Desnakification of Mesh Sorting Algorithms

Jop F. Sibeyn*

Abstract

In all recent near-optimal sorting algorithms for meshes, the packets are sorted with respect to some snake-like indexing. In this paper we present deterministic algorithms for sorting with respect to the more natural row-major indexing.

For 1-1 sorting on an $n \times n$ mesh, we give an algorithm that runs in $2 \cdot n + o(n)$ steps, with maximal queue size five. It is considerably simpler than earlier algorithms. Another algorithm performs k-k sorting in $k \cdot n/2 + o(k \cdot n)$ steps.

Furthermore, we present *uni-axial* algorithms for row-major sorting. Uni-axial algorithms have clear practical and theoretical advantages over bi-axial algorithms. We show that 1-1 sorting can be performed in $2^1\!/\!_2 \cdot n + o(n)$ steps. Alternatively, this problem is solved in $4^1\!/\!_2 \cdot n$ steps for *all* n. For the practically important values of n, this algorithm is much faster than any algorithm with good *asymptotical* performance.

Keywords: theory of parallel computation, meshes, sorting, row-major indexing, uni-axial routing

1 Introduction

Various models for parallel machines have been considered. One of the best studied machines with a fixed interconnection network, is the **mesh**. In this model the processing units, **PUs**, form an array of size $n \times n$ and are connected by a two-dimensional grid of communication links.

Problems. The problems concerning the exchange of packets among the PUs are called **communication problems**. The packets must be sent to their destinations such that at most one packet passes through any wire during a single step. The quality of a communication algorithm is determined by (1) its **run time**, the maximum number of steps T a packet may need to reach its destination, and (2) its **queue size** Q, the maximum number of packets any PU may have to store.

Routing is the basic communication problem. In this problem the packets have a known destination. If all PUs initially hold one packet, and if every PU is the destination of precisely one packet, than we speak of **permutation routing**. The routing problem in which every PU is source and destination of k packets is called the k-k **routing** problem.

Sorting is, next to routing, one of the most considered communication problems. Several variants of the problem have been studied. In the 1-1 sorting problem, each PU initially holds a single packet, where each packet contains a key drawn from a totally ordered set. The packets have to be rearranged such that the packet with the key of rank i is moved to the PU with index i, for all i. In the k-k sorting problem, each PU is the source and destination of k packets.

Scattering is the problem of rearranging packets holding keys from a totally ordered set such that as little as possible packets with the same key stand in the same column. It can be performed by sorting the packets in row-major order on the keys. But, almost the same effect can be achieved at lesser expense. Scattering is not a problem with great independent importance. However, it is an important subroutine of deterministic algorithms for other communication problems: the queue size of such algorithms often linearly depends on the number of packets in any column with destinations in the same row. In this application the scattering is performed in $s \times s$ submeshes, and the key of a packet is given by its destination row.

Models. Frequently it is assumed that the PUs can communicate with all their neighbors at the same time: in a single step they can send and receive at most four packets. This model is called the **MIMD mesh**. Alternatively all PUs may send only packets in a specific direction during any step, the **SIMD mesh** model. Less considered is the model in between these two, which we call the **half-MIMD mesh**. In a half-MIMD all PUs can either send and receive packets along the horizontal or along the vertical connections. Algorithms that only use the routing capacity of a half-MIMD are called **uni-axial**. Algorithms for the MIMD will be called **bi-axial**.

The MIMD may be stronger than realistical. If MIMD algorithms are directly run on an SIMD, then they are slowed-down by a factor four. Often specific SIMD algorithms perform much better. The half-MIMD has a certain universality: running half-MIMD algorithms with a slow-down factor two on an

*Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. E-mail: jopsi@mpi-sb.mpg.de. This research was partially supported by EC Cooperative Action IC-1000 (Project ALTEC: Algorithms for Future Technologies).

SIMD gives competitive results; on the other hand, for certain problems, half-MIMD algorithms perform on an MIMD almost as good as MIMD algorithms. There are other reasons to consider algorithms for the half-MIMD: on an MIMD two of these algorithms can be perfectly overlapped; in MIMD permutation routing algorithms (see [8, 1]), 'non-critical' packets are uni-axially scattered while 'critical' packets are routed orthogonally without loss of time.

Indexings. Several recent sorting algorithms [2, 7, 5] were designed for (blocked) snake-like row-major indexings. This indexing may be good, but in many cases it is desirable to have the packets in the more natural row-major (column-major) order. Furthermore, sorting in snake-like order is unsuited as a subroutine for scattering algorithms.

In the one-packet model considered by Schnorr and Shamir [12], the lower bound for row-major sorting is higher than for sorting in snake-like row-major order. In our model a PU may hold a constant number of packets and packets may be copied. From the results of this paper it follows that in this model, sorting in row-major order is not substantially harder than sorting in snake-like row-major order. Only on the half-MIMD, where there are no matching lower bounds, the situation is not yet fully clarified. Proving non-trivial lower bounds is hard, as such a proof should at least involve the queue size and the uni-axiality: if the queue size would not play a role, then the greedy algorithm (uni-axial!) could be used.

Results. This paper gives numerous improvements for row-major sorting. They are resumed in Table 1.

	Uni-Axial		**Bi-Axial**
k	all n	large n	large n
1	$4^1/_2 \cdot n$	$2^1/_2 \cdot n$	$2 \cdot n$
2	$5^3/_4 \cdot n$	$3 \cdot n$	$2^1/_2 \cdot n$
k	$(2 \cdot k + 2^1/_2) \cdot n$	$k \cdot n$	$k/2 \cdot n$

Table 1: Run times for k-k sorting in row-major order. In the results for large n, we left away the lower-order terms.

The queue sizes in the algorithms for 1-1 and 2-2 sorting range between four and nine, and in the k-k sorting the queue sizes are $k + 1$ or $k + 2$. Actually the results for large n are much more general: they do not just hold for sorting in row-major order, but for sorting with respect to any indexing that is piecewise-continuous (see Definition 1 in Section 2).

Theoretically the result for large n are the most appealing. So far, the fastest bi-axial row-major sorting algorithm has $T = 2^1/_4 \cdot n + o(n)$ and $Q = \mathcal{O}(1)$. It was recently designed by Krizanc and Narayanan [6]. However, this algorithm works only for the subproblem that all the keys are 0 or 1 (though some extension seems possible). For sorting in blocked snake-like row-major order $T = 2 \cdot n + o(n)$ was achieved first by Kaklamanis and Krizanc [2] with a randomized algorithm, and then also deterministically by Kaufmann, Sibeyn and Suel [5]. These algorithms are considerably more involved then the algorithm of this paper, and have queue sizes around 20. The best uni-axial row-major sorting algorithm so far is a modification of the algorithm of Schnorr and Shamir. It takes $4 \cdot n + o(n)$ steps.

Most current communication algorithms strive for $T = \alpha \cdot n + o(n)$, with α as small as possible. This completely neglects the fact that actual meshes tend to be of fairly moderate sizes, for which the $o(n)$ term easily may dominate. Typically this term gives the number of steps for several sorting and rearrangement operations in submeshes of size $n^{2/3} \times n^{2/3}$ or $n^{3/4} \times n^{3/4}$. Even when this term is just $10 \cdot n^{2/3}$, then still it exceeds n for all $n < 1000$. This clearly expresses the utmost importance of algorithms with a routing time not involving any hidden terms. Therefore, is our uni-axial $4^1/_2 \cdot n$ row-major sorting algorithm of great *practical* importance. A sorting or scattering time which can be expressed as $T \leq \alpha \cdot n$, for all n, is even relevant in a *theoretical* setting: in recursive or divide-and-conquer algorithms, in which these algorithms are used as subroutines, the submeshes on which they are applied are small.

The first near-optimal algorithm for k-k sorting was discovered by Kaufmann and Sibeyn [4]. Then in [7] by Kunde and slightly later also in [5], deterministic versions of this randomized algorithm were described. All these algorithms use a blocked snake-like row-major indexing. In this paper we present the first near-optimal algorithm for k-k sorting in row-major order.

The remainder of the paper is organized as follows: in Section 3 we give the algorithms for uni-axial row-major sorting for all n. Then we introduce in Section 4 the 'desnakification' of the k-k sorting algorithm for large n. This powerful technique is then applied in Section 5 for very fast uni-axial 1-1 sorting, for near-optimal bi-axial 1-1 sorting, and finally for 2-2 sorting.

2 Preliminaries

Basics of Routing and Sorting. We speak of **edge contention** when several packets residing in a PU have to be routed over the same connection. Contentions are resolved using a priority scheme. We apply the **farthest-first strategy**, which gives priority to the packets that have to go farthest. For the analysis of the routing on higher dimensional meshes we need the 'routing lemma' for routing a distribution of packets on a one dimensional mesh [3] and the corresponding 'sorting lemma' [1]. Define for a given distribution of packets over the PUs $h_{\text{right}}(i, j) = \#\{$*packets passing from left to right through both* P_i and $P_j\}$, where P_i denotes the PU with index i. Define $h_{\text{left}}(j, i)$ analogously.

Lemma 1 *Routing a distribution of packets on a linear array with n PUs, using the farthest-first strategy, takes* $\max_{i<j}\{\max\{h_{\text{right}}(i, j), h_{\text{left}}(j, i)\} + j - i - 1\}$ *steps. This bound is sharp. When the packets are evenly distributed, then the same bound can be achieved for sorting.*

Because of the distance a packet may have to go $2 \cdot n - 2$ steps is a lower bound for any general routing or sorting problem on the two-dimensional mesh.

A **0-1 distribution**, is a distribution of packets that all have key zero or one. In a 0-1 distribution a row is called **dirty**, if it contains both zeros and ones. In our analyses we frequently use the so-called '0-1 lemma' (see [9]), that states that under certain, in our case satisfied, conditions a sorting algorithm is correct iff it sorts any 0-1 distribution.

Indexings. The PUs can be indicated by giving their coordinates within the mesh, the PU at position (i, j), $0 \le i, j < n$, is denoted $P_{i,j}$. Here position $(0, 0)$ lies in the upper-left corner. In the common row-major indexing $P_{i,j}$ has index $i \cdot n + j$. In the column-major indexing $P_{i,j}$ has index $i + j \cdot n$. In the reversed row-major indexing $P_{i,j}$ has index $i \cdot n + (n - j)$. In the snake-like row-major indexing, the indexing of the odd rows is reversed. For a given indexing we denote the PU with index i, $0 \le i < n^2$, by P_i.

If we consider a k-k sorting, then there are two natural ways to index the $k \cdot n^2$ destination locations. Our default is a **non-layered** indexing. In this case, location r in P_i, $0 \le r < k$, $0 \le i < n^2$, has index $k \cdot i + r$. In the case of a non-layered row-major indexing, this is the index as if we have an $n \times k \cdot n$ mesh in row-major order, see Figure 1 on the left. Alternatively, in a **layered** indexing, location r in P_i has index $r \cdot n^2 + i$. In the case of row-major sorting we use a particular **semi-layered** indexing, under which location $P_{i,j}$, has index $(i + r) \cdot n + j$. This is the index as if we have a $k \cdot n \times n$ mesh in row-major order, see Figure 1 on the right.

A row i is said to be **sorted rightwards** if the packets stand in increasing order from $P_{i,0}$ to $P_{i,n-1}$. Analogously, rows can be sorted **leftwards** and columns **downwards** and **upwards**.

An indexing is called **continuous** if for all i, $1 \le i \le n - 2$, P_{i-1} and P_{i+1} are adjacent to P_i in the

0 1	2 3	4 5	6 7
8 9	10 11	12 13	14 15
16 17	18 19	20 21	22 23
24 25	26 27	28 29	30 31

0 4	1 5	2 6	3 7
8 12	9 13	10 14	11 15
16 20	17 21	18 22	19 23
24 28	25 29	26 30	27 31

Figure 1: Non-layered indexing (left), and semi-layered indexing (right), for $k = 2$, $n = 4$.

mesh. Snake-like indexings are continuous.

Definition 1 *An indexing is called* **piecewise-continuous** *with parameter s if for every i,* $0 \le i < n^2$, *there is an interval* $\mathcal{I}_i \subset [0, n^2 - 1]$, *with* $i \in \mathcal{I}_i$ *and* $\#\mathcal{I}_i \ge s$, *such that for all* $j \in \mathcal{I}_i$, P_j *is adjacent to* P_{j-1} *and* P_{j+1}, *whenever* $j - 1, j + 1 \in \mathcal{I}_i$.

The row-major indexing is piecewise-continuous with parameter n. One of the achievements of this paper is to show that for efficient sorting it is sufficient to have *piecewise*-continuous indexings.

Subdivisions. In our algorithms the mesh is divided in regular $s \times s$ submeshes. Let $m = n/s$. The submeshes are indexed as the PUs: starting with $(0, 0)$ in the upper-left corner. We refer to submesh (i, j) by $B_{i,j}$. Define **row-bundle** i to consist of the PUs in $\cup_{j=0}^{m-1} B_{i,j}$. Likewise, **column-bundle** j consists of $\cup_{i=0}^{m-1} B_{i,j}$. Additionally the mesh is subdivided in **sections**. A section is a subset of the PUs with consecutive indices. In Section 4 and Section 5,

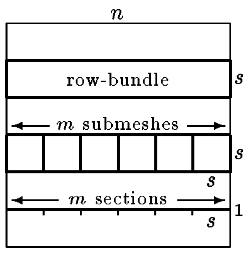


Figure 2: Subdivisions for the case $s = n/6$, $m = 6$.

we use sections of length s, and there section l, denoted S_l, $0 \le l < m \cdot n$, consists of the PUs with index $s \cdot l, \ldots, s \cdot (l + 1) - 1$. Under a row-major indexing the sections regularly subdivide the rows and the submeshes. All subdivisions are depicted in Figure 2.

Definition 2 *An m-***way merge** *is a procedure that turns a mesh that is divided in* m^2 *sorted* $s \times s$ *submeshes into a sorted* $n \times n$ *mesh.*

3 Uni-Axial Sort for Small n

3.1 Powers of Two

Lemma 2 *Uni-axial sorting in arbitrary order can be performed on 2×2 meshes in 3 steps, with queue size two.*

Proof: Perform gossiping (all-to-all routing) along rows and then along columns. This takes three steps. A PU that finally should hold the packet with rank 0 or 1, needs to conserve only the two smallest packets, the other PUs only the two largest packets. $\square$

For $n = 2^l$, $l > 1$, we use an optimized merge-sort algorithm combining several recent techniques and adding some new ideas. Initially we assume that we have four sorted $n/2 \times n/2$ submeshes: those in the left half in row-major order; those in the right half in reversed row-major order. The first merge sort algorithm with the optimal time *order* was given by Thompson and Kung [13]. Our merging consists of five easy steps:

Algorithm MERGE

1. In the left half, shift the packets $n/4$ steps to the right. In the right half, shift the packets $n/4$ steps to the left.

2. In the central $n/2$ columns, sort the packets downwards.

3. Copy the smallest packet in every $P_{i,j}$, $0 < i \leq n-1$, $n/4 \leq j \leq 3/4 \cdot n - 1$, to $P_{i-1,j}$. Copy the largest packet in every $P_{i,j}$, $0 \leq i < n-1$, $n/4 \leq j \leq 3/4 \cdot n - 1$, to $P_{i+1,j}$.

4. In every row, sort the section of the row that lies in the central $n/2$ columns. If this submesh is going to be the right half of a larger mesh in the next merge, then the sorting is leftwards, otherwise rightwards.

5. Throw away the packets in $P_{i,j}$ with $j \in [n/4, 3/8 \cdot n - 1] \cup [5/8 \cdot n, 3/4 \cdot n - 1]$. For any $P_{i,j}$, with $3/8 \cdot n \leq j \leq 5/8 \cdot n - 1$, send the packet with rank r, $0 \leq r \leq 3$, to $P_{i,4 \cdot (j - 3/8 \cdot n) + r}$.

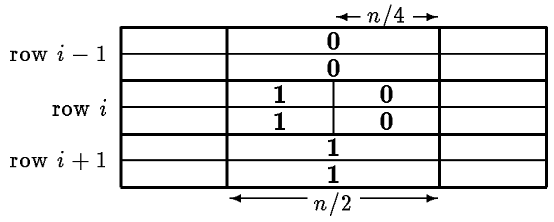
We analyze the routing time and the correctness of MERGE. Step 1 takes $n/4$ steps, Step 2 can be performed in n steps, and Step 3 takes a single step. This step can easily be made to coincide with the last step of the sorting. Its purpose is expressed by

Lemma 3 *After Step 2 all packets that actually should be in a row can be found either in the row itself, or among the smallest packets of the row below, or among the largest packets of the row above.*

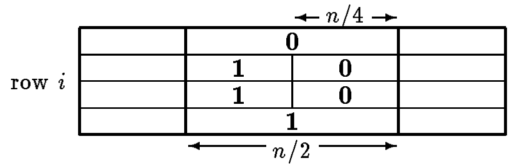
Proof: First we consider a modified problem. Suppose that initially four $n/2 \times n/2$ submeshes stand above each other in an $2 \cdot n \times n/2$ mesh. Two of these submeshes are sorted in row-major order, the other two in reversed row-major order. Consider a 0-1 distribution. It is easy to check that after sorting the columns of this mesh, there are at most two dirty rows. These dirty rows can be resolved as follows: copy every row to the row above and the row below; sort the rows; spread the packets from the central $n/3$ columns. In the real problem every two rows of the high and narrow mesh are compressed in one row in which every PU in the center holds two packets. $\square$

Lemma 4 *Step 4 can be performed in $3/4 \cdot n$ steps.*

Proof: For the number of required steps we analyze the worst possible 0-1 distributions after Step 2. These are of the following form:



After Step 3, we have the following distribution in row i:



According to Lemma 1, sorting this row takes $3/4 \cdot n$ steps. $\square$

Finally, Step 5 takes $3/8 \cdot n$ steps. Hence,

Lemma 5 MERGE *takes less than $2^3/_8 \cdot n$ steps. The queue size is at most four.*

The algorithm might be further improved by performing Step 4 and Step 5 together more efficiently. After $n/2 - 1$ steps we are sure that the packets with the largest and with smallest keys have reached their destination. From that moment on we can kill on both sides one packet every further step. It is not clear how this can be exploited.

Starting with sorted 2×2 meshes, MERGE can be used repeatedly for sorting on an $n \times n$ mesh. Call this algorithm SORT. We have

Theorem 1 *For all $n = 2^l$,* SORT *performs row-major sorting on an $n \times n$ mesh in $4^3/_4 \cdot n$ steps.* SORT *is uni-axial, and the queue size is four.*

Proof: Summing the number of steps required for all merges, we find that the sorting takes less than $3 + 2^3/_8 \cdot (4 + 8 + \cdots + n) < 2^3/_8 \cdot n \cdot \sum_{i=0} 2^{-i}$ steps. $\square$

3.2 Powers of Two, Three, ...

We derived an efficient 1-1 sorting algorithm for $n = 2^l$. However, in practice processor networks may not have such beautiful side lengths. Furthermore, some algorithms in which sorting is used as a subroutine, e.g., the algorithms of [1, 15] specifically require that $n = 5^l$ or 6^l. In principle we could use SORT by rounding n up to the nearest power of 2. But, this might give sorting times that are almost twice as large as necessary. In this section we present m-way merge algorithms, which perform good for $m \leq 5$. By combining them, we can efficiently sort $n \times n$ meshes for arbitrary n.

$m \times m$ Meshes. Consider an $m \times m$ mesh. Suppose that we want to sort this mesh in (reversed) row-major order. A simple algorithm performs well:

1. In all rows i, concentrate the packets in $P_{i,\lfloor m/2 \rfloor}$.

2. Sort the packets in column $\lfloor m/2 \rfloor$ downwards.

3. In all rows i, spread the packets over the row.

Lemma 6 *Uni-axial sorting on $m \times m$ meshes can be performed in $m^2/2 + m$ steps, for m even, and $m \cdot (m+1)/2$ steps, for m odd, with queue size m.*

Proof: The steps take $\lfloor m/2 \rfloor$, $m \cdot \lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor$ steps, respectively. □

Larger n. The algorithm for uni-axial sorting in row-major order on $n \times n$ meshes for $n = m^l$ is analogous to the algorithm for $n = 2^l$: $n/m \times n/m$ submeshes are appropriately sorted, and the submeshes are merged. For this merging, we can proceed as in MERGE: wiping all submeshes together, sorting the columns, etc. However, algorithms of this type are not very suited for an m-way merge with $m \geq 3$: the number of dirty rows equals $\lceil m^2/2 \rceil$, which leads to long queues, and rapidly growing time to resolve them. It is better first to sort the rows-bundles, then to merge the sorted row-bundles. In this way the number of dirty rows is limited to $\lceil m/2 \rceil$. For $m = 2$, this approach is slower, but for larger m it is faster.

The algorithm, for sorting $m^l \times m^l$, $l > 1$, meshes in row-major order starts by sorting recursively all $m^{l-1} \times m^{l-1}$ submeshes. For even m, in every row-bundle $m/2$ submeshes are sorted in row-major order, and $m/2$ in reversed row-major order. For odd m, in the highest $\lceil m/2 \rceil$ row-bundles, $\lceil m/2 \rceil$ submeshes are sorted in row-major order, and in the lowest $\lfloor m/2 \rfloor$ row-bundles, $\lfloor m/2 \rfloor$. The other submeshes are sorted in reversed row-major order. In Figure 3 we give an example for $m = 5$. Then we perform the following merge algorithm. The **central column-bundle**, denotes the subset of columns j, with $(m-1)/(2 \cdot m) \cdot n \leq j < (m+1)/(2 \cdot m) \cdot n$.

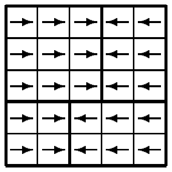


Figure 3: For sorting in row-major order the rightward (leftward) arrows indicate submeshes that are sorted in (reversed) row-major order.

Algorithm MERGE-m

1. Shift the packets in the column-bundles as blocks to central column-bundle.

2. In the central column-bundle, sort the sections of the columns that lie within the row-bundles downwards.

3. Let $q = \lceil m/2 \rceil - 1$. In the central column-bundle, for all i, $0 \leq i < n$, $i \neq 0, n/m, \ldots, (m-1)/m \cdot n$, copy the smallest q packets in $P_{i,j}$, to $P_{i-1,j}$; for all $i \neq n/m - 1, 2/m \cdot n - 1, \ldots, n-1$, copy the largest q packets in $P_{i,j}$, to $P_{i+1,j}$.

4. In the central column-bundle, for all i, $0 \leq i < n$, sort the section of row i. If $i < \lceil m/2 \rceil / m \cdot n$, then sort rightwards, else sort leftwards.

5. In every row, throw away the $q \cdot n/m$ packets with the smallest and with the largest keys. The remaining n packets stand in the central $\lceil n/(m + 2 \cdot q) \rceil$ PUs. Spread these packets over the central $\lceil n/2 \rceil$ PUs, such that they come to stand in semi-layered order. If m is odd, then one packet with key ∞ should be added on the right.

6. In the central $\lceil n/2 \rceil$ columns, sort the packets downwards.

7. In the central $\lceil n/2 \rceil$ columns, for all $0 < i \leq n-1$, copy the smallest q packets in $P_{i,j}$, to $P_{i-1,j}$; for all $0 \leq i < n-1$, copy the largest q packets in $P_{i,j}$, to $P_{i+1,j}$.

8. In the central $\lceil n/2 \rceil$ columns, sort all rows rightwards.

9. In every row i, $0 \leq i < n$, throw away the $q \cdot \lceil n/2 \rceil$ packets with the smallest and with the largest keys. If m is odd, then throw away one more packet with a large key. The remaining n packets stand in the central $\lceil n/(2 + 2 \cdot q) \rceil$ PUs. Send the packet with rank j to $P_{i,j}$.

The correctness of MERGE-m is obvious: after Step 2 and Step 6, there are $q+1$ dirty rows in a 0-1 distribution. These are resolved by the steps that follow. For $m > 6$, in Step 7, packets from more than one row below and one row above every row should be copied into it. The steps required for Step 3 and Step 7 can be saved by modifying the final q steps of the preceding steps. The queue size equals $Q = m + 2 \cdot q$. That is, $Q = 2 \cdot m - 2$, for m even, and $Q = 2 \cdot m - 1$, for m odd. The time consumption of the steps are listed

Step	**Time Consumption**
	even m
1	$(m-1)/(2 \cdot m) \cdot n$
2	$\lfloor n/4 \rfloor$
3	0
4	$\max\{n/m, \lfloor \frac{2 \cdot m - 2}{5 \cdot m - 4} \cdot n \rfloor\}$
5	$\max\{\lfloor n/4 \rfloor, \lfloor \frac{m-2}{2 \cdot m - 2} \cdot n \rfloor\}$
6	n
7	0
8	$\max\{\lfloor n/2 \rfloor, \lfloor m/6 \cdot n \rfloor\}$
9	$(m-1)/(2 \cdot m) \cdot n$
	odd m
1	$(m-1)/(2 \cdot m) \cdot n$
2	$\lfloor (1 - 1/m^2) \cdot n/4 \rfloor$
3	0
4	$\max\{n/m, \lfloor \frac{2 \cdot m^2 - 3 \cdot m + 1}{5 \cdot m^2 - 3 \cdot m} \cdot n \rfloor\}$
5	$\lfloor \frac{2 \cdot m - 3}{4 \cdot m - 2} \cdot n \rfloor$
6	n
7	0
8	$\max\{\lfloor n/2 \rfloor, \lfloor \frac{m^2 - 1}{3 \cdot m + 1} \cdot \lceil n/2 \rceil \rfloor\}$
9	$m/(2 \cdot m + 2) \cdot n$

Table 2: The time consumptions of the steps of MERGE-m.

in Table 2. These results either equal the maximal distance packets may have to go, or the number of packets that may have to move through a single connection. We analyze the most difficult steps. In order not to get an excessive amount of notation, we consider the case $m = 5$. For other odd m the analysis is analogous; for even m, the analysis is simpler.

Lemma 7 *For $m = 5$, Step 2 can be performed in $\lfloor 6/25 \cdot n \rfloor$ steps.*

Proof: As an example for other proofs, which are handled quite sketchy, we give a detailed proof here. Consider the sorting in the section of column j in row-bundle 0. We refer to the PUs in this section by P_i, $0 \leq i < n/5$. Five sorted submeshes, $B_0, \ldots, B_4$, contribute their packets to this section. Suppose that all packets have key zero or one, and let m_l, $0 \leq a_l \leq n/5$, be the number of ones contributed by B_l. We apply the analogue of Lemma 1 for sorting in columns, and determine the distribution for which $T(i_i, i_2) = h_{\mathrm{down}}(i_1, i_2) + i_2 - i_1 - 1$ is maximal. For $a_0 = a_1 = a_2 = 0$, and $a_3 = a_4 = n/5$, we have $T(3/25 \cdot n - 1, 3/25 \cdot n) = 6/25 \cdot n$.

In the remainder of the proof, we show that there are no distributions that give larger $T(i_1, i_2)$. The reader may want to skip this. $T(i_1, i_2)$ is not decreased when ones are moved upwards. Hence, we may assume that the packets in a section are arranged as if they were sorted in column-major order: we have a distribution as in Figure 4 on the

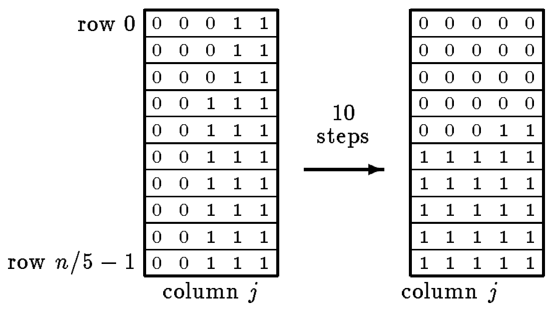
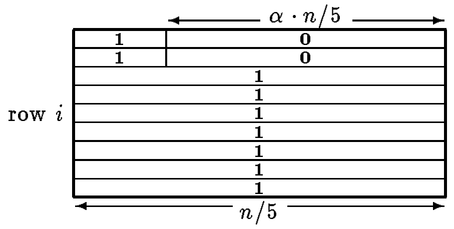


Figure 4: A (bad) 0-1 distribution in a section of a column: on the left before Step 2, on the right after Step 2.

left. Let $a = a_0 + \cdots + a_4$, be the total number of ones in the section. If $n/5 - a \bmod (n/5) > i_1$, then we can remove $a \bmod (n/5)$ ones without decreasing $h_{\mathrm{down}}(i_1, i_2)$: no ones in a row i with $i \leq i_1$ are removed. If $n/5 - a \bmod (n/5) \leq i_1$, then we can add $n/5 - a \bmod (n/5)$ ones without decreasing $h_{\mathrm{down}}(i_1, i_2)$: the number of ones that has to move from a row i, with $i \leq i_1$ to a row i' with $i' \geq i_2$, is at least as large as before. Thus, we may assume that $a = l \cdot n/5$, for some $0 \leq l \leq 5$. The cases $l = 0$ and $l = 5$ are trivial, so we suppose that $1 \leq l \leq 4$. Denote by $zero_0$, the number of zeros in rows i, with $i \leq i_1$, by $zero_1$, the number of zeros in rows i, with $i_1 < i < i_2$, and by $zero_2$, the number of zeros in rows i, with $i_2 \leq i$. Define one_0, one_1 and one_2 analogously. Clearly $h_{\mathrm{down}}(i_1, i_2) \leq \min\{zero_2 - one_1, one_0 - zero_1\}$. It follows, because $one_1, zero_1 \geq i_2 - i_1 - 1$ (here we need that $0 < l < 5$), that taking $i_2 > i_1 + 1$, does not have a positive effect on $T(i_1, i_2)$. For $i_2 = i_1 + 1$, we get $T(i_1, i_2) \leq \min\{zero_2, one_0\} = \min\{(n/5 - i_2) \cdot (5 - l), i_2 \cdot l\}$. Solving gives $i_2 = (1 - l/5) \cdot n/5$. □

Lemma 8 *For $m = 5$, Step 4 can be performed in $\lfloor 18/55 \cdot n \rfloor$ steps.*

Proof: Consider a 0-1 distribution. There are at most three dirty rows after Step 2. However, by the special way the submeshes are sorted it is not possible that after Step 3 one row holds three layers of packets that are sorted falsely. The worst possible distributions in some row i with $i < 3/5 \cdot n$, are of the following form:



The sorting time is maximal for $\alpha = 9/11$. $\square$

The analysis of Step 8 is analogous.

Lemma 9 *For $m = 5$, Step 5 can be performed in $\lfloor 7/18 \cdot n \rfloor$ steps.*

Proof: At the beginning of Step 5, the n packets stand sorted rightwards in the central $\lceil n/9 \rceil$ PUs of a row. $2/9 \cdot n$ packets stay there, $\lfloor 7/18 \cdot n \rfloor$ packets have to move out of this section leftwards and rightwards. The maximal distance any packet has to go, is $\lfloor n/4 \rfloor$: consider a packet p with rank $r < n/2$. p comes in the 'lower layer'. The packet with $r = n/2 - 1$ has to move farthest: $\lfloor n/4 \rfloor$ steps, from column $\lfloor n/2 \rfloor$ to column $\lfloor 3/4 \cdot n \rfloor$. $\square$

Substituting in Table 2, we obtain for $2 \leq m \leq 6$ the following time consumptions (omiting the factor n):

Step	**Time Consumption**				
m	2	3	4	5	6
1	1/4	1/3	3/8	4/10	5/12
2	1/4	2/9	1/4	6/25	1/4
4	1/2	1/3	3/8	18/55	5/13
5	1/4	3/10	1/3	7/18	2/5
6	1	1	1	1	1
8	1/2	1/2	2/3	3/4	1
9	1/4	3/8	3/8	5/12	5/12
total	3	3.07	$3^3/8$	3.53	3.87

Starting with sorted $m \times m$ meshes and then repeating $l - 1$ times MERGE-m, meshes of size $m^l \times m^l$ can be sorted. Call this algorithm SORT-m. In Table 3, we give an overview of its performance for $2 \leq m \leq 6$. For $m = 6$, the performance starts to deteriorate. We check the result for $m = 5$:

m	2	3	4	5	6
T	$6 \cdot n$	$4.61 \cdot n$	$4^1/2 \cdot n$	$4.42 \cdot n$	$4.65 \cdot n$
Q	2	5	6	9	10

Table 3: Run times and queue sizes of uni-axial row-major sorting algorithms for $n = m^l$.

Theorem 2 *For all $n = 5^l$, SORT-5 performs row-major sorting on an $n \times n$ mesh in $4.41 \cdot n$ steps. SORT-5 is uni-axial, and the queue size is nine.*

Proof: Summing the number of steps required for all merges, we find that the sorting takes less than $15 + 3.53 \cdot (25 + 125 + \cdots + n) < 3.53 \cdot n \cdot \sum_{i=0} 5^{-i}$ steps. The queue size was analyzed before. $\square$

Powers of Two. Using MERGE 4, we obtain an alternative algorithm for sorting $n \times n$ meshes with $n = 2^l$: if l is odd, then we start by sorting the 2×2 submeshes; if l is even, then first the 4×4 submeshes are sorted. Hereafter MERGE-4 is applied repeatedly. This gives

Theorem 3 *Row-major sorting on an $n \times n$ mesh with $n = 2^l$, $l > 0$, can be performed in $4^1/2 \cdot n$ steps by a uni-axial algorithm, with queue size six.*

Proof: The total time for the sorting is less than $3 + 3^3/8 \cdot (8 + 32 + \cdots + n)$, for odd l; and less than $12 + 3^3/8 \cdot (16 + 64 + \cdots + n)$, for even l. $\square$

We already have three algorithms for the case $n = 2^l$, one more will be presented in Section 3.4. They show a trade-off between run time and queue size, see Table 4.

T	Q	Section
$6 \cdot n$	2	3.2
$5^2/3 \cdot n$	3	3.4
$4^3/4 \cdot n$	4	3.1
$4^1/2 \cdot n$	6	3.2

Table 4: Run times and queue sizes of uni-axial row-major sorting algorithms for $n = 2^l$.

3.3 Mixed Powers

Sorting on $n \times n$ meshes for arbitrary n, can be performed by approximating n by the closest number of the form $2^{l_2} \cdot 3^{l_3} \cdot 5^{l_5}$, and then using the basic three-way, four-way and five-way merges:

```
    Algorithm SORT-ALL($n$)
Determine the minimal $n' = 2^{l_2} \cdot 3^{l_3} \cdot 5^{l_5} \geq n$;
{ Assume that $l_3 > 0$ }
if $l_2$ is odd then sort the $6 \times 6$ submeshes
else sort the $3 \times 3$ submeshes;
repeat $l_3 - 1$ timesMERGE-3;
repeat $\lfloor l_2/2 \rfloor$ timesMERGE-4;
repeat $l_5$ timesMERGE-5.
```

If $l_3 = 0$, then first the 2×2 or the 5×5 submeshes are sorted. MERGE-3 is performed first, when the meshes are still small, because it is the least efficient. SORT-ALL is sophisticated enough to achieve the following interesting result:

Theorem 4 SORT-ALL *performs uni-axial row-major sorting on $n \times n$ meshes in $4.90 \cdot n$ steps, for all n. The queue size is at most nine.*

Proof: For $n < 90$, the proof can be given by checking the n one-by-one. For $n \in [90, 180]$, we find the following numbers of the correct form: $\mathcal{N}_{90,180} = \{90, 96, 100, 108, 120, 125, 128, 135, 144, 150, 160, 162, 180\}$. The worst performance gives $n = 163$ with $n' = 180 = 5 \cdot 4 \cdot 3^2$. Hence, its sorting time can be estimated on $(3.53 + 3.375/5 + 3.07/20 \cdot (1 + 1/3)) \cdot 180 < 4.41 \cdot 180 < 4.87 \cdot 163$. If $n' \in \mathcal{N}_{90,180}$, then $2 \cdot n' \in \mathcal{N}_{180,360}$, $4 \cdot n' \in \mathcal{N}_{360,720}$, and so on. This implies, that the ratios between the n and the n' do not further increase (on the contrary). Also, if $T_{\text{sort}}(n') \leq \alpha \cdot n'$, for some constant α, then $T_{\text{sort}}(2 \cdot n') \leq \alpha \cdot 2 \cdot n'$. Hence, we may conclude that for any n, the sorting can be performed in $180/162 \cdot 4.41 \cdot n$. □

For larger n it may be advantageous not to round up to the minimal $n' = 2^{l_2} \cdot 3^{l_3} \cdot 5^{l_5}$, when l_3 is large. It is always possible to get $l_3 < 5$, by replacing $3^5 = 243$ by $3^5 \cdot 2 = 250$. This gives a considerable reduction of the sorting time.

3.4 k-k Sorting

We present an algorithm for uni-axial k-k sorting in row-major order. We assume that $n = 2^l$, for some $l > 0$. For large n better performance is achieved by the uni-axial version of the algorithm of Section 4 which requires $\max\{4 \cdot n, k \cdot n\} + \mathcal{O}((k \cdot n)^{5/6})$. But, the here presented algorithm is far better for small n.

The merging is almost the same as MERGE of Section 3.1. We assume that the four submeshes are sorted in semi-layered row-major order on the left, and semi-layered reversed row-major order on the right.

Algorithm KKMERGE

1. $P_{i,j}$, $0 \leq i, j < n$, sends its packet with rank r, $0 \leq r < k$, to $P_{i,(j+n/2) \bmod n}$ if $\text{odd}(k \cdot i + r + j)$.

2. In all columns, sort the packets downwards.

3. In every $P_{i,j}$, $0 < i \leq n-1$, $0 \leq j \leq n-1$, copy the smallest packet to $P_{i-1,j}$. In every $P_{i,j}$, $0 \leq i < n-1$, $0 \leq j \leq n-1$, copy the largest packet to $P_{i+1,j}$.

4. Sort the rows. If this submesh is going to be the left half of a larger mesh in the next merge, then the sorting is rightwards, otherwise leftwards.

5. In every row, throw away the n packets with the smallest and the n packets with the largest indices. If this is the final merge step, then spread the remaining $k \cdot n$ packets that stand in every row. Else route the packets to the destinations as given by Step 1 of the next merge, and continue with Step 2.

For the correctness of KKMERGE it is important that, by the semi layered indexing, our merging corresponds to a 1-1 merge on a $k \cdot n \times n$ mesh. It is easy to see that if we have a 0-1 distribution, that then after Step 2 there are at most two dirty rows.

For $k = 1$ the algorithm is correct but less efficient than the algorithm of Section 3.1. It can be shown that 1-1 sorting takes $5^2\!/\!_3 \cdot n$ steps. We further assume that $k > 1$. Step 1 takes $k \cdot n/4$ steps. However, it only has to be performed during the merge of 2×2 meshes. So, for determining the time order we can concentrate on the other steps. Step 2, takes $k \cdot n/2$. Step 3 can be overlapped with the last step of Step 2.

Lemma 10 *Step 4 takes at most $k \cdot n/4 + n/2$ steps.*

Proof: It is easy to check that, by the semi-layered indexing and by the way the packets are selected in Step 1, there are no 0-1 distributions that after Step 3 result in such homogeneous blocks of zeros on one side and ones on the other side as in the worst-case example of the proof of Lemma 4. For example, for $k = 4$, a worst-case distribution in row i is like

	00000000000000000000000000000000
	00000000000000000000000000000000
row i	01010101010101010101010101010101
	01010101010101010101010101010101
	11111111111111111111111111111111
	11111111111111111111111111111111

For this example Step 4 takes $3/2 \cdot n$ steps. □

In the final merge, Step 5 takes $k/(k+2) \cdot n$ steps, otherwise $\frac{k}{2} \cdot \frac{k+3}{k+2} \cdot n$ steps. This step is that expensive because that many packets have to move out of the central $k/(k+2) \cdot n$ PUs through a single connection.

Lemma 11 *An intermediate application of* KKMERGE *can be performed in $(5 \cdot k^2 + 14 \cdot k + 4)/(4 \cdot k + 8) \cdot n$ steps, the final application in $(3 \cdot k^2 + 12 \cdot k + 4)/(4 \cdot k + 8) \cdot n$ steps.*

Let KKSORT1 be the k-k sorting algorithm based on KKMERGE.

Theorem 5 *For all $k \geq 2$,* KKSORT1 *performs uni-axial k-k sorting in row-major order on $n \times n$ meshes in $(4 \cdot k^2 + 13 \cdot k + 4)/(2 \cdot k + 4) \cdot n$ steps, with queue size $k + 2$.*

Proof: We start with sorted PUs. It takes $k/2$ steps to obtain the situation at the beginning of Step 2 of the merge in 2×2 meshes. Thus, the general estimate for k-k sorting on $n \times n$ meshes is $k/2 + (5 \cdot k^2 + 14 \cdot k + 4)/(4 \cdot k + 8) \cdot (2 + 4 + \cdots + n/2) + (3 \cdot k^2 + 12 \cdot k + 4)/(4 \cdot k + 8) \cdot n < (4 \cdot k^2 + 13 \cdot k + 4)/(2 \cdot k + 4) \cdot n$ □

From Theorem 5 we computed the results in Table 5. For small n they are extremely competitive, even though asymptotically k-k sorting can be performed twice as fast with the algorithm of Section 4.

k	T	Q
2	$5^3/_4 \cdot n$	4
3	$7^9/_{10} \cdot n$	5
4	$10 \cdot n$	6
k	$(2 \cdot k + 2^1/_2) \cdot n$	$k+2$

Table 5: Run times and queue sizes for uni-axial k-k sorting in row-major order.

4 k-k Sorting for Large n

Earlier algorithms for k-k sorting [4, 7, 5] work according to the following basic scheme:

1. Route all packets to random destinations.

2. Estimate the ranks of the packets by local comparisons.

3. Route all packets to their preliminary destinations.

4. Rearrange the packets locally to bring them to their final destinations.

In the version of [5], the mesh is divided in $s \times s$ submeshes with $s = n^{2/3}/k^{1/3}$, and the randomization of Step 1 is replaced by sorting the packets in the submeshes and unshuffling them regularly over the submeshes. Step 2 is performed by sorting within the submeshes. Step 4 is performed by sorting pairs of adjacent submeshes. On an MIMD the total sorting time is $k \cdot n/2 + \mathcal{O}(k^{2/3} \cdot n^{2/3})$. As the algorithm is given, Step 4 requires that the indexing is continuous. In this section we introduce a novel technique, we call it **desnakification**, to handle the final local sorting such that piecewise-continuous indexings are allowed.

The continuity of the indexing is required only for sorting together pairs of submeshes with consecutive indices. Sorting such pairs of submeshes is necessary and sufficient because the estimate of the rank in Step 2 is accurate up to one submesh. So it may happen that after Step 3, a packet is not present in its destination submesh B_i, but resides in the preceding submesh B_{i-1} or the succeeding submesh B_{i+1}. However, this is easy to overcome: send for all packets p, of which the destination submesh is not uniquely determined, a copy to both submeshes in which its destination may lie. Now it is sufficient to sort within the submeshes. If for B_i the numbers cl, of packets that actually belong in B_{i-1}, and ch, of packets that belong in B_{i+1}, are exactly known, then the smallest cl and largest ch packets in B_i are thrown away, and the remaining packets are redistributed within B_i. All this is very similar to (and was inspired by) the way dirty rows are resolved in the algorithms of Section 3. The only possible problem is, that routing the copies might slow-down the algorithm.

We work the desnakification out in detail for bi-axial sorting. In order to bound the number of copies, we take the submeshes larger than in [5]: in our case $s = n^{5/6}/k^{1/6}$, and $m = n/s = k^{1/6} \cdot n^{1/6}$. We suppose that the indexing is piecewise-continuous with parameter s. For the sake of a simple exposition we assume that the mesh is divided in sections of length s, each of which is fully contained in a single submesh. The algorithm proceeds as follows:

Algorithm KKSORT2

1. In each submesh, sort the packets. The intermediate destination of a packet p with rank r, $0 \leq r < k \cdot s^2$, lies in submesh $r \bmod m^2$. If $r \bmod (2 \cdot m^2) < m^2$, then color p white, else black.

2. In each submesh rearrange the white (black) packets such that those with intermediate destinations in column-bundle l (row-bundle l), $0 \leq l < m$, stand in the columns (rows) $[l \cdot s/m, (l+1) \cdot s/m - 1]$ of the submesh.

3. From column-bundle j, $0 \leq j < m$, route the white packets with intermediate destinations in column-bundle l, $0 \leq l < m$, as a block to the columns $[j \cdot s/m, (j+1) \cdot s/m - 1]$ of column-bundle l. Route the black packets analogously.

4. In each submesh rearrange the white (black) packets such that those with intermediate destinations in row-bundle l (column-bundle l), $0 \leq l < m$, stand in the rows (columns) $[l \cdot s/m, (l+1) \cdot s/m - 1]$ of the submesh.

5. From row-bundle i, $0 \leq i < m$, route the white packets with intermediate destinations in row-bundle l, $0 \leq l < m$, as a block to the rows $[i \cdot s/m, (i+1) \cdot s/m - 1]$ of row-bundle l. Route the black packets analogously.

6. In each submesh, sort the packets. The preliminary destination of a packet p with rank r, $0 \leq r < k \cdot s^2$, lies in section S_l, with $l = \lfloor r \cdot m^2/(s \cdot k) \rfloor$. If $\lfloor (r \cdot m^2 - m^4)/(s \cdot k) \rfloor = l - 1$, then create a copy p' of p with preliminary destination in S_{l-1}. If $\lfloor (r \cdot m^2 + m^4)/(s \cdot k) \rfloor = l + 1$, then create a copy p' of p with preliminary destination in S_{l+1}. If r is even, then color p (and p') white, else black.

7. Like Step 2 for the preliminary destinations.

8. Like Step 3 for the preliminary destinations.

9. Like Step 4 for the preliminary destinations.

10. Like Step 5 for the preliminary destinations.

11. Route the packets within the submeshes to the sections of their preliminary destinations.

12. In each section, sort the packets.

13. In each section S_l, $0 \leq l \leq m \cdot n - 1$, throw away the m^4 packets with the smallest keys (except for S_0), and the m^4 packets with the largest keys (except for $S_{m \cdot n - 1}$). Redistribute the remaining $k \cdot s$ packets within S_l.

If packets have the same key, then special care should be taken not to throw away both copies of a packet, while keeping both packets of another packet. Most practical is to take the index of the PU where a packet started as an additional comparison criterion, to assure that all packets have different keys. The algorithm can be made uni-axial by leaving out the coloring, and applying only uni-axial local operations.

Theorem 6 *Let $s = n^{5/6}/k^{1/6}$.* KKSORT2 *performs bi-axial k-k sorting with respect to a piecewise-continuous indexing with parameter s in* $\max\{4 \cdot n, k \cdot n/2\} + \mathcal{O}(k \cdot s)$ *steps. The queue size is* $k+2$. *Uni-axial k-k sorting takes* $\max\{4 \cdot n, k \cdot n\} + \mathcal{O}(k \cdot s)$ *steps, with queue size* $k + 1$.

Proof: We analyze the presented bi-axial algorithm, KKSORT2. Its uni-axial version can be analyzed analogously. For the case that the sections are not entirely contained within the submeshes, the algorithm should be modified slightly (packets with preliminary destination in some section, must be sent to the (at most four) submeshes that intersect their sections, in proportion to the length of the intersection).

The following facts imply the correctness of KKSORT2. In Step 6, the estimate of the global rank of a packet p with rank r within its submesh, $r \cdot m^2$, is accurate up to m^4. Hence, the index of the destination PU of p is accurate up to m^4/k. Thus after Step 11, a (copy) of a packet resides in its destination section. After Step 11 there are m^4 packets in S_l, $0 < l \leq m \cdot n - 1$, that belong in S_{l-1}, because from every of the m^2 submeshes precisely m^2 copies of packets with estimated destination in S_{l-1} are sent to S_l. Likewise there are m^4 packets in S_l, $0 \leq l < m \cdot n - 1$, that belong in S_{l+1}.

For the time analysis, only the four main steps, Steps 3, 5, 8 and 10, are of importance. The other steps can be performed in $\mathcal{O}(k \cdot s) = \mathcal{O}(k^{5/6} \cdot n^{5/6})$ steps. Step 3 and Step 5 are very regular. It is easy to check that no connection has to transfer more than $k \cdot n/8$ packets, and that packets travel less than n steps. At the beginning of Step 8, there are in every submesh exactly m^3 packets and $2 \cdot m^2$ copies of packets with destination in any section S_l, $0 < l < m \cdot n - 1$ (m^2 copies for $l = 0$ or $l = m \cdot n - 1$). Because the sections are fully contained in the submeshes, this implies that every submesh holds $m^3 \cdot n$ packets and $2 \cdot m^2 \cdot n$ copies of packets with destination in any column-bundle. This means that Step 7 can be performed such that the PUs in the columns $[l \cdot s/m, (l + 2 \cdot k/m) \cdot s/m - 1]$ all hold $k+1$ packets and the PUs in all other columns exactly k packets. Clearly Step 8 now takes $(1 + 2/m) \cdot k \cdot n/8 = k \cdot n/8 + s/4$. Performing Step 9 appropriately, the same bound can be shown for Step 10.

A PU never holds more than $k/2$ packets and one copy of both colors, and thus $Q \leq k+2$. □

5 1-1 Sorting for Large n

We start with a uni-axial algorithm for 1-1 sorting in row-major order. It runs in $2^1\!/\!_2 \cdot n + o(n)$ steps. Asymptotically this is much faster then the algorithms of Section 3. This algorithm is obtained by combining our new insight in merge sorting and the desnakification technique, with old knowledge about sorting with splitters. In Section 5.2 it is turned into a near-optimal bi-axial algorithm. Without loss of generality, we assume that all packets have different keys.

5.1 Uni-Axial Sorting

The mesh is divided in $s \times s$ submeshes. In the algorithm of this section $s = n^{5/6}$, and $m = n/s = n^{1/6}$. We distinguish packets and **splitters**. The splitters are copies of a small subset of the packets. They are broadcast and the packets estimate their ranks by comparison with the splitters. This widely known idea (going back on work of Reischuck [10], and Reif and Valiant [11]) has been used for randomized [3, 2] and deterministic [5] sorting on meshes. In the k-k sorting algorithm of Section 4 we do not need splitters because the packets are fully distributed over the mesh, and thus reliable estimates of the ranks of the packets can be obtained by local comparison among the packets themselves. In the case of 1-1 sorting this does not lead to efficient algorithms. The splitters allow us to spread the necessary information rapidly, while the packets are involved in more useful operations.

First we give the algorithm for selecting and routing the splitters:

Algorithm SPLITTER-ROUTE

1. In every submesh, sort the packets. Copy the packets with ranks $i \cdot m^2$, $0 \leq i \leq s^2/m^2 - 1$: the splitters.

2. In every submesh $B_{i,j}$, $0 \leq i, j < m$, rearrange the splitters such that they stand in the positions (i', j') of $B_{i,j}$, with $i \cdot s/m \leq i' < (i+1) \cdot s/m$, and $j \cdot s/m \leq j' < (j+1) \cdot s/m$.

3. Send the splitters along the rows. A splitter starting in position (i', j') of $B_{i,j}$ drops copies in the positions (i', j') of $B_{i,l}$, for all $0 \leq l < m$.

4. Send the splitters along the columns. A splitter starting in position (i', j') of $B_{i,j}$ drops copies in the positions (i', j') of $B_{l,j}$, for all $0 \leq l < m$.

Lemma 12 SPLITTER-ROUTE *takes* $2 \cdot n + \mathcal{O}(s)$ *steps to complete. No connection has to transfer more than* $\mathcal{O}(s)$ *packets. Finally, each PU holds precisely one splitter, and all splitters are available in every* $s \times s$ *submesh.*

Proof: Step 1 and Step 2 take $\mathcal{O}(s)$ steps, Step 3 and Step 4 take less than n steps. The rearrangement is such that the splitters in $B_{i,j}$ stand in 'subsubmesh' (i, j). After the broadcast these splitters occupy the subsubmeshes (i, j) in all submeshes: the splitters from different submeshes perfectly fit next to each other. This arrangement also assures that during Step 3 and Step 4 a connection has to transfer at most $m/2 \cdot s/m = s/2$ splitters. □

When splitters and packets want to use the same connection, priority is given to the splitters. By the lemma this delays the packets by at most $\mathcal{O}(s)$.

For the packets we perform a kind of m-way merge algorithm. We give the first part of the algorithm.

Algorithm 11SORT

1. In every submesh, sort the packets in row-major order.

2. In every submesh $B_{i,j}$, $0 \leq i,j < m$, shift the packets in row l, $0 \leq l < s$ to row l of $B_{i,(j+l) \bmod (m/2)}$, and copies to row l of $B_{i,(j+l) \bmod (m/2) + m/2}$.

3. In all columns, sort the packets downwards.

After Step 3, there are in a 0-1 distribution at most m^2 dirty rows. For a general distribution this means that a packet resides at most $m^2 - 1$ rows away from its destination row. These three steps take $2 \cdot n + \mathcal{O}(s)$ steps, just as SPLITTER-ROUTE. So, we may assume that after Step 3 the splitters are available in the submeshes. The final steps of 11SORT resemble the final steps of KKSORT2 for $k = 1$:

4. In every submesh, determine for every packet its 'rank', the number r, $0 \leq r \leq s^2$ of splitters that are smaller. The preliminary destination of a packet p with rank r, lies in section S_l, with $l = \lfloor r \cdot m^2/s \rfloor$. If $\lfloor (r \cdot m^2 - m^4)/s \rfloor = l - 1$, then create a copy p' of p with preliminary destination in S_{l-1}. If $\lfloor (r \cdot m^2 + m^4)/s \rfloor = l + 1$, then create a copy p' of p with preliminary destination in S_{l+1}. Discard the splitters, and the (copies of) packets that have preliminary destination in the other half of the mesh.

5. In every submesh, sort the packets in column-major order on their preliminary destination column-bundles.

6. In every row, route the packets to the first PUs in their preliminary destination column-bundles that hold less than two packets.

7. In each submesh, sort the packets in row-major order on their preliminary destination section.

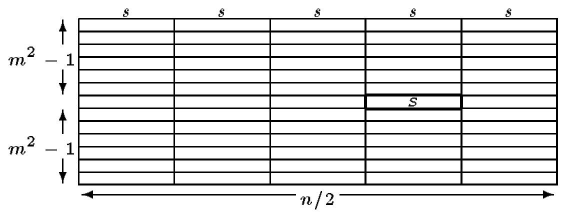
8. In every column, route the packets to the sections of their preliminary destinations.

9. In every section, sort the packets.

10. In every section S_l, $0 < l \leq m \cdot n - 1$, throw away the m^4 packets with the smallest keys, and in each S_l, $0 \leq l < m \cdot n - 1$, throw away the m^4 packets with the largest keys. Redistribute the remaining $k \cdot s$ packets within S_l.

As the algorithm is given, it is not entirely correct. It is *not* true that, as in KKSORT2, exactly m^4 packets must be thrown away on both sides of every section: SPLITTER-ROUTE orders the packets, but the sections do not necessarily hold exactly s packets. Fortunately, the numbers of packets that must be thrown away in a section on the low and high side, respectively, can be determined in an elegant way.

We give a detailed description. Consider some section S and the sections from which it may receive packets after Step 3:



$(2 \cdot m^2 - 1) \cdot n$ packets are stored in these sections, among which the s packets with destination in S. After Step 8, these s packets all reside in S, but also some packets that do not belong in S. How can we figure out which packets to keep, and which packets to throw away? Suppose that S is the l-th section, $(m^2 - 1) \cdot n/s \leq l < m^2 \cdot n/s$, in the involved (whole) rows. Then finally S should hold the packets with ranks r, $l \cdot s \leq r < (l+1) \cdot s$ from among the $(2 \cdot m^2 - 1) \cdot n$ packets. Analogously to the merge algorithms of Section 3, we could copy *all* packets to S, sort them, and throw away the smallest $l \cdot s$ packets and the largest $(2 \cdot m^2 - 1) \cdot n - (l+1) \cdot s$ packets. This gives a correct but very inefficient algorithm. However, it is not necessary to copy all packets to S. It is sufficient if for each contributing section i the counters, the numbers $under_{S,i}$ and $over_{S,i}$ of packets that are *not* sent to S because they are definitely too small or definitely too large, respectively, are known in S. The counters can easily be determined in Step 4. They can be transferred to S during the subsequent steps, in parallel with the packets. As every section sends and receives only $\mathcal{O}(m^3)$ counters in total, they can be routed without causing substantial delay. The numbers $Under_S = \sum_i under_{S,i}$ and $Over_S = \sum_i over_{S,i}$ can be computed in Step 9. Finally, in Step 10, the smallest $l \cdot s - Under_S$ packets and the largest $(2 \cdot m^2 - 1) \cdot n - (l+1) \cdot s - Over_S$ packets in S are thrown away, leaving exactly the s packets belonging in S. Now we get

Theorem 7 *Uni-axial* 1-1 *sorting in row-major order can be performed in* $2^1\!/\!_2 \cdot n + \mathcal{O}(n^{5/6})$ *steps. The queue size is five.*

Proof: Let $s = n^{5/6}$. For the routing time and correctness, we only have to prove that Step 6 can be performed in $n/2 + \mathcal{O}(s)$ steps. All other steps can be performed in $\mathcal{O}(s)$ steps.

The estimate of the rank of a packet, $r \cdot m^2$, is accurate up to m^4. This means that for some section S_l, only a packet (or its copy) with actual destination in some PU P_k, with $k \in [l \cdot s - 2 \cdot m^4, (l+1) \cdot s + 2 \cdot m^4]$, may get preliminary destination in S_l. Hence, at most $s^2 + 4 \cdot m^4 \cdot s$ packets have preliminary destination in any submesh $B_{i,j}$. By the sorting in Step 5, they are distributed almost optimally over the rows of row-bundle i: at most $s + \mathcal{O}(m^4)$ packets stand in any row. The $m^2 \cdot s$ packets with destination in $B_{i-1,j}$ and $B_{i+1,j}$, that may stand in row-bundle i have no serious influence. This shows that Step 6 can be performed as specified: no PU in $B_{i,j}$ has to receive more than two packets.

We consider the routing time of Step 6. For a rightwards moving packet p, residing in some PU $P_{i,j}$ and moving to column l, with $j, l < n/2$, we are interested in the number h_l of packets within row i that have to go to some column k, with $k \geq l$. By the above analysis, we know that $h_l \leq n/2 - l + \mathcal{O}((n/2 - l)/s \cdot m^4) \leq n/2 - l + \mathcal{O}(s)$. p is delayed at most h_l times, and hence p finishes Step 6 within $n/2 + \mathcal{O}(s)$ steps.

A PU may hold up to four (copies of) packets during Step 4 and Step 5. In addition Step 4 can be organized such that a PU holds at most one splitter or counter. Hence, $Q \leq 5$. □

The algorithm is not suited for sorting with respect to any piecewise-continuous indexing with parameter $s \geq n^{5/6}$: it is essential that after Step 3 the packets do not have to make another long vertical move. However, the algorithm *is* correct for any piecewise indexing in which the pieces are scrambled within the rows.

5.2 Bi-Axial Sorting

Essentially 11SORT consists of three main routing phases: horizontal, vertical and horizontal (Step 2, Step 3 and Step 6). These phases take n, n and $n/2$ steps, respectively. The connections between the left and right half are not used anymore after step $n/2$. Thus it may happen that a packet p_1 that stands in column 0 after Phase 1 is routed to a preliminary destination in column $n/2 - 1$ in Phase 3. This is unnecessary: a copy of p_1 stands in column $n/2$. In a uni-axial algorithm this observation does not lead to a faster algorithm: there may be a packet p_2, after Phase 1 in column $n/2 - 1$ and with preliminary destination in column 0, which has to travel $n/2$ steps in Phase 3. On the other hand, in a bi-axial algorithm, it is possible to coalesce the phases. Then p_2 can start Phase 3 after $3/2 \cdot n + \mathcal{O}(s)$ steps, and will reach its preliminary destination after $2 \cdot n + \mathcal{O}(s)$ steps.

We work out the ideas. Only Step 4 is changed: instead of discarding the packets that have their destinations in the other half, we now perform

> In all columns j, $0 \leq j < n/2$, discard the (copies of) packets that have preliminary destination in some column j', with $j' \geq 2 \cdot j$. For $n/2 \leq j < n$, discard the packets with $j' < 2 \cdot j - n$.

Notice that by this rule again exactly one of the copies of a packet reaches every possible destination section.

The steps are coalesced. Most importantly, this means that Step 3 begins in column j after $n/2 + |n/2 - j|$ steps, and Step 6 after $3/2 \cdot n + |n/2 - j|$ steps.

Theorem 8 *Bi-axial* 1-1 *sorting in row-major order can be performed in* $2 \cdot n + \mathcal{O}(n^{5/6})$ *steps. The queue size is five.*

Proof: A packet that starts Step 6 after $2 \cdot n - d + \mathcal{O}(s)$ steps, has to travel at most d steps to reach the column-bundle of its preliminary destination. We check this for a packet p that is routed in Step 2 to some column j, with $j < n/2$. p starts Step 6 after $2 \cdot n - j + \mathcal{O}(s)$ steps. In Step 4 the preliminary destination of p is determined. p survives only when it goes to some column l, with $l < 2 \cdot j$: p has to travel at most j steps. By a refinement of the analysis in the proof of Theorem 7, it can be shown that p is not delayed more than $2 \cdot j - l$ times. Hence, Step 7 can start in all submeshes after $2 \cdot n + \mathcal{O}(s)$ steps. □

In fact this algorithm is still *locally* uni-axial: every PU uses only horizontal or vertical connections.

5.3 2-2 Sorting

For efficient 2-2 sorting, we modify the uni-axial version of 11SORT. We still select s^2/m^2 splitters in every submesh.

For uni-axial 2-2 sorting, the sorting in Step 1 is performed in (semi-) layered row-mayor order. Step 2 is replaced by

2. In every submesh $B_{i,j}$, $0 \leq i, j < m$, shift the packets in row l, $0 \leq l < s$ to row l of $B_{i,(j+l) \bmod m}$.

The estimate of the rank of a packet is accurate only up to $2 \cdot m^4$, but the accuracy of the estimate of the preliminary destination of a packet is unchanged. In Step 4, no (copies of) packets are thrown away. In Step 6, the PUs receive up to three packets.

Theorem 9 *Uni-axial* 2-2 *sorting in row-major order can be performed in* $3 \cdot n + \mathcal{O}(n^{5/6})$ *steps. The queue size is five.*

Proof: Step 2 and Step 3 are 2-2 routing and sorting operations on a linear array, and can be performed in n steps each. Step 6 now takes $n + \mathcal{O}(s)$ steps. This can be proven with Lemma 1, by observing that at most $\mathcal{O}(m^4)$ copies go to any section: the distribution of source/destination pairs within a row is close to forming a 2-2 routing problem. □

For bi-axial 2-2 sorting, we essentially apply two orthogonal versions of 11SORT. For the necessary coloring, the packets should first be sorted within the submeshes. Then the packets with even rank are colored white, the others black. The white packets perform Step 1 through 8 of 11SORT, the black packets perform orthogonal steps. Then all packets perform Step 9 and 10 together. We need only one set of splitters.

Theorem 10 *Bi-axial* 2-2 *sorting in row-major order can be performed in* $2^1\!/\!_2 \cdot n + \mathcal{O}(n^{5/6})$ *steps. The queue size is nine.*

Acknowledgement

The questions of Uli Meyer, concerning the precise implementation of the algorithm of [1], urged me to investigate the subject of row-major order sorting for *all* n. All other results were born out of this research. Michael Kaufmann pointed out some improvements in the analysis and the presentation. Torsten Suel noticed that the original algorithms of Section 5 did not result in an even distribution of the packets.

6 Conclusion

We presented novel uni-axial and bi-axial row-major algorithms for sorting on two-dimensional meshes. They are considerably faster than existing algorithms. A tremendous improvement is our near-optimal algorithm for 1-1 sorting: it is much simpler than the earlier algorithm, it is suited for more useful indexings, it is locally uni-axial, and it has queue size five.

Future research could address (1) the optimality of the uni-axial sorting algorithm with run time $2^1\!/\!_2 \cdot n + o(n)$ steps; (2) a further improvement of the merge sort algorithm, in order to obtain even faster sorting for all n.

References

[1] Chlebus, B.S., M. Kaufmann, J.F. Sibeyn, 'Deterministic Permutation Routing on Meshes,' *Proc. 5th Symposium on Parallel and Distributed*, IEEE, pp. 814-821, 1993.

[2] Kaklamanis, C., D. Krizanc, 'Optimal Sorting on Mesh-Connected Processor Arrays,' *Proc. 4th Symposium on Parallel Algorithms and Architectures*, pp. 50-59, ACM, 1992.

[3] Kaufmann, M., S. Rajasekaran, J.F. Sibeyn, 'Matching the Bisection Bound for Routing and Sorting on the Mesh,' *Proc. 4th Symposium on Parallel Algorithms and Architectures*, pp. 31-40, ACM, 1992.

[4] Kaufmann, M., J.F. Sibeyn, 'Randomized k-k Sorting on Meshes and Tori,' manuscript, 1992.

[5] Kaufmann, M., J.F. Sibeyn, T. Suel, 'Derandomizing Algorithms for Routing and Sorting on Meshes,' *Proc. 5th Symposium on Discrete Algorithms*, ACM-SIAM, 1994, to appear.

[6] Krizanc, D., L. Narayanan, 'Zero-One Sorting on the Mesh,' *Proc.A5th Symposium on Parallel and Distributed Processing*, IEEE, pp. 641-647, 1993.

[7] Kunde, M., 'Block Gossiping on Grids and Tori: Deterministic Sorting and Routing Match the Bisection Bound,' *Proc. European Symposium on Algorithms*, LNCS 726, pp. 272-283, Springer-Verlag, 1993.

[8] Leighton, T., F. Makedon, Y. Tollis, 'A $2n - 2$ Step Algorithm for Routing in an $n \times n$ Array with Constant Size Queues,' *Proc. Symposium on Parallel Algorithms and Architectures*, pp. 328-335, ACM, 1989.

[9] Leighton, T., *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan-Kaufmann Publishers, San Mateo, California, 1992.

[10] Reischuk, R., 'Probabilistic Parallel Algorithms for Sorting and Selection,' *SIAM Journal of Computing*, 14, pp. 396-411, 1985.

[11] Reif, J., L.G. Valiant, 'A logarithmic time sort for linear size networks,' *Journal of the ACM*, 34, pp. 68-76, 1987.

[12] Schnorr, C.P., A. Shamir, 'An Optimal Sorting Algorithm for Mesh Connected Computers,' *Proc. 18th Symposium on Theory of Computing*, pp. 255-263, ACM, 1986.

[13] Thompson, C.D., H.T. Kung, 'Sorting on a Mesh-Connected Parallel Computer,' *Communications of the ACM*, 20, pp. 263-271, 1977.

[14] Sibeyn, J.F., M. Kaufmann, 'Deterministic 1-k Routing on Meshes, with Applications to Worm-Hole Routing,' *Proc. 11th Symposium on Theoretical Aspects of Computer Science*, Springer Verlag, 1994.

[15] Sibeyn, J.F., B.S. Chlebus, M. Kaufmann, 'Permutation Routing on Meshes with Small Queues,' submitted to *MFCS 94*, 1993.