# Optimal Deterministic Sorting and Routing on Grids and Tori with Diagonals[1]

Manfred Kunde
Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau,
Postfach 0565, D-98684 Ilmenau, Fed. Rep. of Germany
Manfred.Kunde@theoinf.tu-ilmenau.de

Rolf Niedermeier
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
niedermr@informatik.uni-tuebingen.de

Klaus Reinhardt
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
reinhard@informatik.uni-tuebingen.de

Peter Rossmanith
Fakultät für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Fed. Rep. of Germany
rossmani@informatik.tu-muenchen.de

[1]Research partially supported by DFG-SFB 0342 TP A4 "KLARA" and DFG Project La 618/3-1 "KOMET." This paper continues "Faster Sorting and Routing on Grids with Diagonals" presented at the *11th Symposium on Theoretical Aspects of Computer Science* held in Caen, France, February 24–26, 1994.

**Abstract**

We present deterministic sorting and routing algorithms for grids and tori with additional diagonal connections. For large loads ($h \geq 12$), where each processor has at most $h$ data packets in the beginning and in the end, the sorting problem can be solved in optimal $hn/6 + o(n)$ and $hn/12 + o(n)$ steps for grids and tori with diagonals, respectively. For smaller loads, we present a new concentration technique that yields very fast algorithms for $h < 12$. For a load of 1, the previously most studied case, sorting only takes $1.2n + o(n)$ steps and routing only $1.1n + o(n)$ steps. For tori, we can present optimal algorithms for all loads $h \geq 1$. The above algorithms all use a constant size memory for all processors and never copy or split packets, a property that the corresponding lower bounds make use of.

If packets may be copied, 1–1 sorting can be done in only $2n/3 + o(n)$ on a torus with diagonals.

Generally gaining a speedup of 3 by only doubling the number of communication links compared to a grid without diagonals, our work suggests building grids and tori *with* diagonals.

**Keywords:** parallel architectures, mesh connected processor arrays, diagonal connections, parallel algorithms, sorting, routing

# 1 Introduction

Mesh-connected processor arrays have been in the focus of research on parallel computation for many years. Among others, one of the reasons for their popularity lies in their scalability, an important property that many other architectures lack [1, 2, 24, 30]. Routing and sorting are important algorithmic problems studied for mesh architectures because they are the building blocks for many algorithms. For conventional grids of processors with four-neighborhood, there was a strong research focus until optimal results for deterministic sorting and routing were finally obtained [5, 10]. In this paper, we study grids with eight-neighborhood, that is, grids with diagonals, presenting optimal results for sorting and routing.

The standard grid architecture with its four-neighborhood has been extended in several ways. Thus it is quite popular to study higher-dimensional grids or grids with additional wrap-around connections, so-called tori. Another possibility to enlarge the neighborhood of grid processors is to equip them with additional diagonal connections. In spite of the fact that meshes with diagonals are well-known and have been used for some applications like matrix multiplication and LU decomposition [14, 27], very little is known about how to exploit the additional communication links for faster sorting and routing. Equipping grids with additional wrap-around connections often leads to sorting and routing algorithms which are twice as fast as without wrap-arounds [5, 10]. Equipping grids with diagonals first means doubling the number of data channels. We show that with diagonal connections there exist sorting and routing algorithms that are *more* than twice as fast as algorithms for grids without diagonals. Our algorithms for grids and tori with diagonals are deterministic and optimal—they match the bisection bound asymptotically. Duplication of packets is not used and the lower bounds make use of this property.

In the following, we deal with $h$–$h$ problems where each processor initially and

Table 1: Comparison of selected results for grids with and without diagonals. We omit sublinear terms. The algorithm of Kaklamanis and Krizanc is randomized. All other algorithms are deterministic. Except for $h = 1$ the results for sorting and routing are the same.

| Problem | New results with diagonals | Without diagonals | |
|---|---|---|---|
| 1–1 routing | $1.1n$ | $2n$ | Leighton et al. [18] |
| 1–1 sorting | $1.2n$ | $2n$ | Kaklamanis and Krizanc [3], Kaufmann et al. [5] |
| 4–4 sorting | $1.6n$ | $4n$ | Kunde [9] |
| 8–8 sorting | $1.86n$ | $4n$ | Kunde [10], Kaufmann et al. [5] |
| 12–12 sorting | $2n$ | $6n$ | Kunde [10], Kaufmann et al. [5] |

finally contains $h$ packets. We regard the load $h = O(1)$ as a small constant—we don't consider the case when $h$ is a function of the grid size. For two-dimensional $n \times n$ meshes without diagonals 1–1 problems have been studied for more than twenty years. Several 1–1 sorting algorithms exist for buffer size 1, i.e., each processor can store only one packet at each time. The fastest algorithms need $3n+o(n)$ steps [19, 23, 28]. For buffer size 2, the 1–1 sorting problem can be solved deterministically in $2.5n + o(n)$ transport steps [9]. Kaklamanis and Krizanc [3] presented a randomized algorithm (with constant buffer size) that sorts in only $2n + o(n)$ steps with high probability. Using derandomization techniques, this algorithm can even be made deterministic [5]. For 1–1 routing, Leighton, Makedon, and Tollis [18] presented an optimal deterministic algorithm (with constant buffer size) that exactly matches the distance bound of $2n-2$ steps. Rajasekaran and Overholt [22] further reduced the buffer size. We present algorithms for grids with diagonals that need $1.2n + o(n)$ steps for 1–1 sorting and $1.1n + o(n)$ steps for 1–1 routing. For grids with diagonals, we summarize some of the new results in Table 1 and compare them with the so far known best results on grids without diagonals. In the table we omit all sublinear terms because they are of no importance for the asymptotic complexity.

Rajasekaran [21] and also Kaufmann and Sibeyn [6] invented randomized

Table 2: Comparison of selected results for tori with and without diagonals. We omit sublinear terms. All algorithms are deterministic. Except for $h = 1$ the results for sorting and routing are the same. For 1–1 sorting where replication of data is allowed, we provide a more efficient algorithm than for our standard model that disallows copying of elements.

| Problem | New results with diagonals | Without diagonals | |
|---|---|---|---|
| 1–1 routing | $0.66n$ | $n$ | Kaufmann et al. [5] |
| 1–1 sorting with copying | $0.67n$ | $\geq n$ | distance bound |
| 1–1 sorting | $n$ | $1.25n$ | Kaufmann et al. [5] |
| 4–4 sorting | $n$ | $2n$ | Kunde [9] |
| 8–8 sorting | $n$ | $2n$ | Kunde [10], Kaufmann et al. [5] |
| 12–12 sorting | $n$ | $3n$ | Kunde [10], Kaufmann et al. [5] |

algorithms for $h$–$h$ problems on an $n \times n$ mesh that need $hn/2 + o(n)$ steps if $h \geq 8$. It is possible to solve the $h$–$h$ routing and sorting problems within the same number of steps deterministically [5, 10]. These results are optimal, since they match the simple bisection bound of $hn/2$ steps valid for this type of architecture. On meshes with diagonals, we reach $hn/6 + O(n^{2/3})$ steps for deterministic $h$–$h$ sorting and routing, provided that $h \geq 12$. This gives an acceleration factor of 3 and also matches the bisection bound.

For wrap-around meshes (or tori) without diagonals, Kaufmann and Sibeyn [6] presented a randomized $h$–$h$ sorting algorithm with $hn/4 + o(n)$ steps for $h \geq 8$. There exist equally fast deterministic algorithms [5, 10]. Both algorithms match asymptotically the respective bisection bound of $hn/4$. If we add diagonals to tori, we can sort and route in only $hn/12 + O(n^{2/3})$ steps if $h \geq 12$. This means we again get a speedup of 3. Though the diameter of a torus with diagonals is $n/2$ and the bisection bound is $hn/12$, our best algorithm for the $h$–$h$ problem with $h \leq 12$ still needs $n + o(n)$ steps. However the algorithm still remains optimal for $h < 12$ since it asymptotically matches a lower bound of Krizanc and Narayanan. They showed that even the 1–1 sorting problem takes at least $n - o(n)$ steps on a torus with diagonals if data packets cannot be copied and the buffer size is 9 [7]. Thus we have optimal $h$–$h$ sorting algorithms for tori for all $h$. Recently, Sibeyn independently discovered an optimal sorting algorithm for tori with diagonals for large $h$ [26]. We also show that the requirement of no data replication is necessary for the lower bound of Krizanc and Narayanan: We show that copying of packets enables sorting in $2n/3 + o(n)$ time while still using only a buffer of size 9. We summarize some of the new results in Table 2 and compare them with the known results for tori without diagonals. Again, we omit all sublinear terms.

The results of this paper demonstrate that grids with diagonals are a promising architecture because the gain of reduced running times is obviously bigger than the extra costs of additional links. Note that doubling row and column links would have the same cost as additional diagonal links, but the bisection bound for such an architecture is $hn/4$. By way of contrast, we beat this bound and obtain $hn/6$ algorithms with diagonal links. Our work and its predecessor [12] have inspired related work [7, 26].

We use a sorting method that is mainly based on all-to-all mappings [10]. This method was the breakthrough to deterministic algorithms that match the bisection bound. Roughly speaking, this scheme consists of two kinds of operations: local sorting in blocks of processors (cheap) and global communication in a regular communication pattern (expensive). The sorting algorithm performs the global communication, called all-to-all mapping, twice. In the third section, we present this method in more detail and show that the central task in obtaining an efficient algorithm for sorting is to devise an efficient algorithm for all-to-all mapping.

In Section 4, we present an optimal algorithm for all-to-all mapping for tori with diagonals. Compared to grids without wrap-around connections, the advantage of tori is that there are no border processors, so the situation is identical for *all* processors. Having obtained an optimal algorithm for all-to-all mapping for tori in Section 5, we proceed with an embedding of tori into grids, culminating in the presentation of an optimal algorithm for all-to-all mapping for grids without wrap-arounds. In particular, this implies one of our main results, namely that $h$–$h$ sorting with $h \geq 6$ can be done in asymptotically $hn/6$ steps. Finally, in Section 6 we apply this result to obtain fast algorithms for values $h < 12$ (see Tables 1 and 2 for a small selection), thereby using concentration techniques, most notably, concentrating all-to-all mappings.

## 2 Preliminaries

In this section, we present basic definitions and notations.

A *processor grid with diagonals* is a network of $n^2$ processors arranged in an $n \times n$ array. Processor $(r, c)$ in row $r$ and column $c$ on the grid is directly connected by a bi-directional communication link to processor $(r', c')$ if $\max\{|r - r'|, |c - c'|\} = 1$. We speak of a diagonal connection if $|r - r'| = |c - c'| = 1$. A *torus* is a grid with wrap-around connections. Since tori have no borders, each processor is the center of an eight-neighborhood.

For a full *$h$–$h$ routing problem*, each processor contains exactly $h$ packets initially. Each packet has a destination address, and each processor is destination of exactly $h$ packets. The routing problem is to transport each packet to its destination address. For the more general sorting problem, the destination of each packet is not fixed, but determined by its rank according to some linear order. We assume that each packet in a processor $P$ lies in a (memory) place $(P, j)$, where $0 \leq j < h$. For a given $j$ the set of places $\{ (P, j) \mid P$ is a processor $\}$ is called the $j$th layer. There are exactly $h$ disjoint layers, numbered from 0 to $h - 1$. The places are indexed by an index function $g$ that is a one-to-one mapping from the places onto $\{0, \ldots, hn^2 - 1\}$. Then the sorting problem with respect to $g$ is to transport the $i$th smallest element to the place indexed with $i - 1$.

For a full $h$–$h$ routing problem, one can supply each packet with an index of its destination processor. In this manner, the full $h$–$h$ routing problem becomes an $h$–$h$ sorting problem.

A tool of central importance in what follows is the so-called *all-to-all mapping* [10]. Assume that for some suitable $m$ we partition the $n \times n$-mesh into $m^2$ quadratic $n/m \times n/m$ submeshes, called blocks. Then an all-to-all mapping is the problem to transport for each block $1/m^2$ of its elements to each other block of the mesh.

The model of computation is the conventional one, where only nearest neighbors exchange data [15, 17]. In general, we disallow replication (that is, copying) of packets in our algorithms. A communication link can in one step transport at most one packet in each direction. Processors may store more than $h$ packets, but the number has to be bounded by a constant that is independent of the number of processors. For complexity considerations, we count only communication steps and ignore operations within a processor.

Each processor has eight links. We assume that diagonal links leading out of the grid at its border are connected together. These additional connections along the border are called *outer links* (cf. Figure 2.1).

# 3 Sorting and routing with all-to-all mappings

In this section, we briefly describe how to sort the elements on a grid with the help of an all-to-all mapping that distributes data uniformly all over the mesh. We then discuss how we can use the same method even for partial $h$–$h$ routing problems. You can find a more detailed description in the paper that introduced all-to-all mappings [10]. There is a similarity between sorting with all-to-all mappings and Leighton's Columnsort [16]. The general results have been stated in previous work. The sorting method works on arbitrary networks. The problem is how to implement an efficient and fast all-to-all mapping. Our aim here is to repeat the general results and give an idea as to why they are correct.

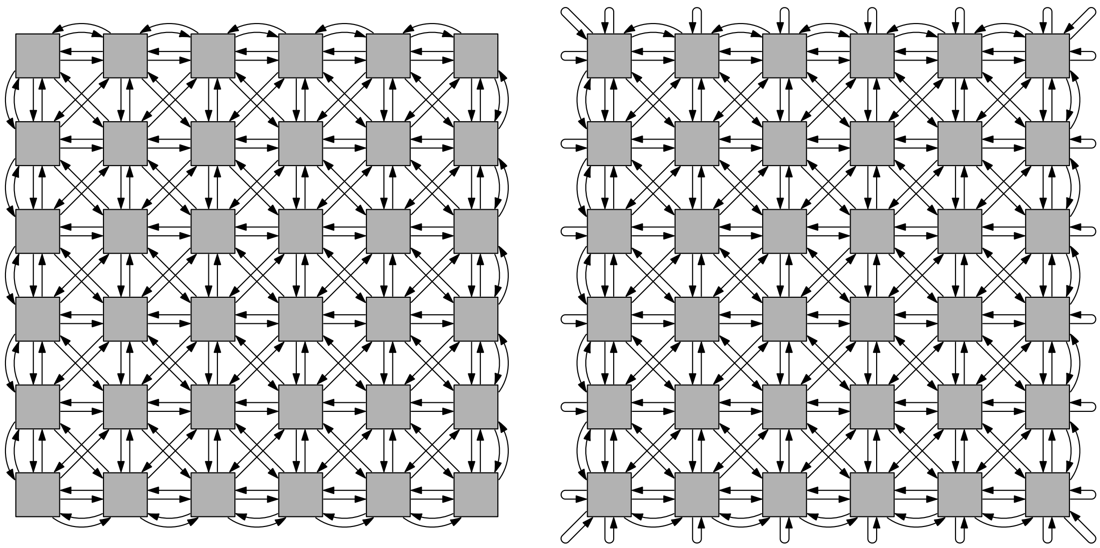

Figure 2.1: Grids with diagonals and outer links. For grids with 8-neighborhood we assume that each processor has 8 bidirectional communication links. At the border we connect neighboring processors with additional links that would not be used otherwise doubling effectively the transport capacity between them (left side). Other wires are not really used by our algorithms, but it yields conceptually simpler algorithms when assuming that the remaining outgoing and incoming links of a processor are connected in loop-back mode (right side).

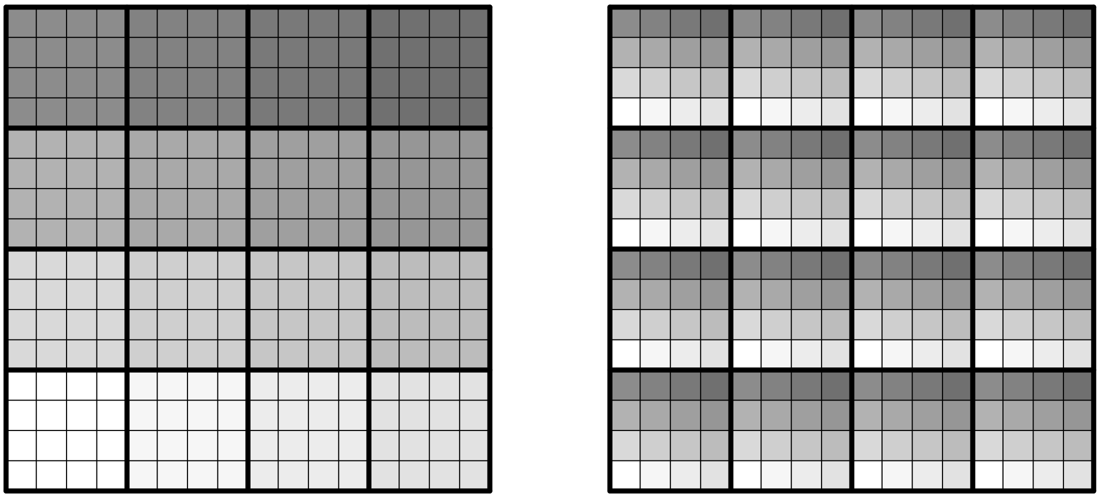

Figure 3.1: The standard all-to-all mapping on a mesh with 16 blocks.

For sorting, we divide the $n \times n$-mesh into $m^2$ quadratic $n/m \times n/m$–submeshes, called blocks. We further divide each block into $m^2$ subblocks and call such a subblock a *brick.* This means each block contains $m^2$ bricks. We number the blocks from 0 to $m^2 - 1$ such that block $i$ and block $i+1$ are neighbors. We must choose the indexing $g$ in such a way that all places in block $i$ have smaller indices than all places in block $i+1$. We call such an indexing *block-wise continuous.* To see the correctness of the following sorting method we use the 0–1 principle [**?**, 17].

First we distribute all data over the mesh in order to get approximately *the same number of ones* into each block. We start by sorting each block individually as follows: The $i$th brick gets elements $i$, $i + m^2$, $i + 2m^2$, and so on. In this way, the number of ones in each brick differs at most by 1. Next, we send from every block exactly one brick to every block on the mesh as illustrated in Figure 3.1. Now each block contains almost the same number of ones (the difference is at most $m^2$). We reach such a global distribution of data by an all-to-all mapping [10]. In a second step, we sort each block in such a way that the first brick contains the smallest elements and the last brick the largest ones. So, at most, one brick contains zeros *and* ones. Let us call it the *dirty brick.* Since each block contains almost the same *number* of ones, the position of the dirty brick is also almost the same in each block: The positions of the dirty bricks differ at most by one, say, the position is either the $k$th or $(k+1)$st brick, provided that a brick contains at least $m^2$ elements.

Now an all-to-all mapping maps the first brick of each block to the first block, the second brick of each block to the second block, and so on. Afterwards, all dirty bricks are in the $k$th or $(k+1)$st block, so the whole mesh is nearly sorted. To finish, we sort all adjacent pairs of blocks.

How long does it take to sort by the above method? We perform two all-to-all mappings and sort three times locally in blocks. For distance reasons the all-to-all mapping takes $\Omega(n)$ steps. The local operations take $O(n/m)$ steps, since a block is $n/m$ processors wide. The smaller the blocks are, the faster the algorithm will be. As mentioned above, the method only works if each brick contains at least $m^2$ elements. There are $m^4$ bricks in the mesh that contains altogether $hn^2$ elements, so each brick contains $hn^2/m^4$ elements. The above condition implies

$$\frac{hn^2}{m^4} \geq m^2.$$

This inequality leaves a lot of freedom. We choose $m = \sqrt[3]{n}$ and assume that $m$ is an integer. In total, the complexity of performing an all-to-all mapping asymptotically governs the time bound of our method. The above sorting algorithm directly applies to full $h$–$h$ routing problems. For partial $h$–$h$ routing problems with a total loading of 75 percent (that is, $0.75hn^2$ packets instead of a full load of $hn^2$ packets), for example, the sorting algorithm would route the packets to wrong destinations. This is caused by the nonexisting $0.25hn^2$ dummy packets for which it is not clear in which place they have to be transported.

This problem can be overcome in the following way: Instead of sorting the blocks in the beginning of the second step, we send packets with block address $j$ to those bricks which are going to the block with index $j$. That is, we use the bricks as a basic transport unit. It may happen that too many packets want to go to their transport bricks. One can show, however, that after the first all-to-all mapping in each block, the number of packets destined for an arbitrary block $j$ is at most $hn^2/m^4 + e_j$, where $\sum_j e_j \leq m^2$. In this case, at most $e_j$ packets must move to bricks destined to either block $j-1$ or to block $j+1$. It can be shown that this routing within the blocks takes only $O(m^2)$ additional steps which is negligible.

# 4 An Optimal Algorithm for All-to-all Mapping on the Torus

Since the efficiency of performing an all-to-all mapping predominates the overall complexity of sorting with all-to-all mappings, the main task in the following is to present an efficient algorithm for all-to-all mapping. In this section, we give an efficient algorithm for tori, where we assume a load of $h \geq 12$.

We assume that the torus consists of $(2k+1) \times (2k+1)$ blocks and each block consists of $(2k+1)^2$ bricks. We have to send one brick from each block to each other block. So, we have to fix a system of movements for each pair of blocks to transport the contents of the brick. There are $(2k+1)^4$ such pairs. To make the algorithm simple, we want to make such a system of movements between two blocks dependent only on their relative positions. In this way, we need to describe only the routes from one fixed block to the other $(2k+1)^2 - 1$ blocks. Additionally:

- One link between two processors can transfer exactly one packet in one step. We assume that it can transfer slightly more, i.e., $1 + 1/(2k+1)$.
- Bricks need not be transported as a whole. Parts of bricks may reach their destination at different times and on different ways (and indeed they will).[1]

Under these assumptions, all routes look as shown in Figure 4.1. The arrows

[1]Note that it is also possible to send all bricks on the same way, splitting them into at most 12 parts. This more practical method is demonstrated at the WWW page `http://www-fs.informatik.uni-tuebingen.de/~reinhard/triang.html`. However, to prove its correctness is more difficult.

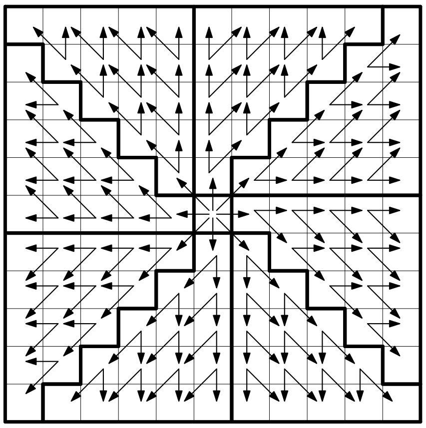

Figure 4.1: Routes from the center block to all other blocks.

show how data flows from the center block to all the other blocks. The algorithm operates in $k$ phases. In each phase, some data is transported one block farther in the direction of the arrows. In the first phase, of course, only the inner eight arrows are used and in each subsequent phase, data reaches farther towards the outer blocks at the borders. In the $k$th phase, data is transferred along *all* arrows.

Data is transferred in such manner that in the end each block will have received a fraction of $1/(2k+1)^2$ of the center block's data. Before going into detail, i.e., describing how much data is transferred over each arrow in each phase—we exploit some more symmetry. Figure 4.1 shows 8 *triangles*. These 8 triangles are quite similar; one can map one onto the other by rotation or shearing or both. The corresponding arrows of different triangles always transport the same amount of data in each step. Therefore it suffices to describe only the data movements in one triangle to describe the whole algorithm.

However we distribute the capacity in one phase of the data movement in one triangle to the two different possible directions in the triangle, the capacity along one direction being used in two triangles will again sum up to be the same. This means as long as we use exactly the capacity $c$ for the sum of all movements in one triangle, this rotation-shearing symmetry, together with the fact that the data distributing movements coming from the other $(2k+1)^2 - 1$ blocks run exactly in parallel, will guarantee that for every connection of blocks the sum of all capacities used by the different systems of movement is again $c$.

Each triangle consists of $k$ rows where the block in the first row is adjacent to the center and the $k$th row is at the border. The data movements are constructed in a way such that in the $j$th phase the $i$th row receives an amount of

$$m_{ij} = \frac{12(k+1)((k+2)i^2 + (k-j)i)}{j(j+1)(j+2)(2k+1)^2}$$

packets from the $i-1$st row for $i \leq j$. For $j = 1$ and $i = 1$ this means $m_{1,1} = 1 + 1/(2k+1)$, which is exactly the amount shifted from the center block into each triangle in the first phase. Hereby we transport data in such a way that each block within a row gets exactly the same amount of data, that is $m_{ij}/i$. This also means that each block in the $i-1$st row for $1 < i < j$ must spend $m_{ij}/(i-1)$. This is accomplished by moving in the $j$th phase $(i-l)m_{ij}/(i(i-1))$ from the $l$th block in the $i-1$st row to the $l$th block in the $i$th row and $lm_{ij}/(i(i-1))$ from the $l$th block in the $i-1$st row to the $l+1$st block in the $i$th row. This means that the $l$th block in the $i$th row receives $(l-1)m_{ij}/(i(i-1))$ from the $l-1$st block in the $i-1$st row and thus receives $(i-l+l-1)m_{ij}/(i(i-1)) = m_{ij}/i$ in total.

Each phase takes only $n/(2k+1)$ steps if the capacity of a link is $1+1/(2k+1)$. Now we show that the above algorithm indeed uses only a link capacity of $1 + 1/(2k+1)$. For each of the $(2k+1)^2$ blocks, there are 8 triangles. For $i \geq j$, the $i$th rows of these $8(2k+1)^2$ triangles receive $m_{ij} \cdot b$ packets during the $j$th phase where $b$ is the number of processors in one block. The total amount of moved packets is therefore $8(2k+1)^2 \cdot \sum_{i=1}^{j} m_{ij} \cdot b$ packets. For symmetry reasons, each link is subject to an equal flow of data. There are $(2k+1)^2 \cdot b$ processors in the grid. Each has 8 links, so the capacity per link is $\sum_{i=1}^{j} m_{ij} = 1 + 1/(2k+1)$:

$$\sum_{i=1}^{j} m_{ij} \quad = \quad 12(k+1)\frac{(k+2)\sum_{i=1}^{j} i^2 + (k-j)\sum_{i=1}^{j} i}{j(j+1)(j+2)(2k+1)^2}$$

$$
\begin{aligned}
&= 12(k+1)\frac{\frac{1}{6}(k+2)j(2j+1)(j+1)+\frac{1}{2}(k-j)j(j+1)}{j(j+1)(j+2)(2k+1)^2} \\
&= 1+\frac{1}{2k+1}.
\end{aligned}
$$

We assumed one link has capacity of $1+1/(2k+1)$ packets instead of one packet at a time. If we return to the normal capacity of 1, then the above algorithm can be performed with a slowdown of $1+1/(2k+1)$, which means it needs $(1+1/(2k+1))\cdot n/2$ instead of $n/2$ steps. We choose $k=n^{1/3}$ which corresponds to a block-size of $n^{4/3}$. This means we can perform an all-to-all mapping in $n/2+O(n^{2/3})$ steps on a torus with diagonals.

It remains to consider the effect of the movements, that is, to show that the packets are distributed in the desired way. By an induction on $j$, we show that the amount of packets in the $i$th row of a triangle after the $j$th phase is

$$
a_{ij}=\frac{12(k+1)}{(2k+1)^2}\cdot\frac{(k+2)i+k-j}{(j+1)(j+2)}
$$

for $i\leq j$. (For $i>j$, obviously $a_{ij}=0$.) The $i$th row receives $m_{ii}=a_{ii}$ in the $i$th step. From now on the amount in the $j$th step changes by

$$
m_{ij}-m_{i+1,j}=\frac{12(k+1)}{(2k+1)^2}\cdot\frac{(k+2)(2i+1)+k-j}{j(j+1)(j+2)},
$$

which means we get $a_{i,j-1}+m_{ij}-m_{i+1,j}$

$$
\begin{aligned}
&= \frac{12(k+1)}{(2k+1)^2}\cdot\left(\frac{(k+2)i+k-j+1}{j(j+1)}-\frac{(k+2)(2i+1)+k-j}{j(j+1)(j+2)}\right) \\
&= \frac{12(k+1)}{(2k+1)^2}\cdot\frac{(k+2)ij+kj-j^2}{j(j+1)(j+2)}=a_{ij}.
\end{aligned}
$$

After the $k$th stage we have a fraction

$$
a_{ik}=\frac{12i}{(2k+1)^2}
$$

of the center block in the $i$th row, which means that we have $12/(2k+1)^2$ in each block, since we transport data in such a way that each block within a row gets exactly the same amount of data. Here we can see that this method works for a load of $h=12$ packets per processor.

This fast algorithm for all-to-all mapping immediately yields a fast sorting algorithm.

**Theorem 4.1** *A torus with diagonals can solve the $h$–$h$ sorting problem for $h \geq 12$ in asymptotically optimal $hn/12 + O(n^{2/3})$ steps.*

*Proof.* Two all-to-all mappings need $hn/12 + O(n^{2/3})$ steps using the above algorithm. Local sorting needs another $O(n^{2/3})$ steps. Every sorting algorithm needs at least $hn/12$ steps, the bisection bound. Thus our result is asymptotically optimal. □

# 5 An Embedding of Tori with Diagonals into Grids with Diagonals

At first sight, due to its wrap-around connections, the torus appears to be more complicated than the grid. By now, however, the torus with diagonals has come nearer to its bisection bound for sorting than the grid [12]. The reason for this is the symmetry of the torus (no center, no borders). This symmetry leads to simple algorithms.

A torus without diagonals can be embedded into a grid with delay 2 [25]. In this section, we show that there is also an embedding for tori with diagonals into grids. Again the delay is 2. The rough idea is to fold the torus two times, bringing together 4 processors each time, and then again unfolding it as described in Figure 5.1. Since the embedding jumbles a sorting algorithm's indexing, we also have to show that embedding all-to-all mappings on a torus onto the grid results in all-to-all mappings. As a consequence, we get an optimal algorithm for all-to-all mapping for the grid with diagonals.

To embed a torus algorithm into a grid, a one-to-one mapping from torus processors to grid processors is first necessary. Secondly, we must show how

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 | 32 |
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |

| | | | |
|---|---|---|---|
| 1 8 | 2 7 | 3 6 | 4 5 |
| 57 64 | 58 63 | 59 62 | 60 61 |
| 9 16 | 10 15 | 11 14 | 12 13 |
| 49 56 | 50 55 | 51 54 | 52 53 |
| 17 24 | 18 23 | 19 22 | 20 21 |
| 41 48 | 42 47 | 43 46 | 44 45 |
| 25 32 | 26 31 | 27 30 | 28 29 |
| 33 40 | 34 39 | 35 38 | 36 37 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 8 | 2 | 7 | 3 | 6 | 4 | 5 |
| 57 | 64 | 58 | 63 | 59 | 62 | 60 | 61 |
| 9 | 16 | 10 | 15 | 11 | 14 | 12 | 13 |
| 49 | 56 | 50 | 55 | 51 | 54 | 52 | 53 |
| 17 | 24 | 18 | 23 | 19 | 22 | 20 | 21 |
| 41 | 48 | 42 | 47 | 43 | 46 | 44 | 45 |
| 25 | 32 | 26 | 31 | 27 | 30 | 28 | 29 |
| 33 | 40 | 34 | 39 | 35 | 38 | 36 | 37 |

Figure 5.1: An embedding of an $8 \times 8$-torus into an $8 \times 8$-grid with the intermediate step of a $4 \times 4$-grid resulting from the "folding process."

moves from one processor to another are translated into moves on the grid. Here a *move* simply means a data transport between neighboring processors.

Subsequently, for ease of presentation, we first concentrate on the one-dimensional case, that is, embedding a ring of processors into a linear array. Assume that the processors of the ring and of the array are consecutively numbered from 0 to $n-1$. Then the mapping of the ring processors to the array processors is given by the bijective function $s : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$,

$$s(i) = \begin{cases} 2i & \text{if } 0 \le i < n/2 \\ 2(n-i-1)+1 & \text{if } n/2 \le i < n. \end{cases}$$

The following lemma shows that $s$ maps neighboring ring processors to array processors that have at most distance 2 from each other. Therefore delay 2 is the best we can hope for.

**Lemma 5.1** *$|s(i) - s(i+1)| \le 2$ for $0 \le i < n-1$ and $|s(n-1) - s(0)| = |s(n/2-1) - s(n/2)| = 1$.*

*Proof.* First consider $0 \le i < n/2-1$, then $|s(i) - s(i+1)| = |2i - 2(i+1)| = 2$. Second consider $n/2 \le i < n-1$, then $|s(i) - s(i+1)| = |2(n-i-1)+1-2(n-(i+1)-1)+1| = 2$. Finally we have $|s(n/2-1) - s(n/2)| = |2(n/2-1) - 2(n-(n/2-1)-1)+1| = 1$ and $|s(n-1) - s(0)| = |2(n-(n-1)-1)+1-2 \cdot 0| = 1$. $\square$

Describing how moves in the torus are simulated by, at most, two moves in the grid requires the introduction of some more notation. Starting with a notion for moves and double moves in the next definition, it will be possible to give a precise and simple description of the translation of moves on the ring (which may be "to the left"($-1$), "to the right" ($+1$), or "remain where you are" ($0$)) into double moves on the array. We make additional use of the symbolic values "$-0$" and "$+0$" for the description of double moves. In fact both these zeros mean the move leads from a processor to itself using outer links. The real importance of $-0$- and $+0$-moves lies in two-dimensional tori and grids.

**Definition 5.2**

1. There are five kinds of possible *move directions*, represented by the symbols $-1$, $-0$, $0$, $+0$, and $+1$. A $-1$ ($+1$) represents a move to the left (right) and $0$ represents "remain where you are." The special symbols $-0$ and $+0$ represent a move to the left (resp. right) that turns around on half the way and returns to the processor it started, using the outer links.

2. A *move* is represented by a pair $(i, r)$, where $i \in \{0, \ldots, n-1\}$ denotes the processor where the move starts and $r \in \{-1, 0, +1\}$ denotes the direction of the move.

3. A *double move* is represented by a pair $(i, [r_1, r_2])$, where $i \in \{0, \ldots, n-1\}$ denotes the processor where the move starts and $r_1, r_2 \in \{-1, -0, 0, +0, +1\}$ denote the directions of the double move.

The following definition presents the transformation of moves on the ring to double moves on the array.

**Definition 5.3** The function $m$ mapping moves to double moves is defined as

$$m(i, r) = (s(i), \delta(i, r)),$$

where $\delta$ is defined via

| $\delta(i,r)$ | $r=-1$ | $r=0$ | $r=+1$ |
|---:|:---:|:---:|:---:|
| $i=0$ | $[-0,1]$ | $[0,0]$ | $[1,1]$ |
| $0<i<n/2$ | $[-1,-1]$ | $[0,0]$ | $[1,1]$ |
| $i=n/2-1$ | $[-1,-1]$ | $[0,0]$ | $[1,+0]$ |
| $i=n/2$ | $[+0,-1]$ | $[0,0]$ | $[-1,-1]$ |
| $n/2<i<n-1$ | $[1,1]$ | $[0,0]$ | $[-1,-1]$ |
| $i=n-1$ | $[1,1]$ | $[0,0]$ | $[-1,-0]$ . |

The definition of $m(i,r)$ guarantees that the resulting double move is in fact possible, e.g., that $[+0,-1]$ is applied only at the right border, while $[1,1]$ is never applied at the right border.

To show the correctness of our proposed translation of ring algorithms into array algorithms, we have to show that no link between neighboring processors is used for more than one transport at any point of time. To formalize this, we introduce the notion of a collision between double moves. Two double moves collide if they make use of the same link between two processors at the same point of time in the same direction. Note that a collision may occur only between the first moves or the second moves of double moves, because first and second moves take place at different times.

**Definition 5.4** A *collision* occurs if there are two double moves $(i,[t_1,t_2])$ and $(j,[r_1,r_2])$ such that $i=j$ and $r_1 \equiv t_1$ (collision during first move) or $i+r_1=j+t_1$ and $r_2 \equiv t_2$ (collision during second move). Herein "$\equiv$" means syntactic equality on $\{-1,-0,0,+0,+1\}$ (that is, for example, $-0 \not\equiv 0$).

**Lemma 5.5** *Let $(i_1,r_1)$ and $(i_2,r_2)$ be two moves on a ring and $(i_1,r_1) \neq (i_2,r_2)$. Then the corresponding double moves on the array $m(i_1,r_1)$ and $m(i_2,r_2)$ do not collide.*

*Proof.* We can safely assume $r_1, r_2 \neq 0$ since 0 stands for "remain where you are." The double moves $m(i_1,r_1)$ and $m(i_2,r_2)$ cannot collide during their *first* move unless $i_1 = i_2$, that is, they start from the same processor. Let us assume that indeed $i_1 = i_2 =: i$, but $r_1 \neq r_2$, i.e., $r := r_1 = -r_2$. Let $m(i,r) = (s(i), [r_{11}, r_{12}])$ and $m(i,-r) = (s(i), [r_{21}, r_{22}])$. From the definition of $m$ (see Definition 5.3 and compare the columns $r = -1$ and $r = +1$ for each value of $i$) follows $r_{11} \neq r_{21}$, so the first moves of the double moves $m(i,r)$ and $m(i,-r)$ lead into two different directions.

Showing that also the second moves of double moves do not collide completes the proof. The only possible directions for double moves are $[0,0]$, $[+1,+1]$, $[-1,-1]$, $[+1,+0]$, $[-1,-0]$, $[+0,-1]$, and $[-0,+1]$ as the definition of function $m$ shows. The only cases where the second components coincide but the first ones are different are the double moves $[+1,+1]$ and $[-0,+1]$ respectively $[-1,-1]$ and $[+0,-1]$. Inspection of the table given in Definition 5.3 shows that this may only occur for $i = 0$ in the first case and $i = n/2$ in the second case. But this implies that the second components of the respective double moves are performed by different array processors, making a collision impossible. In all other cases, it holds that if the second components of two such directions are identical, so are the first. This means that if a collision occurred during the second move, then there would be a collision during the first move. $\Box$

Having dealt successfully with the one-dimensional case, we now proceed with the definition of move, double move, and collision for two-dimensional tori and grids.

**Definition 5.6**

1. A *(two-dimensional) move* is a pair $(m_x, m_y)$ of one-dimensional moves $m_x$ and $m_y$, called the $x$- and $y$-part of $(m_x, m_y)$. A move $((i_x, r_x), (i_y, r_y))$ is performed by sending a packet from processor $(i_x, i_y)$ to processor $(i_x + r_x, i_y + r_y)$ over a link according to Figure 5.2.

2. A *(two-dimensional) double move* is a pair $(M_x, M_y)$ of one-dimensional

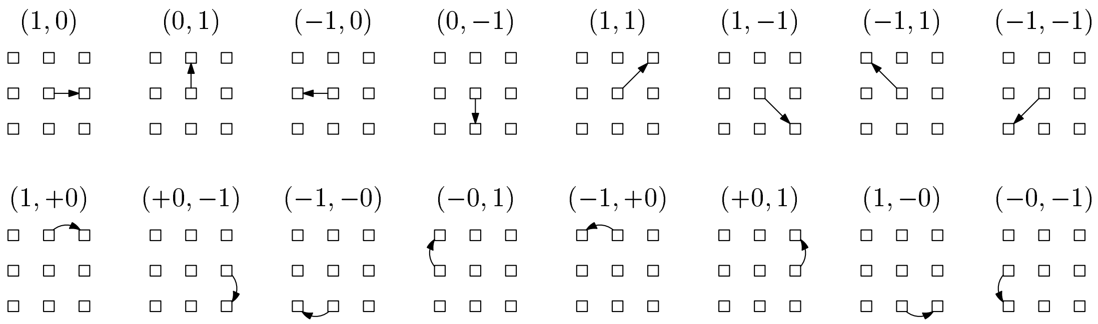


Figure 5.2: Realization of all valid moves $(r_x, r_y)$ in the two-dimensional case. The moves in the upper row employ conventional links, the moves in the lower row outer links.

double moves $M_x$ and $M_y$, called the $x$- and $y$-part of $(M_x, M_y)$. A double move $((i_x, [r_x, s_x]), (i_y, [r_y, s_y]))$ is performed by sending a packet from processor $(i_x, i_y)$ to processor $(i_x + r_x, i_y + r_y)$ and then to processor $(i_x + r_x + s_x, i_y + r_y + s_y)$ over the two links according to Figure 5.2 (first $(r_x, r_y)$, then $(s_x, s_y)$).

3. Two two-dimensional double moves $(M_x, M_y)$ and $(N_x, N_y)$ *collide* if both pairs $M_x$ and $N_x$, and $M_y$ and $N_y$ collide.

4. A (two-dimensional) move $(m_x, m_y)$ on the torus is mapped to a double move on the grid by the function $M(m_x, m_y) := (m(m_x), m(m_y))$.

The following lemma provides the correctness of our methodology also in the two-dimensional case.

**Lemma 5.7** *Let $(m_x, m_y)$, $(n_x, n_y)$ be two moves on a torus and $(m_x, m_y) \neq (n_x, n_y)$, then $M(m_x, m_y)$ and $M(n_x, n_y)$ do not collide.*

*Proof.* Let us assume that $M(m_x, m_y)$ and $M(n_x, n_y)$ do collide. Then $m(m_x)$ and $m(n_x)$, and $m(m_y)$ and $m(n_y)$ collide. By Lemma 5.5 we may conclude $m_x = n_x$ and $m_y = n_y$, a contradiction to the precondition $(m_x, m_y) \neq (n_x, n_y)$. $\square$

Now we are ready to state one of our main results. It provides a general translation of torus algorithms onto grids with a delay factor of 2. For this purpose, we introduce the two-dimensional embedding function $f$ from tori into grids. The *embedding function* $f$ from $\{0,\ldots,n-1\}\times\{0,\ldots,n-1\}$ to $\{0,\ldots,n-1\}\times\{0,\ldots,n-1\}$ maps torus processors to grid processors in a component-wise fashion with respect to the two-dimensional coordinates of the processors. That is, it makes use of the mapping $s$ from the one-dimensional case such that we have

$$f(i,j) := (s(i), s(j)).$$

The subsequent theorem now demonstrates that the mathematical embedding given by the functions $f$ and $M$ can be realized in our model of computation.

**Theorem 5.8** *An algorithm on a torus can be simulated on a grid of same size with delay* 2 *such that the uniquely determined processor* $f(i,j)$ *on the grid plays the role of processor* $(i,j)$ *on the torus.*

*Proof.* Besides performing internal operations, processors only send packets to their neighbors and receive packets from them. In every second step, processor $f(i,j)$ simulates processor $(i,j)$ by sending and receiving identical packets and performing identical internal operations. The directions of the sends and receives, however, are not identical, but the images under $M$. Within two steps, each packet must reach its destination going over one intermediate processor, which must route incoming packets into the appropriate direction. Fortunately, this is simple because the incoming and outgoing directions are always identical, so no additional information needs to be added to the packets themselves. □

We can now translate sorting algorithms for the torus into sorting algorithms for the grid. However, the indexing gets transformed. To get a sorting algorithm for the grid with respect to an arbitrary block-wise continuous indexing function directly, we show that the all-to-all mapping performed along the embedded torus also describes an all-to-all mapping on the grid without wrap-arounds.

If we consider a block in the grid then this block is normally not a block in the embedded torus. However, if a block $B$ in the grid has sidelength $2b$, where $b$ is the sidelength of a block on the torus, then $B$ is the image of four blocks of the torus.

Let $B$ denote blocks in the grid, and $A$ denote blocks in the torus. Let $I(i)$ denote the interval $[2ib, 2(i+1)b-1]$ for $i = 0, \ldots, n/(2b)-1$. Then let $B(i,j) = I(i) \times I(j)$ denote a block in the grid. Let $I_0(i)$ denote the set of even integers from $I(i)$ and $I_1(i)$ the odd ones. Then $A(i,j)[x,y] := s^{-1}(I_x(i)) \times s^{-1}(I_y(j))$ describes a block in the torus for all $x, y \in \{0,1\}$, where $s$ is defined as in the beginning of the section. It is easily seen that $A(i,j)[x,y] \cap A(l,m)[u,v] = \emptyset$ for $(x,y) \neq (u,v)$ or $(i,j) \neq (l,m)$ and all $x,y,u,v \in \{0,1\}$. That is, the union of all $A(i,j)[x,y]$ fills the whole torus. Further, note that

$$f^{-1}(B(i,j)) = A(i,j)[0,0] \cup A(i,j)[0,1] \cup A(i,j)[1,0] \cup A(i,j)[1,1].$$

**Lemma 5.9** *The function $ata' = f \circ ata \circ f^{-1}$ is an all-to-all mapping on the grid, if $ata$ is an all-to-all mapping on the torus.*

*Proof.* We have to show for all blocks $B(i,j)$ and $B(l,m)$ that $|ata'(B(i,j)) \cap B(l,m)| = c$ for a fixed value $c$. (For an $h$–$h$ problem $c = 16hb^4/n^2$.)

Since $ata$ is an all-to-all mapping on the torus, we have

$$|ata(A(i,j)[x,y]) \cap A(l,m)[u,v])| = c'$$

for all $i,j,l,m$ and all $x,y,u,v$. Therefore

$$|ata(f^{-1}(B(i,j))) \cap f^{-1}(B(l,m))| = 16c'.$$

Since $f$ is a bijection, we conclude

$$|f(ata(f^{-1}(B(i,j)))) \cap f(f^{-1}(B(l,m)))| = 16c'$$

for all $i,j,l,m$. Hence $ata'$ is an all-to-all mapping on the grid. □

For the following theorem remember that the complexity of sorting is asymptotically governed by the complexity of two all-to-all mappings (cf. Section 3).

**Theorem 5.10** *A grid with diagonals can solve the h–h sorting problem in asymptotically optimal $hn/6 + O(n^{2/3})$ steps for every blockwise continuous indexing scheme for $h \geq 12$.*

*Proof.* A torus can perform an all-to-all mapping for load 12 in $n/2 + O(n^{2/3})$ steps. Thus by Lemma 5.9 and Theorem 5.8, a grid can perform an all-to-all mapping in $n + O(n^{2/3})$ steps. The result easily generalizes to $h \geq 12$. □

# 6 Results for Small Loads Using Concentration Techniques

Concentrating data into a smaller area of a grid turns a 1–1 problem into an $h$–$h$ problem. Since $h$–$h$ problems were not studied intensively until quite recently [13] (though already Valiant and Brebner [29] considered them as early as in 1981 and others maybe even earlier), data concentration was introduced a short time ago [9]. The first use of concentration was to solve the 1–1 sorting problem in $2.5n+o(n)$ steps, while the previous best known bound without using concentration was $3n + o(n)$ [23]. (Today an optimal $2n + o(n)$ steps algorithm is known [3, 5].)

We solved 12–12 sorting in optimal time. Therefore $h$–$h$ sorting with $h < 12$ is a candidate for speed-up via concentration. Let us start with the fastest algorithm for grids in this paper, an algorithm for the 1–1 routing problem.

**Theorem 6.1** *Let $s(n)$ be the time a 9–9 sorting algorithm needs on a grid with diagonals. Then routing works on a grid with diagonals in $8n/9 + s(n/9) + O(n^{2/3}) \leq 89n/81+O(n^{2/3})$ steps and on a torus with diagonals it works in $4n/9+ s(n/9) + O(n^{2/3}) \leq 53n/81 + O(n^{2/3})$ steps.*

*Proof.* Let us divide the torus or the grid into 9 submeshes each $n/3 \times n/3$ big and each submesh into 9 subsubmeshes each $n/9 \times n/9$ big. We route a packet in three stages to its destination. The destination of each packet consists of a submesh-number (1–9), a subsubmesh-number (1-9), and a position within a subsubmesh ($(x, y) \in \{1, \ldots, n/9\} \times \{1, \ldots, n/9\}$). In the first stage, each packet whose subsubmesh-number is not already correct is shifted into one of the right subsubmeshes leaving its relative position in the subsubmesh unchanged. In the case of a torus, the nearest correct subsubmesh is chosen and the shift takes $n/9$ steps. In the case of a grid, the right subsubmesh within the original submesh is chosen and the shift takes $2n/9$ steps. (A direct approach to do this can be adapted from [12]. Another method is to embed the obvious routing scheme for a torus that takes $n/9$ steps into the subgrid. In order to get *exactly* $2n/9$ steps, the additional outer links are essential and also that $n/81$ is even since inside the grid outer links have to be simulated by diagonals leading into the neighboring subgrid. Otherwise, one additional step is necessary. Note that for the statement of the theorem only $2n/9 + O(n^{2/3})$ steps are necessary. This can be achieved easily even without outer links at all.) Next, the position within the subsubmesh is adjusted using 9–9 sorting algorithm for grids. This takes another $s(n/9)$ steps. Finally, each packet is routed to the right submesh without changing its position within the submesh. This takes $n/3$ steps for a torus and $2n/3$ steps for a grid. For grids, Theorem 6.2 says $s(n) \leq 17n/9 + O(n^{2/3})$. $\Box$

Krizanc and Narayanan [7] showed lower bounds for sorting on meshes and tori with diagonals: 1–1 sorting takes at least $1.166n$ steps on a grid and at least $n - o(n)$ steps on a torus if data replication is forbidden and queue-size bounded by 9. Using the first part of Theorem 6.1, they concluded that routing is faster than sorting on a grid. The second part of Theorem 6.1 now demonstrates that sorting is also harder than routing on *tori*.

Let us next turn to sorting. We present results for the 1–1, 2–2, 3–3, 4–4, 5–5, 6–6, 7–7, 8–8, and 9–9 sorting problem on grids with diagonals. The technique used in these algorithms is a combination of concentration and all-to-all mappings. This means, we present routing schemes that move all the data to a small area and simultaneously sending bricks from each block to each block. We can describe all routing schemes by diagrams that show data movement and load after each phase.

In general, we divide the mesh into square shaped clusters. In the beginning, we perform a local all-to-all mapping on each cluster individually. Then an equal portion of all clusters is sent into each cluster of the *concentration region*. This concludes the concentrating all-to-all mapping.

We must give a diagram for each cluster in the concentration area illustrating how it receives data from each cluster in the grid. We can cut down the number of these diagrams by exploiting symmetries.

**Theorem 6.2** *A grid with diagonals can solve the $h$–$h$ sorting problem in $t + O(n^{2/3})$ steps with buffer size $b$ using $c \times c$ many clusters that are concentrated into $d \times d$ many clusters located in the center, where $t$, $c$, $d$, and $b$ are as follows for the varying $h$.*

| $h$–$h$ | $t$ | $c \times c$ | $d \times d$ | $b$ |
|---|---|---|---|---|
| 1–1 | $\frac{6}{5}n = 1.2n$ | $20 \times 20$ | $4 \times 4$ | 25 |
| 2–2 | $\frac{7}{5}n = 1.4n$ | $10 \times 10$ | $4 \times 4$ | 15 |
| 3–3 | $\frac{6}{4}n = 1.5n$ | $8 \times 8$ | $4 \times 4$ | 13 |
| 4–4 | $\frac{8}{5}n = 1.6n$ | $10 \times 10$ | $6 \times 6$ | 14 |
| 5–5 | $\frac{5}{3}n \approx 1.67n$ | $6 \times 6$ | $4 \times 4$ | 12 |
| 6–6 | $\frac{7}{4}n = 1.75n$ | $8 \times 8$ | $6 \times 6$ | 13 |
| 7–7 | $\frac{11}{6}n \approx 1.83n$ | $12 \times 12$ | $10 \times 10$ | 14 |
| 8–8 | $\frac{13}{7}n \approx 1.86n$ | $14 \times 14$ | $12 \times 12$ | 15 |
| 9–9 | $\frac{17}{9}n \approx 1.89n$ | $18 \times 18$ | $16 \times 16$ | 16 |

*Proof.* We divide the $n \times n$ mesh into $c \times c$ square shaped clusters of size $n/c \times n/c$ each and describe an algorithm for a concentrating all-to-all mapping that routes $1/d^2$ of the data in each of the $c^2$ clusters into each of the $d^2$ clusters in the center of the mesh. The algorithm for concentrating all-to-all mapping consists of several phases. In each phase data are transported between neighboring clusters.

In principle, we have to describe $d^2$ routing schemes that concentrate $1/d^2$ of the data in each of the $c^2$ clusters in one of the $d^2$ center clusters. These routes all are scheduled in parallel. Actually, it is sufficient to give the description of $(d-2)d/8 + d/2$ types of these routes, because, due to symmetry, we only face $(d-2)d/8 + d/2$ basically different types of goal clusters. The above numbers were obtained from a program that recursively searches a routing scheme for a $c \times c$ to $d \times d$ concentrating all-to-all mapping. The data produced by the program is quite big.

We present the routing scheme only for the 3–3 problem in detail, since the involved diagrams describing the schemes are quite space consuming.[2]

For the 3–3 problem, we have $c = 8$ and $d = 4$, so up to symmetry there are 3 goal clusters, called $A$, $B$, and $C$ (see Figure 6.1). For these three types we present each time five diagrams exhibiting the routing scheme that takes five phases for the 3–3 problem.

For the 3-3 problem, we define the basic transport unit to be $n^2/1024$ packets. This means that the original load of $3n^2/64$ packets of a cluster consisting of $n^2/64$ processors consists of 48 basic transport units, where a group of 3 of them have the same destination but may go different ways. An arrow simply means a move of a transport unit from a cluster to one of its neighbors. Labeled arrows denote multiple transport units. The superposition of everything does not exceed 16 basic transport units, which means that not more than $n^2/64$ packets from any cluster are transported to each of its neighbors during one phase. Figure 6.1 depicts the three routing schemes. You can check the correctness by counting the arrows between two arbitrary neighbors, taking the 4 existing symmetries into account.

Altogether this implies that one phase works in $n/8$ steps. Thus we can realize a concentrating all-to-all mapping by first all-to-all mapping all clusters

[2] An algorithm showing all cases $h \leq 9$ is available under the following address: `http://www-fs.informatik.uni-tuebingen.de/~reinhard/concmap.html`.

individually ($n/8$ steps) and then performing the five phases ($5n/8$ steps). □

Concentration is a technique not very well suited for tori, since a proper subgrid of a torus is a grid, but *not* a torus. The gain by concentration is usually more than compensated by the loss of having to work on a grid instead of a handy torus. We overcome this difficulty by *data replication*, but only in the following case. All other algorithms in this paper never copy data packets.

**Theorem 6.3** *If replication is allowed, a torus can solve the* 1–1 *sorting problem in* $2n/3 + O(n^{2/3})$ *steps with buffer size* 9.

*Proof.* We divide the $n \times n$ torus into 9 subgrids of size $n/3 \times n/3$. We concentrate *all* data in all 9 subgrids, which requires data replication. Now all 9 subgrids contain all data, identically. We can now use our sorting algorithm *for tori* individually on all 9 subgrids, sorting in layer first order. We can use the algorithm for tori, since each subgrid behaves just as a torus: If some element is shifted downwards across the border of the subgrid, it reappears at the upper border, since in the above subgrid the same algorithm shifts the same data element downwards.

In the end, all nine subgrids contain all data sorted in layer first order. Now the $i$th subgrid again gets rid of all data save the $i$th layer, hence all data is sorted. The concentration takes $n/3$ steps and sorting of subgrids takes $n/3 + O(n^{2/3})$ steps according to Theorem 4.1. Thus the overall running time is $2n/3+O(n^{2/3})$. □

Data replication was crucial in order to achieve this running time. Krizanc and Narayanan showed that with buffer size 9, and without making copies, a torus with diagonals needs at least $n - o(n)$ steps for sorting [7].

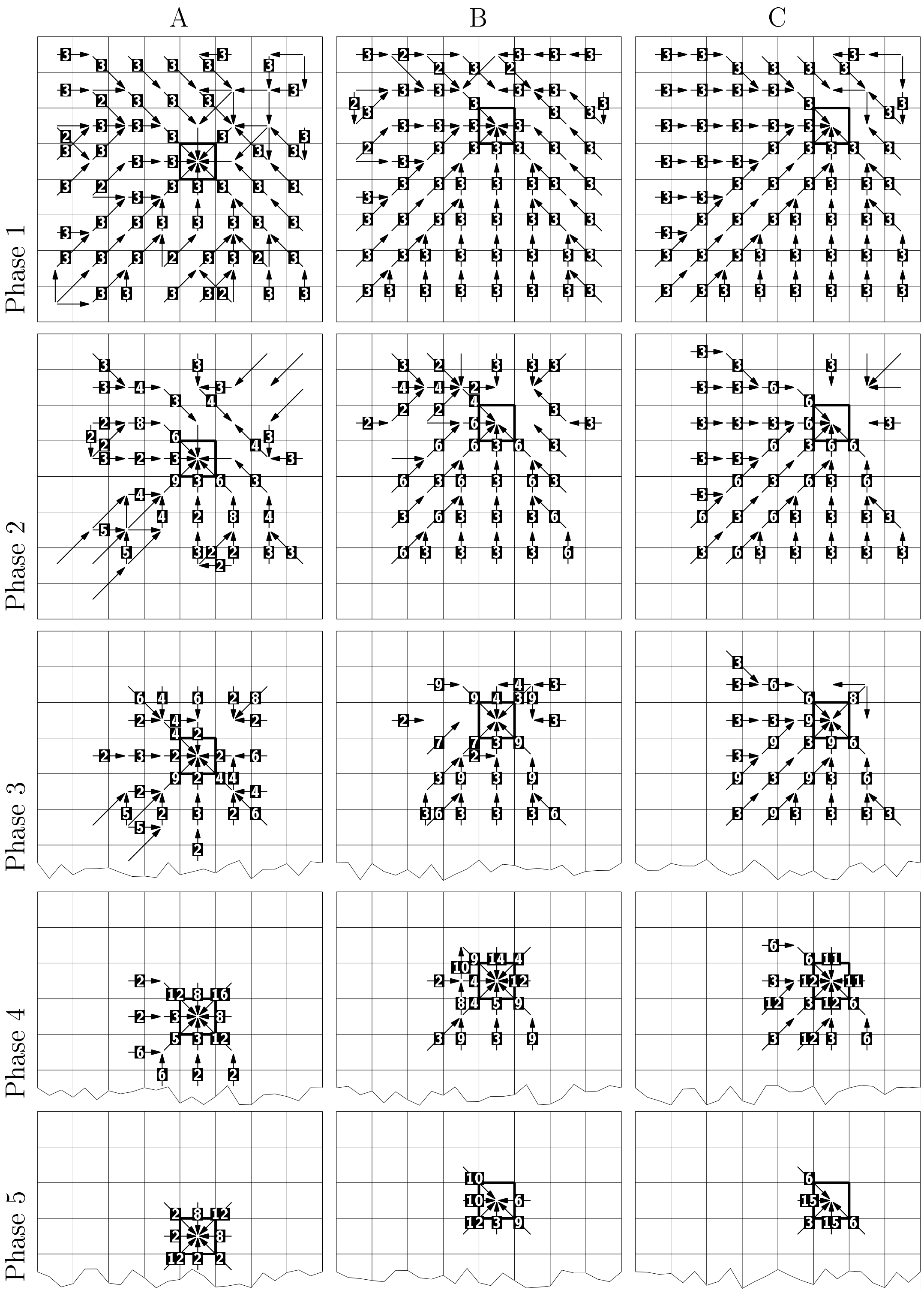


Figure 6.1: The three types of routing schemes of an algorithm for concentrating all-to-all mapping for the 3–3 sorting problem. $A$, $B$, and $C$ represent different types of destinations. The schemes for all other destinations are symmetrical to one of them. The goal is to get $1/64$ of the data in each submesh into each destination submesh. This works in 5 phases of $n/8$ steps each. Each arrow denotes transport of $3n^2/64$ packets, labeled arrows a multiple thereof according to the label.

# 7 Historical remarks

In this section, we give a short account to the main points in the history of sorting and routing algorithms on grids (also see [25]).

Thompson and Kung [28] and Nassimi and Sahni [20] were the first that presented $O(n)$ steps algorithms for sorting on meshes. In 1986, Schnorr and Shamir [23] presented an optimal $3n+o(n)$ steps algorithm under the assumption of buffer size 1 (also see [8] for the corresponding lower bound). Schnorr and Shamir's result has been improved for buffer size greater than 1. Introducing concentration techniques, the running time could be improved to $2.5n + o(n)$ steps [9]. Then Kaklamanis and Krizanc developed an optimal $2n + o(n)$ steps algorithm, which, however, was randomized [3]. Finally, Kaufmann, Sibeyn, and Suel derandomized the latter algorithm and found the first deterministic, asymptotically optimal algorithm [5]. Note that for the corresponding routing problem an optimal algorithm, even up to additive constants (it matches the distance bound $2n - 2$), had been known for quite some time [18].

As to the $h$–$h$ sorting problem for $h \geq 8$, first a $hn + o(n)$ steps algorithm for sorting was given [9]. Later an optimal randomized $hn/2 + o(n)$ steps algorithm matching the bisection bound was discovered [4]. Recently, the first optimal deterministic algorithm was presented [10]. Later, by derandomizing the optimal randomized algorithm, Kaufmann, Sibeyn, and Suel obtained the same algorithm in a different way [5].

For meshes with diagonals, first a result better than the bisection bound for meshes without diagonals, that is, a $2hn/9 + o(n)$ steps algorithm, was presented [12]. We improved this to optimal $hn/6 + o(n)$ steps using completely new techniques.

For deterministic *average case* sorting for grids and tori with and without diagonals, one can obtain results that, in general, are twice as fast as in the worst case [11].

# 8 Conclusion

Doubling the capacity of each individual communication link in a mesh obviously leads to twice as fast algorithms. By adding diagonal connections, we not only doubled the *overall* capacity of all communication links, but got *three times* as fast algorithms. This counterintuitive result suggests that parallel computers should be built as grids or tori with diagonals rather than without diagonal links, though the algorithms are not practical for small processor numbers. The constant factors in the low order terms, mostly $O(n^{2/3})$, are rather high. To give an idea of the low order terms, take the algorithm for 12–12 sorting on a torus. There are two all-to-all mappings each taking $(1+1/(2n^{1/3}+1))n/2 \leq n/2+n^{2/3}/4$. Additionally, we need to sort $n^{2/3} \times n^{2/3}$ submeshes and $2n^{2/3} \times n^{2/3}$ submeshes two times. Making the rough assumption that 12–12 sorting on a $n^{2/3} \times n^{2/3}$ mesh can be performed in at most $6n^{2/3}$ steps and on a $2n^{2/3} \times n^{2/3}$ mesh in at most $12n^{2/3}$ steps (both without additional low order terms), this gives an upper bound of $n + 37n^{2/3}$ steps. The low order term does not seem so big, but for small $n$ it makes a large contribution to the running time. Note that even for $n = 50000$ the value of $37n^{2/3}$ is larger than $n$. The low order terms of the other algorithms are similar in magnitude. We presented asymptotically optimal algorithms for all practical (with respect to load) cases (i.e., large $h$), and many other cases as well. However, the development of algorithms with smaller low order terms, which would be of practical benefit, remains an open problem.

By using a different sorting scheme with only *one* all-to-all mapping, it is possible to halve the running times of many of our algorithms in the *average case* [11]. Particularly, one can obtain, in the average, optimal $h$–$h$ sorting and routing algorithms for tori and grids with diagonals for *all* $h$ (see [11]).

**Acknowledgment** We thank Corinne Chauvin for writing the program that computes the exact data movements between clusters to get concentrating all-to-all mappings in Theorem 6.2. We are also indebted to anonymous referees for pointing out numerous improvements and corrections concerning the presentation of the paper.

# References


[1] G. Bilardi and F. P. Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27:172–182, 1995.

[2] Y. Feldman and E. Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, 35(10):61–73, October 1992.

[3] C. Kaklamanis and D. Krizanc. Optimal sorting on mesh-connected processor arrays. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 50–59, 1992.

[4] M. Kaufmann, S. Rajasekaran, and J. F. Sibeyn. Matching the bisection bound for routing and sorting on the mesh. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 31–40, 1992.

[5] M. Kaufmann, J. F. Sibeyn, and T. Suel. Derandomizing algorithms for routing and sorting on meshes. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 669–679, 1994.

[6] M. Kaufmann and J.F. Sibeyn. Randomized multipacket routing and sorting on meshes. *Algorithmica*, 17:224–244, 1997.

[7] D. Krizanc and L. Narayanan. Sorting and selection on arrays with diagonal connections. In *First Canada-France Conference on Parallel and Distributed Computing*, volume 805 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, May 1994.

[8] M. Kunde. Lower bounds for sorting on mesh-connected architectures. *Acta Informatica*, 24:121–130, 1987.

[9] M. Kunde. Concentrated regular data streams on grids: Sorting and routing near to the bisection bound. In *Proceedings of the 32d IEEE Conference on Foundations of Computer Science*, pages 141–150, 1991.

[10] M. Kunde. Block gossiping on grids and tori: Sorting and routing match the bisection bound deterministically. In T. Lengauer, editor, *Proceedings of the 1st European Symposium on Algorithms*, number 726 in Lecture Notes in Computer Science, pages 272–283, Bad Honnef, Federal Republic of Germany, September 1993. Springer-Verlag.

[11] M. Kunde, R. Niedermeier, K. Reinhardt, and P. Rossmanith. Optimal average case sorting on arrays. In E. W. Mayr and C. Puech, editors, *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science*, number 900 in Lecture Notes in Computer Science, pages 503–514. Springer-Verlag, 1995.

[12] M. Kunde, R. Niedermeier, and P. Rossmanith. Faster sorting and routing on grids with diagonals. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science*, number 775 in Lecture Notes in Computer Science, pages 225–236. Springer-Verlag, 1994.

[13] M. Kunde and T. Tensi. ($k$–$k$) Routing on multidimensional mesh-connected arrays. *Journal of Parallel and Distributed Computing*, 11:146–155, 1991.

[14] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings 1978*, pages 256–282. Society for Industrial and Applied Mathematics, 1979.

[15] F. T. Leighton. Methods for message routing in parallel machines. *Theoretical Computer Science*, 128:31–62, 1994.

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

[17] T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[18] T. Leighton, F. Makedon, and I. Tollis. A $2n-2$ step algorithm for routing in an $n \times n$ array with constant size queues. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, 1989.

[19] Y. Ma, S. Shen, and I. D. Scherson. The distance bound for sorting on mesh-connected processor arrays is tight. In *Proceedings of the 27th IEEE Conference on Foundations of Computer Science*, pages 255–263, 1986.

[20] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Trans. Computers*, C-30(2):101–106, 1981.

[21] S. Rajasekaran. $k$–$k$ routing, $k$–$k$ sorting, and cut-through routing on the mesh. *Journal of Algorithms*, 19:361–382, 1995.

[22] S. Rajasekaran and R. Overholt. Constant queue routing on a mesh. *Journal of Parallel and Distributed Computing*, 15(2):160–166, June 1992.

[23] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 255–263, 1986.

[24] A. Schorr. Physical parallel devices are not much faster than sequential ones. *Information Processing Letters*, 17:103–106, 1983.

[25] J. F. Sibeyn. Overview of mesh results. Technical report MPI-I-95-1-018, Max-Planck-Institut für Informatik, Saarbrücken, 1995.

[26] J. F. Sibeyn. Routing on triangles, tori and honeycombs. In W. Penczek and A. Szatas, editors, *Proceedings of the 21st Conference on Mathematical Foundations of Computer Science*, number 1113 in Lecture Notes in Computer Science, pages 529–541, Cracow, Poland, September 1996. Springer-Verlag.

[27] L. Snyder. Introduction to the configurable, highly parallel computer. *Computer*, pages 47–56, January 1982.

[28] C. T. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20:263–270, 1977.

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

[30] P. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM Journal on Computing*, 17(4):659–672, August 1988.