block snake-like ordering used here.

A complete description of the problem of selecting the element of rank $k$ out of $N = n^2$ elements on a $n \times n$ mesh must include the identity of the processor where the element is output. Due to the mesh’s relatively large diameter, we consider the case where the selected element must reach the middle processor (i.e., the one labeled $(\lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil)$ in the labeling described below). The sorting result of Schnorr and Shamir immediately implies an $O(n)$ time algorithm for selection in general and a $3n + o(n)$ algorithm for selecting the median (the element of rank $\lceil \frac{N}{2} \rceil$) at the middle processor. The results of Kunde [Kun89] imply a lower bound of $2n - o(n)$ steps for selecting the median at the middle processor in the Schnorr and Shamir model.

We give a randomized algorithm for selection at the middle processor which runs in $1.25n + o(n)$ steps, with high probability. Again the model used for this result is the commonly used model of the mesh described above. The only known lower bound for selection in this model is the distance bound of $n$ steps.

The rest of the paper is organized in the following way. The next section gives some preliminary definitions and facts, including details of the model of the mesh that we use. Sections 3 and 4 contain descriptions of our randomized algorithms for sorting and selection, respectively. We close with some discussion of how our algorithms can be extended to higher dimensional meshes.

2 Preliminaries

The $n \times n$ mesh-connected array of processors (or two-dimensional mesh) contains $N = n^2$ processors arranged in a two-dimensional grid without wrap-around edges. More precisely, it corresponds to the graph, $G = (V, E)$, with $V = \{(x, y) \mid x, y \in \langle n \rangle \}$ and $E = \{(x, y), (x, y + 1) \mid x \in \langle n \rangle, y \in \langle n-1 \rangle \} \cup \{(x, y), (x + 1, y) \mid x \in \langle n-1 \rangle, y \in \langle n \rangle \}$, where $\langle n \rangle = \{1, \ldots, n\}$. The $d$-dimensional mesh is the logical extension of the two-dimensional version to higher dimensions.

The input to both of our problems is a set $X = \{x_1, \ldots, x_N\}$, the elements of which may be linearly ordered. An indexing scheme is a bijection from $\langle N \rangle$ to $\langle n \rangle \times \langle n \rangle$. The sorting problem on the mesh is: Given a set $X$, stored with one element per processor, and an indexing scheme, $I$, move the element of rank $k$ in $X$ to the processor labeled $I(k)$.

The selection problem is: Given a set $X$, stored with one element per processor, an integer $1 \leq k \leq N$, and a specified processor $(i, j)$, move the element of rank $k$ in $X$ to the processor labeled $(i, j)$. In what follows we will consider only the case where the specified processor is labeled $(\lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil)$, referred to as the middle processor below. It is generally straightforward how to modify the algorithm for the case of another designated processor.

The computations above are to be performed using the following model of the mesh: During a single parallel communication step, each processor can send and receive a single packet along each of its incident edges, where a packet consists of at most a single element of $X$ along with $O(\log N)$ bits of header information used for routing and counting purposes.
Between communication steps, processors can store packets in their local queues, which are of bounded size. Furthermore, they can perform a constant number of simple operations (e.g., copying, addition, comparison) on the elements and the header information of packets.

Our algorithms for sorting and selection are randomized and therefore have some probability of failure. In this paper, with high probability means with probability at least $1 - n^{-\beta}$ for some appropriate constant $\beta$. To analyze such probabilities, we make extensive use of the following bounds for the tails of the binomial distribution.

**Fact 1 (Bernstein-Chernoff bounds)** Let $S_{N,p}$ be a random variable having binomial distribution with parameters $N$ and $p$. Then, for any $h$ such that $0 \leq h \leq 1.8Np$,

$$P(S_{N,p} \geq Np + h) \leq \exp \left(-\frac{h^2}{3Np}\right).$$

For any $h > 0$,

$$P(S_{N,p} \leq Np - h) \leq \exp \left(-\frac{h^2}{2Np}\right).$$

Central to our algorithms is the use of a random sample of the keys in order to determine approximately the rank of each key. More specifically, given $N$ keys, consider the problem of choosing $N^\delta$ elements which split the keys into buckets of size between $N^{1-\delta}(1 - N^{-2\delta})$ and $N^{1-\delta}(1 + N^{-2\delta})$. Using ideas from [Rei85, RV87] we describe the following randomized algorithm ($\delta$ is a sufficiently small constant) to select these splitters:

**SELECT-SPLITTERS(N)**

**Phase A** Select a sample of keys by having each key toss a coin with bias $\alpha N^{5\delta - 1} \ln N$, for some constant $\alpha$. Then the average size of the sample will be $\alpha N^{5\delta} \ln N$ and with high probability the size will not differ from its average value by more than $\alpha N^{5\delta/2} \ln N$.

**Phase B** Select every $(\alpha N^{4\delta} \ln N)$-th element to be a splitter.

We insist that the size of the sample be within such a range that phase B is guaranteed to give exactly $N^\delta$ splitters. Otherwise, we consider the algorithm to fail. With high probability this will not happen. Also notice that the margin of error in the size of each bucket is such that the following property holds with high probability for all keys: Each key will be in a bucket whose rank is off by at most one relative to the rank of the corresponding bucket given perfect splitting information.

**Lemma 1** For sufficiently small constants $\delta$, given $N$ keys, the above scheme produces $N^\delta$ splitters which split the keys into buckets of size $N^{1-\delta}(1 \pm N^{-2\delta})$. Then, each key will be in a bucket whose rank is off by at most one relative to the rank of the corresponding bucket given perfect splitting information. The probability that the algorithm fails is smaller than $N^{-\alpha/5}$. 

3
Proof. We use fact 1 to analyze the probabilistic behaviour of the above scheme.

Finally, we present some facts about routing and sorting on one-dimensional and two-dimensional arrays.

Lemma 2 Consider the problem of routing a number of packets in a one-dimensional array with bidirectional edges. Assume that the packets have distinct priorities that are used to resolve contentions for the same edge. Then the time it takes each packet to reach its destination is at most equal to its origin-destination distance plus the number of packets that have priority larger than it.

Proof. Consider a packet $p$ and assume without loss of generality that it moves from left to right. Consider the set of packets with priority greater or equal to the priority of $p$ ($p$ including), and assume that the cardinality of this set is $k$. For simplicity assume that all the packets, except $p$, travel to infinity. At each given time, consider the sequence whose $i$th term tells how many among these packets are at the $i$th node. One can prove by induction that after $k - 1$ steps the initial sequence will be reduced to a sequence of zero and ones. This means that the packets are then free to travel and suffer no more delays (the remaining packets have smaller priorities and can be ignored). In the worst case, $p$ did not move at all so it needs time equal to its origin-destination distance to complete the routing.

Fact 2 There is a deterministic algorithm that, for any constant $k > 0$, sorts $kn^2$ keys in an $n \times n$ array (where initially and finally there are $k$ keys per node) in time $O(n)$. The algorithm works for any indexing scheme and uses constant size queues.

Proof. Use any of the standard algorithms (e.g. [TK77, SS86]), where every comparison/exchange of two elements is achieved by merge and split operations on lists of size $k$.

Lemma 3 For any constant $k > 0$, there is a randomized algorithm for routing any $k$-relation in an $n \times n$ mesh (i.e., at most $k$ packets originate at each node and at most $k$ packets are destined for any node) in time $O(L)$ and using constant size queues, with high probability, where $L$ is the maximum over all origin-destination distances.

Proof. Such a randomized algorithm that preserves locality and uses $O(1)$ sized queues on the average was presented in [RT91]. Using the redistribution technique from that same paper, we can derive an algorithm that uses $O(1)$ sized queues in the worst case.

Lemma 4 There is a deterministic algorithm that, for any constant $k > 0$, computes the parallel prefix of $kn^2$ elements in an $n \times n$ mesh (where initially there are $k$ keys per node) in time $O(n)$ and uses constant size queues.

Proof. The proof is straightforward and omitted.
3 Sorting

In this section, we describe an algorithm for sorting \( N = n^2 \) elements on a \( n \times n \) mesh in \( 2.5n + o(n) \) time and using constant size queues, with high probability.

Let \( \delta < 1/6 \) be a small constant. Divide the mesh in blocks of processors \( B_i, i = 0, 1, \ldots, N^\delta - 1 \), each of size \( b = N^{1-\delta} \), i.e. the \( B_i \) are \( \sqrt{b} \times \sqrt{b} \) submeshes. We will use an indexing scheme that ranks the blocks so that consecutively ordered blocks are physically adjacent on the mesh, e.g. snake-like; the processors inside each \( B_i \) are indexed with indices in the interval \([(i-1)N^\delta, iN^\delta]\) in some arbitrary way.

Consider also the natural division of the mesh into four \( n/2 \times n/2 \) quadrants \( Q_i \). Define the "middle diamond of radius \( k \)" to include all processors that lie within distance \( k \) from the middle processor, i.e. processor \( ([n/2], [n/2]) \); define \( D \) to be the middle diamond of radius \( .5n \); \( D \) consists of four triangles \( T_i \), where \( T_i = D \cap Q_i \). Define the "middle block" \( B \) to be the block that contains the middle processor. Note that these definitions do not depend on the indexing scheme.

Before we present the algorithm in detail we give a high level description of it. The idea is to route a copy of each element to each triangle \( T_i \); in the meantime, we select a random sample (the splitters), and then broadcast it to each quadrant, in order to determine approximately the rank of each element. The elements are then routed to the neighborhood of the processor whose index equals the approximate rank of the element as long as this processor is in the same quadrant as the triangle they are in. If not, the elements do not survive, so for each element only one of the four copies will survive. In the meantime, in each quadrant we compute and broadcast the exact global rank of the splitters. Now we can determine the exact global rank of each of the elements, which are finally routed to the correct processor.

3.1 The Algorithm

Our algorithm \textsc{Sort}(N) consists of the following 10 steps.

1. Select a random sample of size \( S = o(N^{65} \ln N) \). To do this each processor selects itself with probability \( S/N \). Each element that is at a selected processor picks a random destination in the middle block \( B \), and routes itself greedily towards it.

2. Sort the sample elements in the middle block \( B \) using a standard algorithm for sorting into snake-like order. Pick exactly \( S = N^6 \) splitters as in lemma 1. Note that this leaves \( S \) sorted splitters in the \( b \)-sized \( B \).

3. Broadcast the \( S \) splitters in the middle diamond \( D \). This is done by greedily replicating \( B \) in all \( B_i \), that overlap with \( D \).

4. In each quadrant \( Q_i \), randomize the elements row-wise.
5. Send a copy of each element to each triangle \( T \). This is done by moving "together" the elements that start in each quadrant \( Q_i \), and so that the elements that end at processor \((r, c)\) are the ones that started at processors \((r, c)\) or \((c, r)\) (modn/2). Now each triangle of the middle diamond contains all \( N \) elements; on the average there are 8 elements per processor.

6. In each block \( B_i \) that overlaps \( D \) sort all the elements that were sent there in step 4 along with the splitters sent in step 3. Then in each \( B_i \) do a prefix computation so that the elements will know their presumed splitter bucket (and therefore their destination block), and so that the splitters will know their exact rank in \( B_i \). Then kill all element copies (not the splitters) that do not lie in the same quadrant as their presumed destination block.

7. Route all (live) elements to random nodes in their presumed destination blocks. This involves four essentially disjoint routing problems, one for each \( Q_i \).

8. In each \( Q_i \) compute and broadcast the exact global rank of the splitters.

9. In each \( B_i \) sort the elements; do a prefix computation so that the elements will know their exact rank in \( B_i \); then use the exact global splitter rank (broadcast in step 8) to find for each element its exact global rank.

10. Route each element to its exact final position.

The correctness of the algorithm SORT(N) follows from the discussion above. We will next prove it runs in time \( 2.5n + o(n) \) using queues of size \( O(\log n) \), with high probability. This is followed by a discussion of how the queue size can be reduced to a constant.

### 3.2 Running time of SORT(N)

**Lemma 5** Step 1, routing the sample to \( B \), can be done in time \( n + o(n) \) with constant size queues, with high probability.

**Proof.** Because of lemma 1, there are \( \Theta(N^{55} \log N) \) elements in the sample, with high probability. There are \( N^{1-\delta} \) possible destinations in \( B \), and each sample packet selects one of them uniformly at random. Since \( \delta < 1/6 \) and using fact 1, we can prove that there exists a constant \( c > 1/\delta \) such that with high probability, at most \( c \) packets choose the same destination. We will route the sample using a greedy algorithm, where each packet travels to the correct row and then to the correct column. Note that we only need to analyze the sample packets in one quadrant. Clearly no collisions occur while the packets are traveling up the column to the correct row. The queue size can increase only when two packets enter the queue for an edge at the same time, and this means at least one of the packets must have "turned" at the node. Since each packet turns only once, the expected number of packets that turn at any node, over all time is less than one. Consider any column and the half of
it that lies in the quadrant. With high probability, the number of sample packets in the column does not exceed $n^{55}$. Each of these packets picks one of the "middle" $n^{1-5/2}$ rows to turn. By fact 1, there exists an $a > 1$ such that with high probability, the number of packets turning into a given node is not more than $a$. \hfill \Box

**Lemma 6** Step 2, sorting the sample and picking the splitters, can be done in $o(n)$ time, using constant size queues.

**Proof.** Follows from fact 2. \hfill \Box

**Lemma 7** Step 3, broadcasting the splitter information inside $D$ can be done in $.5n + o(n)$ time, with constant size queues.

**Proof.** This is done by having each splitter send from its initial position inside $B$ four copies, one in each direction; as they travel along the rows (or columns), these copies leave new copies of themselves every $\sqrt{b}$ columns (or rows); then each new copy similarly propagates itself to the two directions in the column (or row). At the end there is a copy of $B$ at each $B_i$ in $D$. Note that no more than $o(n)$ splitter elements travel per row or column; the maximum distance any splitter has to travel is $.5n$. \hfill \Box

**Lemma 8** Step 4, row-wise randomization inside each quadrant, takes $n + o(n)$ time, and needs queues of size $O(\log n)$.

**Proof.** This is done the obvious way. Each element picks a random destination in its half-row and goes there. Because of fact 1, no more that $O(\log n)$ elements are destined for each processor. \hfill \Box

**Lemma 9** Step 5, routing copies of the elements to each triangle $T_i$, can be done in $n + o(n)$ time.

**Proof.** Omitted. \hfill \Box

**Lemma 10** Steps 1-2-3 and 4-5 can be done simultaneously in $1.5n + o(n)$ time using queues of size $O(\log n)$.

**Proof.** Assign to sample and splitter movement highest priority. Then steps 1-2-3 are clearly done after $1.5n + o(n)$ time. On the other hand steps 4-5 will be done in time $1.5n + o(n) + d$ where $d$ is the maximum delay that any element will experience due to sample and splitter movement; but steps 1 and 3 involve the movement of $o(n)$ packets per row or column, while step 2 takes $o(n)$ time; therefore $d = o(n)$. \hfill \Box

**Lemma 11** Step 6 can be done in $o(n)$ time, using constant size queues.

**Proof.** Step 6 involves sorting and doing prefix computations of $O(b)$ elements over a $b$-sized mesh, where $b = o(n)$. Because of fact 2 and fact 4, this can be done $o(n)$ time using constant size queues. \hfill \Box
Lemma 12  Step 7, routing each element to its presumed destination block, can be done in \( n + o(n) \) time using \( O(\log n) \) size queues.

Proof. This is done separately in each quadrant \( Q_i \). In each \( Q_i \), there will be exactly \( N/4 \) live elements \( x_j \) to be routed (for even number of blocks per row). All \( x_j \) start inside \( T_i \) and we start with a constant number of elements per processor. The \( x_j \) are partitioned into two sets \( C_i \) and \( R_i \); \( R_i \) contains the \( x_j \) that after step 5 were in the same row \( (\mod \, n/2) \) as before step 5; \( C_i \) contains the \( x_j \) that after step 5 were in the same column \( (\mod \, n/2) \) as the row they were before step 5; because of the randomization done in step 4, the elements in \( R_i \) are in random columns, while the elements in \( C_i \) are in random rows. During step 7, each \( x_j \) is sent to a random processor somewhere inside its presumed destination block. The routing is done in two phases. In the first phase, elements in \( R_i \) are routed greedily along their columns to the row of their destination, while elements in \( C_i \) are routed along their rows to the column of their destination; then in the second phase all elements are routed to their destinations, \( C_i \) along columns and \( R_i \) along rows.

Now, as shown in claim 1 below, the first phase can finish in time \( .5n + o(n) \) with \( O(\log n) \) size queues, if only the elements in \( R_i \) are considered. Almost the same argument applies for the elements in \( C_i \). Then similar arguments apply to phase 2. Therefore, since in each phase the elements in \( R_i \) use edges different from the ones the elements in \( C_i \) use, step 7 can be done in time \( n + o(n) \) using \( O(\log n) \) size queues.

Claim 1  The elements in \( R_i \) can be routed to the rows of their destinations in time \( .5n + o(n) \) using \( O(\log n) \) size queues.

Proof. The size of \( R_i \) is at most \( N/4 \). Each element in \( R_i \) has picked its column at random during step 4. Therefore (fact 1) no more than \( .5n + o(n) \) elements will be routed along each column. Furthermore (fact 1 again), along each column no more than \( n^{1-\delta} + o(n^{1-\delta}) \) elements are destined for the column segment that overlaps some specific block \( B_i \); note that such segments are of length \( n^{1-\delta} \). This means that if a furthest-to-go priority is used, for each element that wants to travel \( .5n - h(n) \) along the column, no more than \( o(h(n)) + o(n^{1-\delta}) \) will have higher priority. Therefore, because of fact 2, the time required is at most \( .5n + o(n) \). Since at the beginning of the phase we start with constant number of elements per processor, it is clear that this routing needs no more than constant queues, except for the accumulation of packets at their destinations along the column. Since each element picks at random the particular processor that it wants to go inside its presumed destination block, we can show that no more than \( O(\log n) \) elements are destined for each processor during this phase. So clearly this phase can be done with \( O(\log n) \) size queues. □

Lemma 13  Step 8, computing and broadcasting in each quadrant the exact global rank of the splitters, can be done in \( n + o(n) \) time using constant size queues.

Proof. In step 6 the relative ranks of the splitters were computed inside each block \( B_i \). Since at that time there was exactly one copy of each element per quadrant, adding in each
quadrant the splitter partial rank information obtained in step 6 is enough to compute the global rank of the splitters. To do this, we first compact inside each $B_i$ the splitter partial rank information; this is compacted in canonical order in a $\sqrt{s} \times \sqrt{s}$ submesh at the left top corner of $B_i$. Then this information propagates towards the center of the quadrant in a process reversely similar as in lemma 5; during the propagation the partial ranks are added at each $B_i$. Thus, after $.5n + o(n)$ time, the global ranks of the splitters are computed at the center of each quadrant. Then this information is broadcast back to each block in the quadrant, again as in lemma 5 in time $.5n + o(n)$.

Lemma 14 Steps 7 and 8 can be done simultaneously in $n + o(n)$ time using $O(\log n)$ size queues.

Proof. Assign highest priority to splitter movement; then the argument is similar to lemma 10 above.

Lemma 15 Step 9, sorting in each $B_i$ and doing prefix computations to find the exact rank of each element, can be done in $o(n)$ time with constant size queues.

Proof. As in lemma 11 above, if step 7 finishes with constant size queues.

Lemma 16 Step 10, routing the elements to their final destinations can be done in $o(n)$ time using constant size queues.

Proof. No element has to travel more than $O(N^{\frac{1}{2}})$ distance because of fact 1. Therefore we can perform step 10 using the algorithm of fact 3.

By combining the lemmas above we get the following theorem:

Theorem 1 There exists a randomized algorithm that sorts $N = n^2$ elements on a $n \times n$ mesh in $2.5n + o(n)$ time using $O(\log n)$ size queues, with high probability.

3.3 Reducing the size of the queues to constant

In this section, we briefly describe how the size of the queues can be reduced to constant with high probability. It follows from the lemmas above that we only need to deal with this issue in steps 4, 5 and 7. For these steps the size of the queues follows a binomial distribution with a constant mean. However the bounds of fact 1 are not strong enough to give constant size queues with high probability.

Now, instead of considering individual processors we consider sets of processors obtained by dividing the rows and columns into consecutive groups of $\log n$ nodes each. Then the expected number of packets per group at the end of step 4, for example, is $\Theta(\log n)$ and using fact 1 we see that with high probability the number of packets per group is also $\Theta(\log n)$. Hence, the packets in each group can be redistributed so that only a constant number of packets resides in each queue. In other words, a packet is not stored at the targeted node
but somewhere in the group in which that node belongs. We refer to [RT91] for one way of implementing this redistribution technique. We use the same technique at the end of both phases of step 7.

Then step 5 can be done exactly as described above; no modification is necessary since it starts and ends with constant queues. Finally, at the beginning of both phases of step 7 the packets start $O(\log n)$ distance from their "real" origins. It can be shown that the extra delay introduced by this discrepancy is $o(n)$. The proof is omitted.

**Theorem 2** There exists a randomized algorithm that sorts $N = n^2$ elements on a $n \times n$ mesh in $2.5n + o(n)$ time using constant size queues, with high probability.

## 4 Selection

In this section, we describe our algorithm for selecting on the mesh the element of rank $k$ among $N = n^2$ elements in $1.25n + o(n)$ steps and using constant size queues, with high probability. Reischuk [Rei85] and Megiddo [Meg82] describe parallel algorithms for selection in the parallel comparison tree model which work in expected constant time. Using some of these ideas, we will construct a parallel algorithm on the mesh and prove that it has the properties stated above.

As above, a random sample of the keys is chosen in order to determine an approximation to key of rank $k$. Specifically, given $N$ keys, we want to select two splitters $u$ and $v$ which split the set into three buckets, such that the element of rank $k$ falls in the middle bucket, and further, the size of the middle bucket is $O(N^{1-\delta})$, for some sufficiently small constant $\delta$.

We use the **SELECT-SPLITTERS(N)** algorithm from Section 2 to choose these bracketing elements. Sort the $N^{6}$ splitters obtained from Phase B of the algorithm. Let $1 \leq j \leq N^{6}$ be such that $(j - 1)N^{1-\delta} < k \leq jN^{1-\delta}$, that is, given perfect splitting information, the element of rank $k$ falls into the $j$-th bucket. From lemma 1, with high probability, the element of rank $k$ falls into one of the three buckets determined by the splitters of rank $j-2, j-1, j, j+1$ in the splitter set. We select $u$ and $v$ to be the splitters of rank $j-2$ and $j+1$, respectively. (If $j = 1$ then $u = -\infty$. If $j = N^{6}$ then $v = +\infty$.) Let $M = \{x_i \mid u \leq x_i \leq v\}$. Then it follows from lemma 1 that with high probability, for $\delta < 1/6$, the element of rank $k$ lies between $u$ and $v$, and $|M| \leq 6N^{1-\delta}$.

As in Section 3, we define $B$ to be the middle block of side $n^{1-\delta/2}$ and $D$ to be the middle diamond of radius $n/8$. We will now give a high level description of our algorithm for selection. First we move all the elements into the middle diamond $D$. In the meantime, we select a random sample and route it to the middle block $B$; there we select the two splitters $u$ and $v$ that bracket the element of rank $k$ with high probability, and we broadcast them inside $D$. Then all the elements in the middle bucket $M$ are routed to the middle of the mesh. In the meantime, the exact global ranks of $u$ and $v$ are computed and propagated to the middle block $B$. Finally the element of appropriate rank is selected among the elements in the block $B$. 
4.1 The Algorithm

Our algorithm SELECT(N, k) to select the element of rank k out of \( N = n^2 \) elements, consists of the following seven steps.

1. Select a random sample of size \( S = \alpha N^{3/5} \ln N \). To do this each processor selects itself with probability \( S/N \). Each element that is at a selected processor picks a random destination in the middle block \( B \), and routes itself greedily towards it.

2. Move all the remaining packets into the middle diamond \( D \) of radius \( n/8 \).

3. Sort the sample packets in \( B \) using a standard algorithm for sorting into snake-like order. Find the splitters \( u \) and \( v \) defined above.

4. Broadcast the values of the splitters \( u \) and \( v \) to all the processors in the middle diamond \( D \). Each processor can calculate for each of the packets it holds, which splitter bucket it belongs to.

5. Compute the exact global ranks of the two splitters \( u \) and \( v \). This can be done by finding the number of packets in each of the three buckets created by the splitters.

6. Packets with values in the middle bucket \( M \) choose a random destination in \( B \). Route these packets to their chosen destinations.

7. If the element of rank \( k \) in \( X \) does indeed lie in \( M \), and if \( |M| \leq 6N^{1-\delta} \), sort \( M \) and find the element of rank \( k - \text{rank}(u) \) (which is the element of rank \( k \) in the set \( X \)), and we are done. Otherwise, broadcast a message to the whole mesh, and restart. Sort the input configuration, so that the element of rank \( k \) reaches the middle processor.

The correctness of the algorithm SELECT(N, k) follows from the discussion above. We will now prove it runs in the claimed number of steps using queues of size \( O(\log n) \). This is followed by a discussion of how the queue size can be reduced to a constant.

4.2 Running time of SELECT(N, k)

Lemma 17 Step 1, routing the sample, can be done in \( n + o(n) \) steps with constant size queues, with high probability.

Proof. Similar to lemma 5. \( \square \)

Lemma 18 Step 2, overlapping the packets into \( D \), the middle diamond of side \( n/8 \), can be done in \( 9n/8 \) steps, with constant size queues.
Proof. Consider the portion of the middle diamond of side \( n/9 \) in a particular quadrant. There are \( n/9 \) rows and columns entering it. By equally dividing the \( n^2/4 \) packets of the quadrant among these rows and columns we can maintain a steady stream of \( 2n/9 \) packets entering this diamond and overlap all the packets into this diamond in \( 9n/8 \) steps. However, the corner processors present a problem, as they have either a row edge or a column edge going into the diamond, but are receiving packets from both row and column edges and thus would require nonconstant size buffers. To solve this problem, we buffer the packets in the \( n/72 \) nodes inside the diamond of side \( n/8 \) but outside the diamond of side \( n/9 \). The overlapping can be achieved in such a way that no node buffers more than a constant number of packets.

Lemma 19

(i) Step 3, sorting the sample and selecting the splitters \( u \) and \( v \), takes \( o(n) \) steps.

(ii) Step 4, broadcasting the values of \( u \) and \( v \) into the middle diamond \( D \), takes \( n/8 + 1 \) steps.

(iii) Step 5, computing the exact global ranks of \( u \) and \( v \), when all the packets are overlapped into the middle diamond \( D \), takes \( n/8 + 1 \) steps.

(iv) Step 7, sorting the middle bucket \( M \), and selecting the element of rank \( k - \text{rank}(u) \), takes \( o(n) \) steps.

Proof. The proof of (ii) follows from the fact that all the packets are in the middle diamond. We need to send two packets (the values of \( u \) and \( v \)) to each processor in the inner diamond. These packets travel up and down along all the center column and out along row edges. The two packets can be routed in a pipelined fashion, thus taking \( n/8 + 1 \) steps. The proof of (iii) is by noting that calculating the size of the three buckets is sufficient to calculate the ranks of \( u \) and \( v \). To do this, each packet participates in two summing operations depending on which bucket it lies in. The summing packets proceed in a manner which is exactly the inverse of the operation described in in the proof of (ii). The proofs of (i) and (iv) follow from fact 2.

Lemma 20 Let all the packets be overlapped into the middle diamond \( D \), such that there are no more than a constant number of packets at any node. Then, step 6, routing the elements in \( M \), can be done in time \( n/8 + o(n) \) using queues of size \( O(\log N) \), with high probability.

Proof. We adapt the algorithm described in [RT91] for permutation routing.

Algorithm ROUTE

Divide the mesh up into \( 1/\varepsilon \) slices containing \( \varepsilon n \) rows each. If packet \((i,j)\) wants to go to \((r,s)\) eventually:

**Phase 1:** Choose a random row in its own slice, say \( p \) and go to \((p,j)\).
Phase 2: Go to \((p,s)\) (correct the column).

Phase 3: Go to \((r,s)\) (correct the row).

Let \(\gamma\) be a bound on the number of packets at any node in the middle diamond. We show that ROUTE solves the problem of routing \(O(N^{1-\delta})\) packets with at most \(\gamma\) packets per origin node and at most \(O(\log n)\) packets per destination node in \(n/8 + \gamma e n + o(n)\) steps with queues of size \(O(\log n)\) with high probability.

**Queue size analysis:** In Phase 1, processor \((p,j)\) can receive \(\gamma\) packets from \(\epsilon n\) processors (the ones at the same strip and column as \((p,j)\)), each with probability \(1/(\epsilon n)\). Let \(E_m\) be the event that more than \(m\) packets will be stored at \((p,j)\) at the end of Phase 1. Then, using fact 2, \(P(E_m) = B\left(m; \gamma e n, \frac{1}{\epsilon n}\right) \leq \exp(m \ln \gamma - m \ln m + m - \gamma) = \exp(c m - m \ln m - \gamma).\) Therefore, the probability that any one processor will have more than \(m\) packets at the end of Phase 1, is less than \(n^2 \exp(c m - m \ln m - \gamma) = \exp(2 \ln n - m \ln m + c m - \gamma).\) By choosing \(m = \Theta(\log n)\) we can make this probability smaller than the inverse of some polynomial in \(n\).

Consider a given node \((p,s)\) at the end of Phase 2. With high probability, not more than \(n^{1-\delta/2}\) packets will choose destinations in column \(s\). Each of these picks node \((p,s)\) with probability less than \(1/\epsilon n\). Using fact 2, we can prove that with high probability, at most \(O(\log n)\) packets accumulate at any processor. Since the maximum of the queue sizes at the end of each phase is an upper bound on the sizes of queues at any step during the algorithm, we have proved that with high probability, the queue size grows to at most \(O(\log n)\).

**Routing Time Analysis:** Phase 1 can be accomplished in \(\gamma e n\) steps, simply by making \(\gamma\) passes to account for \(\gamma\) packets at every node. In Phase 2, we use queuing discipline \(Q\), and using a similar analysis to [RT91], we can show that the probability that the delay is more than \(n^\alpha\) is at most \(\exp(-Cn^{2\alpha-1})\). For Phase 3, we use queuing discipline \(Q'\). Notice that the actual distance traveled by any packet is always less than \(n/8 + \epsilon n + n^{1-\delta/2}\). Consider the effect on the running time if the three phases are coalesced. The only possible conflict is between packets doing their Phase 3 and Phase 1. In such a case, the packet doing its Phase 1 is given preference. If a packet \(q\) doing its Phase 3 contends for an edge with a packet that is doing its Phase 1, then it needs to go a maximum of \(n/8\) steps to get to its destination. Since after \(\gamma e n\) steps, \(q\) will only have to contend with packets doing their Phase 3, it will reach its destination in \(n/8 + \gamma e n\) steps. No other packets suffer additional delays due to coalescing the phases.

Taking \(\epsilon = 1/\log n\), we have shown that there is a randomized algorithm to route the middle bucket \(M\) in \(n/8 + o(n)\) steps and using queues of size \(\Theta(\log n)\) with high probability.

We now show that some of the steps in our algorithm for selection can be combined, thus significantly lowering the total time taken by SELECT\((N,k)\).

**Lemma 21** Steps 1, 2, 3, and 4 can be done simultaneously in \(n + n/8 + o(n)\) steps, using constant size queues.
Proof. Give the sample packets priority over overlapping packets. We know from lemma 17 that the sample packets can be routed in $n$ steps. With high probability, the number of sample packets turning into any row is $o(n)$, and therefore the overlapping packets are never delayed more than $o(n)$. The broadcasting packets are traveling in opposite directions to the overlapping packets; there are never any collisions between these. \hfill \square

Lemma 22 Step 5, collecting information about the sizes of the buckets as well as step 6, routing of $M$, can be done simultaneously in $n/8 + o(n)$ steps, using logarithmic size queues.

Proof. The packets belonging to $M$ take lower precedence. Then each such packet can be delayed at most 2 extra steps due to the regular pattern of the summing of the information. \hfill \square

By combining steps 1, 2 and 3 and 4, and steps 5 and 6 of SELECT(N,k), lemmas 19, 21, 22 prove the following theorem:

Theorem 3 There exists an algorithm for finding the element of rank $k$ out of $N = n^2$ elements on an $n \times n$ mesh that finishes in $1.25n + o(n)$ steps and uses queues of size $\Theta(\log n)$, with high probability.

Notice that the only point where we may use large queues in our algorithm is in the routing of $M$. As in the sorting algorithm (see Section 3.3), we are able to apply redistribution techniques to achieve constant size queues here. Therefore, we have:

Theorem 4 There exists an algorithm for finding the element of rank $k$ out of $N = n^2$ elements on an $n \times n$ mesh that finishes in $1.25n + o(n)$ steps and uses queues of size $O(1)$, with high probability.

5 Extensions

Using similar techniques we can obtain the following results for 3-dimensional meshes:

Theorem 5 There exists a randomized algorithm for selecting the element of rank $k$ out of $n^3$ elements on an $n \times n \times n$ mesh that finishes in $2n + o(n)$ time and uses constant size queues, with high probability.

Theorem 6 There exists a randomized algorithm that sorts $n^3$ elements on an $n \times n \times n$ mesh that finishes in $4n + o(n)$ time and uses constant size queues, with high probability.

Note that the previously known best bound for sorting in the 3-d mesh was $5n + o(n)$ [Kun88] which implied the same bound for selection.

Our techniques can be applied to obtain fast selection algorithms in higher dimensional meshes as well. On the other hand, they do not seem to extend to sorting algorithms for meshes with dimension 5 or higher.
References


