Faster Sorting and Routing on Grids with Diagonals

Manfred Kunde Rolf Niedermeier* Peter Rossmanith*

Fakultät für Informatik, Technische Universität München
80290 München, Arcisstr. 21, Fed. Rep. of Germany

Abstract

We study routing and sorting on grids with diagonal connections. We show that for so-called h-h problems faster solutions can be obtained than on comparable grids without diagonals. In most of the cases the number of transport steps for the new algorithms are less than half of the principle smallest number given by the bisection bound for grids without diagonals.

1. Introduction

Over the last decade a well-studied topic of parallel algorithmics was sorting and routing on fixed-size networks of processors. A lot of papers presented optimal and nearly optimal solutions for the conventional two-dimensional $n \times n$ grid (or mesh) with four-neighborhood. In many cases algorithms for tori of processors (meshes with wrap-around connections) derive from algorithms for ordinary meshes. The additional amount of hardware for the wrap-arounds leads in many cases to solutions twice as fast. Meshes with diagonals (with eight-neighborhood) have twice the number of data channels. 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 [8, 16], near to nothing is known how to exploit the additional communication links for faster sorting and routing. For mesh-connected arrays with diagonals we present

* Work supported by the Deutsche Forschungsgemeinschaft, SFB 342, A4 "KLARA."

Email: kunde/niedermr/rossmani@informatik.tu-muenchen.de

deterministic routing and sorting algorithms that often halve the number of parallel data transfers compared to the best possible solutions on grids without diagonals. In some cases the number of transport steps is strictly smaller than the half. On the other hand, in a fixed period of time we sort and route *always* strictly more than the double amount of data compared to grids without diagonals.

In the following we speak of h-h problems when each processor initially and finally contains exactly h packets. 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. The fastest ones need $3n + o(n)$ steps [12, 15, 17]. For buffer size 2 we showed how to solve the 1-1 sorting problem deterministically in $2.5n + o(n)$ transport steps [6]. Recently, Kaklamanis and Krizanc [1] presented a randomized algorithm (needing constant buffer size) that sorts in only $2n + o(n)$ steps with high probability. For 1-1 routing Leighton, Makedon, and Tollis [11] presented optimal deterministic algorithms (with constant buffer size) that exactly match the distance bound of $2n - 2$ steps. Rajasekaran and Overholt [14] further reduced the buffer size. We present algorithms for grids with diagonals that need $1.33n + o(n)$ steps for 1-1 sorting and $1.11n + o(n)$ steps for routing. For grids with diagonals we summarize some of our new results in Table 1 and compare them with the so far known best results on grids without diagonals. The table contains results obtained by deterministic as well as randomized algorithms (the latter ones are marked with an asterisk). We omit all sublinear terms, which are of no importance for the asymptotic complexity.

Kaufmann, Rajasekaran, and Sibeyn [2, 3, 4] gave randomized algorithms for h-h problems on an $n \times n$ mesh that need $hn/2 + o(hn)$ steps if $h \geq 8$. We showed that it is possible to solve the h-h routing and sorting problems within the same number of steps deterministically [7]. 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 $2hn/9 + O(hn^{2/3})$ steps for deterministic h-h sorting and routing, provided that $h \geq 9$. This gives an acceleration factor of 2.25. (In the paper we deal only with the 9-9 problem. The algorithm easily generalizes to $h > 9$.)

For wrap-around meshes (or tori) without diagonals Kaufmann and Sibeyn [3] gave a randomized h-h sorting algorithm with $hn/4 + o(hn)$ steps for $h \geq 8$. There exists an equally fast deterministic algorithm [7]. Both algorithms match asymptotically the respective bisection bound of $hn/4$. If

Problem	*New results with diagonals*	*Without diagonals*	
1-1 routing	$1.11n$	$2n$	Leighton et al. [11]
1-1 sorting	$1.33n$	$2.5n$	Kunde [6]
		$2n*$	Kaklamanis and Krizanc [1]
2-2 routing	$1.67n$	$3n$	Park and Balasubramanian [13]
		$2.67n*$	Sibeyn and Kaufmann [3]
2-2 sorting	$1.67n$	$3n$	Park and Balasubramanian [13]
4-4 routing	$2n$	$4n$	Kunde [6]
		$2n*$	Kaufmann et al. [2]
4-4 sorting	$2n$	$4n$	Kunde [6]
		$3n*$	Sibeyn and Kaufmann [3]
8-8 sorting		$4n$	Kunde [7]
9-9 sorting	$2n$		*randomized

Table 1 Comparison of results between grids with and without diagonals.

we add diagonals to tori, we can sort and route in only $hn/10 + O(hn^{2/3})$ steps if $h \geq 10$. That means we get a speedup of 2.5. 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 $h \leq 10$ needs still $n + o(n)$ steps.

The results of this paper demonstrate that grids with diagonals are a promising architecture, because they improve the times for sorting and routing. Note that grids with diagonals are not substantially more complicated than plain grids: They have constant degree and are scalable.

We use a sorting method based on all-to-all mappings [7]. Roughly speaking, this method 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 next section we provide some basic definitions. Afterwards we explain the all-to-all mapping and show how to sort with its help. Then we present all algorithms that achieve the results of Table 1. We conclude this paper with sorting algorithms on tori with diagonals ans some open questions.

2. Preliminaries

In this section we present some basic definitions and notations we use in this paper. Leighton presents further details and motivation in his recent book [9].

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 channel to processor (r', c') if $\max\{|r - r'|, |c - c'|\} = 1$. We speak of a "diagonal" connection if $|r - r'| = |c - c'| = 1$. For grids with wrap-arounds (tori) the processors in the two border rows and two border columns are also connected to a processor in the opposite border. In this way 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 specifying the processor to which it must travel. Each processor is destination of exactly h packets. The routing problem is to transport each packet to its destination address. There is a close relationship between the full h-h routing problem and the h-h sorting problem where the packets are elements from a linearly ordered set. Each packet or element in a processor P lies in a (memory) place (P, j), where $0 \leq j < h$. For a given j the set of places $\{ (P, j) \mid P \text{ is a processor } \}$ is called the jth 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 $\{0, \ldots, hn^2 - 1\}$. The sorting problem (with respect to g) is to transport the ith 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.

The model of computation is the conventional one [9, 10], where only nearest neighbors exchange data. In one step a communication channel can transport at most one packet (as an atomic unit) 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; we ignore operations within a processor, especially between different layers.

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

In this section first 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. Then we discuss how we can use the same methods even for partial h-h routing problems. You can find a more detailed description in the paper that introduced all-to-all mappings [7].

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 a layer of such a subblock a *brick*. That means each block contains hm^2 bricks arranged in h layers. We number the blocks from 0 to $m^2 - 1$ where 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 continuous. To see the correctness of the following sorting method we use the 0-1 principle [5, 9].

In a first step we sprinkle all data all 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 ith 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 h bricks to every block on the mesh as illustrated in Figure 1. Now each block contains almost the same number of ones (the difference is at most hm^2). We call such a global distribution of data an *all-to-all mapping* [7].

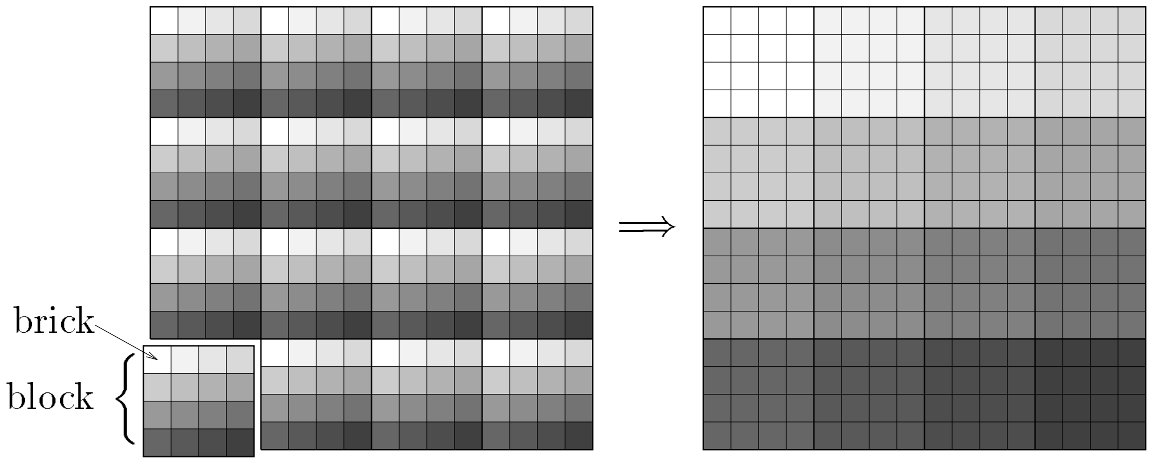


Fig. 1 The standard all-to-all mapping on a mesh with 16 blocks.

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 kth or $(k+1)$st brick, provided that a brick contains at least hm^2 elements.

Now an all-to-all mapping sends the first h bricks of each block to the first block, the second h bricks of each block to the second block, and so on. Afterwards all dirty bricks are in the $\lfloor k/h \rfloor$th or $\lfloor (k+1)/h \rfloor$th block, so the whole mesh is nearly sorted. To finish, we sort all adjacent pairs of blocks. Herein adjacent refers to a continuous block indexing.

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 hm^2 elements. There are hm^4 bricks in the mesh that contains altogether hn^2 elements, so each brick contains n^2/m^4 elements. The above condition implies

$$\frac{n^2}{m^4} \geq hm^2.$$

This inequality leaves a lot of freedom. We choose $m \leq \sqrt[3]{n/h}$ such that $m = \Theta(\sqrt[3]{n/h})$. In total the complexity of the all-to-all mapping asymptotically governs the time of our method.

The above sorting algorithm directly applies to full h-h routing problems. For doing so, assume an order of the packets according to any fixed continous blockwise indexing of their addresses. Applied on a partial h-h routing problem with a total loading of 75 percent, for example, the sorting algorithm would route the packets to wrong destinations. For packet routing problems, however, we can use the fact that packets know their final destination, although they may have trouble to choose the right way to that destination. Instead of sorting the blocks in the third 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 $hm^2 + e_j$, where $\sum_j e_j \leq k^2 \leq m^2$. In this case at most e_j packets have to 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 neglectable.

4. Sorting for loads of four and nine elements

In this section we solve the multi-packet problem for large numbers of packets. Since sorting based on all-to-all mappings needs asymptotically $2n$ steps, we try to sort as many packets as possible within this time bound. We start with an algorithm for the 4-4 sorting problem to present the fundamental ideas of a more complicated algorithm for the 9-9 problem.

The basic ideas: Sorting for load four

Deterministic 4-4 sorting on simple grids is possible in $4n + o(n)$ steps [6]. For grids with diagonals we can halve the number of steps.

Theorem 1. *The 4-4 sorting problem requires at most $2n + o(n)$ steps on a mesh with diagonals.*

Proof. We show how to solve the all-to-all mapping for the 4-4 problem in n steps. Let us divide the mesh into four equally sized submeshes as shown in Figure 2, that is into four squares located in the four corners. We call the submeshes A, B, C, and D. The four layers of A are A_A, A_B, A_C, and A_D. By the all-to-all mapping each block gets $h = 4$ bricks from each other block. So each brick has a source block, from where it comes and a target block, where it goes. The position of a brick within each block is at the moment of no importance.

Our strategy is divide and conquer: We recursively perform an all-to-all mapping on all four submeshes individually. We do this in such a way that in submesh A every brick already reaches its target block relative within submesh A. As a consequence each brick with destination address within A already has reached its final position. To complete the all-to-all mapping on the whole grid, we move all bricks into the proper submesh preserving their relative positions. We perform this final transportation in exactly $n/2$ steps as follows: All packets in A that preserve their location move to the first layer A_A, packets whose target block is in B move to A_B, and so on. All these movements take place within processors and thus do not count for the

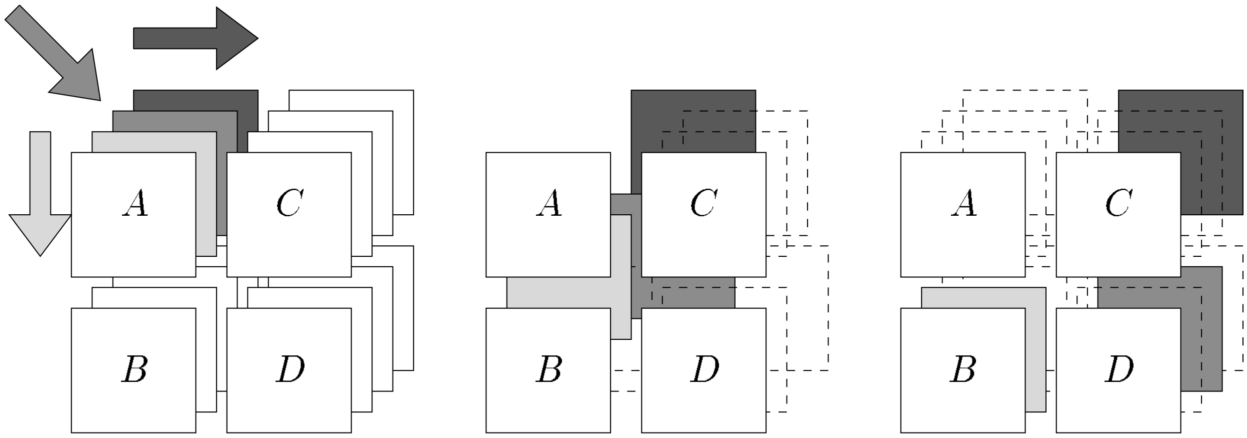


Fig. 2 How the three layers of the first quarter reach their destinations.

time bound. Then layer A_B moves as a whole to B, A_C moves to C, and A_D moves to D. This takes exactly $n/2$ steps.

Summing up the time requirements of all recursive calls yields the total running time. The final transportation as described takes $n/2$ steps on an $n \times n$-grid. In general a transportation step on an $n/2^i \times n/2^i$-grid takes $n/2^{i+1}$ steps. Altogether we need

$$\sum_{i=1}^{k} \frac{n}{2^i} < n$$

steps for the whole recursive method, if there are $2^k \times 2^k$ blocks in the grid. Independent of the block-size we need always less than n steps for a 4-4 all-to-all mapping. ∎

To attack the 9-9 problem, it will be helpful to look at the above algorithm from a different point of view. Recall what an all-to-all mapping should do. There are m^2 blocks arranged in m columns and m rows. Each of those m^2 blocks contains hm^2 bricks. It seems natural to address blocks by pairs $(x, y) \in \{1, \dots, m\} \times \{1, \dots, m\}$, where x denotes the column and y the row. That are the *Cartesian coordinates* of a block. Each brick has a *source block* and a *target block.* Each block is target of hm^2 bricks; from each block it receives exactly h bricks. Our task is to move all bricks to their target block. From the source and target blocks of a brick we can compute the path it travels, assuming that we use the above stated recursive algorithm. For this end we better use a slightly different scheme to address blocks. Since m is a power of two, m^2 must be a power of four. We use a *quaternary* number

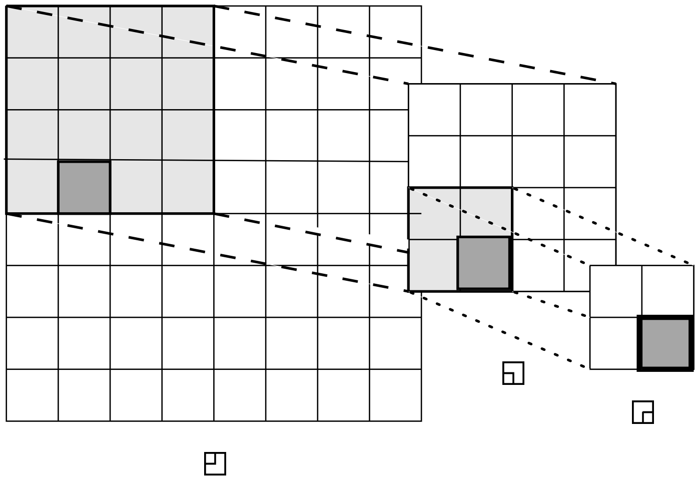


Fig. 3 The quad coordinates are (◰, ◱, ◲).

system to address blocks using the four symbols ◲, ◱, ◳, and ◰ as digits. An example reveals best how to apply this number system.

In Figure 3 we consider a grid containing 8×8 blocks. We want to address the block in the fourth row and second column, i.e., the block whose Cartesian coordinates are $(4, 2)$. If we divide the grid into four equal squares, our brick is located in the upper-left one. Therefore the most significant digit is ◰, a symbol meaning "upper-left." If we extract the upper-left subgrid and divide it again in four squares, our block is now in the lower-left one. This corresponds to ◱, which is the next digit. We can subdivide once more and get ◲ as the rightmost digit. The address is (◰, ◱, ◲). We will subsequently call this kind of address the *quad coordinate* of a block. What is the connection between the quad coordinates of a brick's source and target block and the path on which it travels?

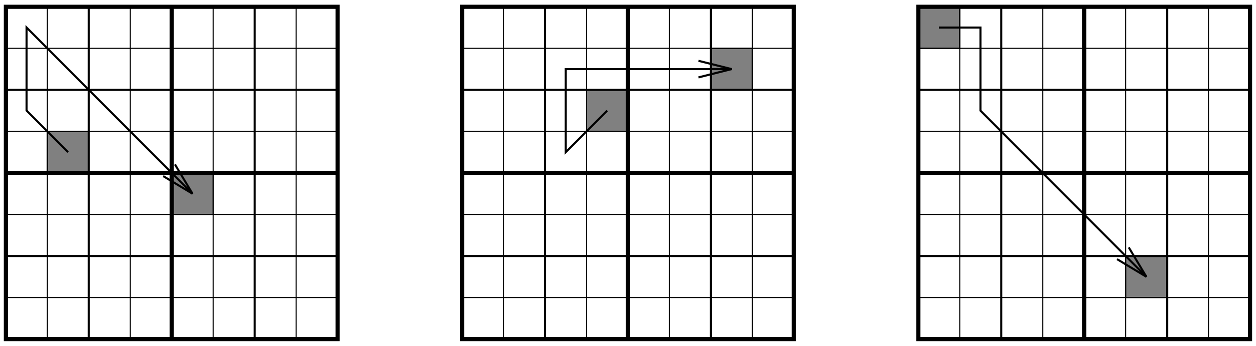


Fig. 4 Three examples of paths on which bricks travel to their destination.

Figure 4 contains three examples for bricks that move from (◰, ◱, ◲) to (◲, ◰, ◰), from (◰, ◲, ◳) to (◳, ◳, ◱), and from (◰, ◰, ◰) to (◲, ◱, ◰).

A packet's path always consists of three shifts. Each of these shifts is either horizontal, vertical, or diagonal. The lengths of the shifts are 1, 2, and finally 4 blocks wide. If a source block of a packet is (s_1, s_2, s_3) and the destination block is (d_1, d_2, d_3), then the first shift leads to (s_1, s_2, d_3), the second one to (s_1, d_2, d_3), and the third one to the final destination (d_1, d_2, d_3). So the ith shift depends only on s_i and d_i. We compute the path of a brick by subtracting its destination address from its source address using componentwise subtraction from Table 2. This is a special way of subtraction: The difference between two coordinates is a path between them.

$-$	◲	◳	◱	◰
◲	$\cdot$	$\downarrow$	$\rightarrow$	$\searrow$
◳	$\uparrow$	$\cdot$	$\nearrow$	$\rightarrow$
◱	$\leftarrow$	$\swarrow$	$\cdot$	$\downarrow$
◰	$\nwarrow$	$\leftarrow$	$\uparrow$	$\cdot$

Table 2 Computing the path between two quad coordinates.

We compute the three paths in Figure 4 as follows:

$$(\text{◰},\text{◱},\text{◲}) - (\text{◲},\text{◰},\text{◰}) = (\searrow,\uparrow,\nwarrow),$$
$$(\text{◰},\text{◲},\text{◳}) - (\text{◳},\text{◳},\text{◱}) = (\rightarrow,\uparrow,\nearrow),$$
$$(\text{◰},\text{◰},\text{◰}) - (\text{◲},\text{◱},\text{◳}) = (\searrow,\downarrow,\rightarrow).$$

Our findings yield the following algorithm:

Algorithm F. (All-to-all mapping for load four)
for $i = 1$ **to** k **do**
  for all $(s_1, \ldots, s_k)$, $(d_1, \ldots, d_k)$ **do in parallel**
    Shift the brick with source address $(s_1, \ldots, s_k)$ *and destination address* $(d_1, \ldots, d_k)$ *in direction* $d_i - s_i$ *for* $n/2^i$ *steps.*

By an *i-shift* we understand to route all bricks in direction $d_i - s_i$, subtracting according to Table 2. The distance of the transport is 2^{n-i} for a $2^n \times 2^n$ grid. For different bricks the direction $d_i - s_i$ and the destination address $(d_1, \ldots, d_k)$ of a brick. A i-shift changes exactly the ith component of a quad coordinate from source to destination. We realize an all-to-all mapping by performing i-shifts for all i. The order of these i-shifts is arbitrary.

Sorting for load nine: The algorithm

We presented a simple and efficient algorithm for the 4-4 problem. It used all principles that we are going to reuse now in order to find an algorithm for the 9-9 sorting problem. Besides old ideas from above we have to use some new techniques.

Theorem 2. *The 9-9 sorting problem requires at most $2n + o(n)$ steps on a mesh with diagonals.*

To develop an algorithm for the 9-9 sorting problem, we divide the grid in *nine* subgrids, called *ninth*s, rather than four. We have to transport one layer from each ninth to each ninth in order to mix data among them. Then we complete the all-to-all mapping by applying this procedure recursively on each ninth, just like Algorithm F did. We can describe this method alternatively by introducing *ninth-coordinate*s (using symbols ◰, ◧, ...) in analogy to quad coordinates. Assume that the $n \times n$ grid consists of $3^k \times 3^k$ blocks. Every block contains $3^k \times 3^k \times 9$ bricks. Each block has its unique ninth-coordinate $(s_1, \ldots, s_k)$. From each block exactly nine bricks must travel to each other block. The following algorithm transports a brick from its *source block* $(s_1, \ldots, s_k)$ to its *destination block* $(d_1, \ldots, d_k)$. During the whole transport a brick stays together as an atomic unit. The transport consists of k phases. In each phase a brick moves a certain distance in some direction. Here a "direction" is not as simple as in Algorithm F. Besides the nine directions $(\cdot, \uparrow, \nearrow, \rightarrow, \searrow, \downarrow, \swarrow, \leftarrow, \nwarrow)$ from Algorithm F, Algorithm N uses some more directions, directions that *change* in exactly the middle of the journey: $\uparrow\!\!\nearrow$, $\swarrow\!\!\bullet$, $\leftarrow$, and so on. Direction $\uparrow\!\!\nearrow$, for example, means "travel the first half of the time upwards and the second half of the time to the northeast." There is also $\swarrow\!\!\bullet$, which means "stay were you are during the first half of time and travel then to the southwest" and $\leftarrow$, which means "travel to the west during the first half of time and stay then where you are." In the ith phase Algorithm N computes a new direction by subtracting the ith component of the destination address from the ith component of the source address in ninth coordinates. It subtracts according to Table 3.

We have not yet solved all problems that Algorithm N bears. It is not at all obvious that we can carry out all the shifts at once. Nevertheless, it is better to look at Algorithm N now and to verify that it really performs an all-to-all mapping, *if it can be implemented at all.* We'll fight the details of the implementation later.

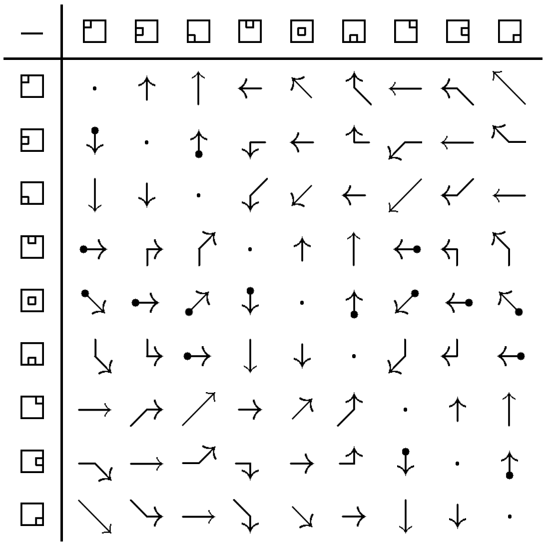


Table 3 Computing the path between two ninth-coordinates.

Algorithm N. (All-to-all mapping for load nine)
for $i = 1$ **to** k **do**
  for all $(s_1, \ldots, s_k)$, $(d_1, \ldots, d_k)$ **do in parallel**
   Shift the brick with source address $(s_1, \ldots, s_k)$ *and destination address* $(d_1, \ldots, d_k)$ *in direction* $d_i - s_i$ *for* $2n/3^i$ *steps.*

We check the correctness by verifying that shifts in direction $d_i - s_i$ really route a brick from block $(d_1, \ldots, d_{i-1}, s_i, s_{i+1}, \ldots, s_k)$ to block $(d_1, \ldots, d_{i-1}, d_i, s_{i+1}, \ldots, s_k)$. Let us consider for example $d_i =$ □ and $s_i =$ □. From Table 3 we get $d_i - s_i =$ ↖↑. So the brick is transported $n/3^i$ steps to the northwest and then $n/3^i$ steps upward, which is obviously correct.

In the same way we can verify the whole Table 3 and then we know that Algorithm N does the right thing. We have to check carefully, whether all these shifts can be performed *in parallel.* Each shift itself is easy, but superimposing all shifts could lead to overloaded communication lines. Superimposing all shifts of the first $n/3^i$ steps leads to a special routing problem shown in Figure 5. We obtain the same figure also for the second $n/3^i$ steps.

You can read Figure 5 as follows: There are two arrows from ninth B to ninth A. That means that Algorithm N attempts to move two full layers of data from B to A. Why? Well, all bricks in ninth B have □ as part of their source address and this component will be changed to one of the nine possible addresses. One ninth of the bricks in B have □ as their destination address. They will move in direction ← according to Table 3. This corresponds to one full layer moving from B to A. Another ninth of B's bricks has destination

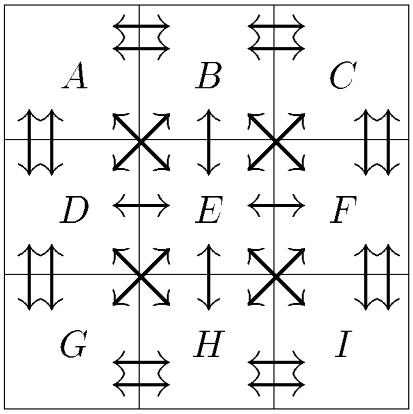


Fig. 5 All shifts at once.

address . They travel in direction . In the first $n/3^i$ steps they contribute another full layer that travels from B to A.

How can we transport *two* layers at once horizontally? At fist glance, this is easy: One layer uses horizontal lines and the other layer performs a *shiver shift* using diagonal lines . This simple solution, however, does not work in our setting, since we additionally have to move data from B to D and from E to A. These transports must use the diagonal lines that we wanted to reserve for the shiver shift.

We solve the problem by a generalization of shiver shifts as shown in Figure 6.

The zig-zag movement has different amplitude and period-length for different packets. In Figure 6 we show only the routes of the *rightmost* packets. All other packets move in the same fashion as the rightmost packet in the same row. We can think of this movement as rectangles moving to the northeast and bouncing at the upper border. The first rectangle is the lower half of the ninth. It bounces only once. The next rectangle is again the lower half of the rest of the ninth. It bounces twice. In Figure 6 the movements of the four rectangles are shown side by side to make it easier to read.

In this way all bricks are shifted horizontally to the left, save the uppermost row. We have to treat the uppermost row in a special way. One possibility to do so is to chop this row into pieces of $\sqrt{n}$ length and to move these pieces down, thus making the uppermost row empty. We also rotate the pieces for 90 degrees. We can move these additional packets left together with the remaining layer. It just has to open up a little bit like an accordion. Afterwards the pieces are put back into the upper row. The overhead is only $O(\sqrt{n})$ steps.

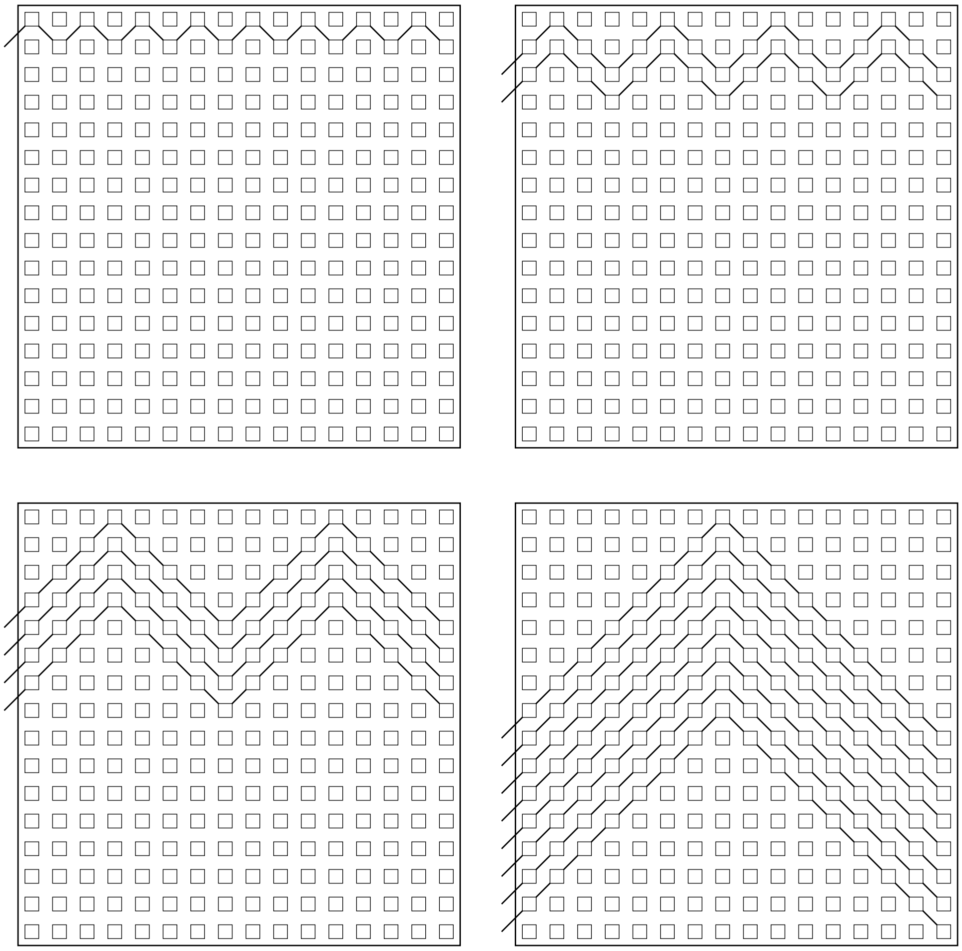


Fig. 6 A shiver shift that transports one layer horizontally to the left at the upper border of a (sub)grid.

Figure 6 explains the case of ninths whose length is a power of two, but in general the length of a ninth is 3^k times the length of a block. At least, we can assume that this length is even. Though shiver shifts on subgrids of arbitrary length tend to be complicated, there is a simple method to describe them: It suffices to cut a T-shaped piece out of the movement-pattern of the next bigger subgrid whose length is a power of two. Figure 7 shows how to obtain a shiver shift on a 12×12 grid from the pattern of a 16×16 grid.

An animated example of a simple shiver shift is in the corners of this report. To look at it leaf the pages quickly starting from the *last* page. You will see ten packets traveling to a ninth in the corner—five packets travel downwards, five packets to the right.

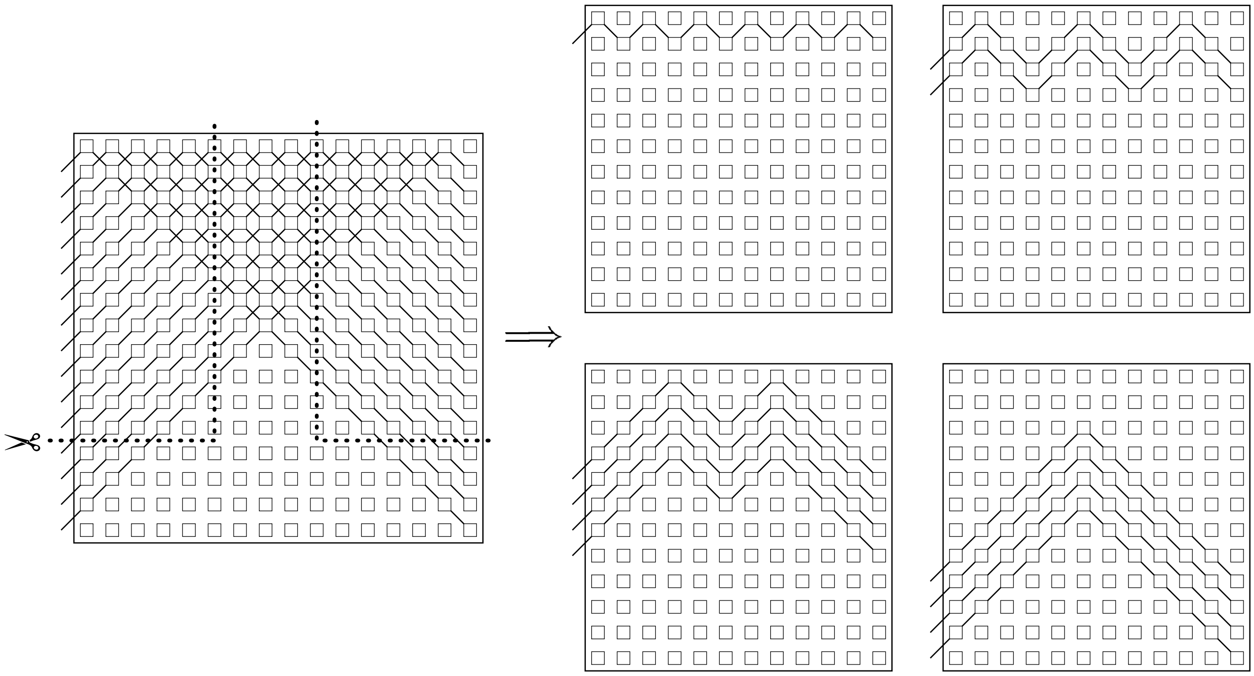


Fig. 7 How to cut an arbitrary sized shiver shift pattern out of a larger one.

Sorting for load nine: Proof of correctness

In this subsection we prove the correctness of generalized shiver shifts as defined above. That is, we show that in the described transport pattern of packets no conflicts occur. A conflict occurs when two or more packets want to travel through the same connection line between neighboring processors at the same time.

Our general plan of attack is as follows. First, we deal with the case of ninths whose side length is a power of two. Second, we describe the general situation where we have ninths with even side lengths, which we obtain by cutting out T-shaped pieces of the movement pattern of the first case.

We again focus on the shift from ninth B to ninth A (cf. Figure 5). Other transports between different ninths are either fully symmetric to the B to A case or are straightforward. We examine all possibilities that might lead to a conflict with a packet transported from B to A. To see that there are no conflicts between the transport from B to A and from B to E is trivial. So essentially six candidates for conflicts remain. We discuss all of them in the following.

Case B to A versus B to A:
We prove that two arbitrary packets travelling from B to A do not interfere with each other. A necessary condition for a conflict between two such elements is that they start in the same column. Clearly, they have to be simultaneously in the same processor some time. The latter is not possible if they belong to the same rectangle of transport. Belonging to the same rectangle of transport means that the packets have same period and amplitude. Since the packets all start at the same point of time, they never meet in a processor. Thus conflicts are impossible.

So let us assume that the two considered packets travel with different period length and amplitude. Both periods are powers of two. In the following we point out that *if* two packets meet in the same processor, *then* they will always leave this processor on different diagonals. Thus no conflicts may occur.

We refer to packets with greater amplitude and period length as *low packets* and to those with smaller amplitude and period length as *high packets*. Let the period length of low packets be 2^i and that of high packets be 2^j where $i > j$. By the *depth* of a packet we understand its distance from the upper border line of ninths A and B. Processors on the border line have depth 0. It suffices to investigate whether conflicts arise during one full period 2^i of low packets, as illustrated in Figure 8.

During which period(s) of time may low and high packets meet in the same processor? Since low packets have their starting point and their destination point in greater depth than high packets, it is clearly necessary that low packets have to come into areas where high packets travel. Low packets reach their tip (point of minimal depth) after 2^{i-1} steps, which is half a period length of low packets. Since 2^j, which is the period length of high packets, divides 2^{i-1}, it follows that then high packets have reached a pit (point of maximal depth). Thus the period of time where low and high packets meet is centered around 2^{i-1}. From time $2^{i-1} - 2^{j-1} + 1$ until 2^{i-1} high packets go southwest to their pit and during time $2^{i-1}+1$ until $2^{i-1}+2^{j-1}$ they go northwest to a tip. After $2^{i-1} - 2^{j-1}$ steps each high packet has reached a tip. Here the depth of the lowest high packet is $2^{j-1} - 1$. But after $2^{i-1} - 2^{j-1}$ steps the highest low packet has reached the height $2^{i-1} - (2^{i-1} - 2^{j-1}) = 2^{j-1}$. That is, until time $2^{i-1} - 2^{j-1}$ low and high packets do not meet in common processors. An analogous consideration yields the same fact for the period of time from $2^{i-1} + 2^{j-1} + 1$ until 2^i. But within the time interval from

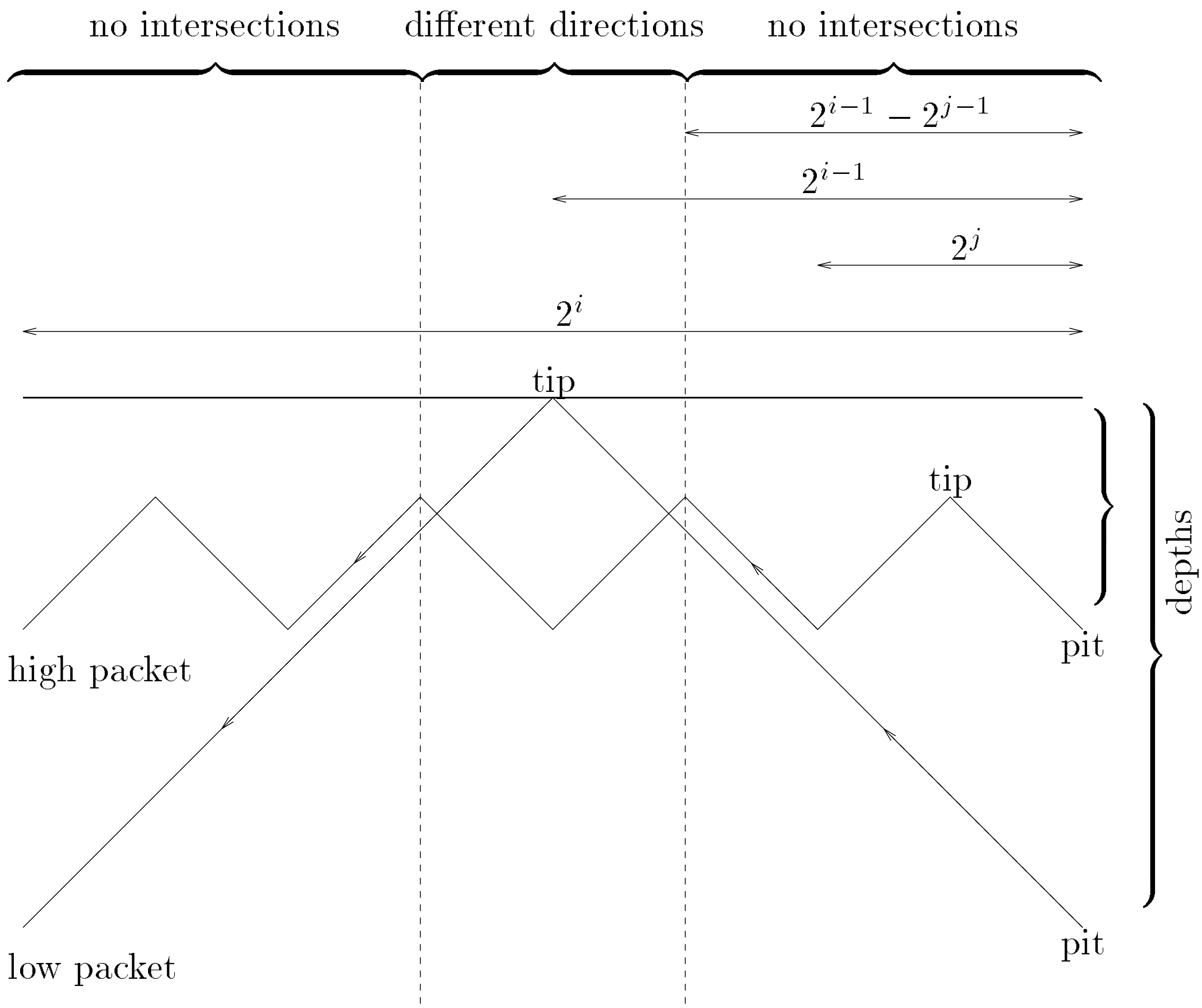


Fig. 8 Parts of the trajectories of low and high packets during a shiver shift.

$2^{i-1} - 2^{j-1} + 1$ until $2^{i-1} + 2^{j-1}$ low and high packets always move into different directions: In the first half low packets move northwest and high packets move southwest, in the second half low packets move southwest and high packets move northwest. Thus in no part of the time interval from 0 to 2^i conflicts arise and the claim follows.

Up to now the case B to A versus B to A dealt with ninths whose side length is a power of two. The correctness of the general case, where we only demand for ninths with even side lengths, is an easy consequence from this. Remember that we only cut out T-shaped pieces to obtain the movement pattern of the general case. But this means that for packets starting in the same column exactly the same piece is cut out. This yields the conflict-freeness, because the starting in the same column obviously still is a necessary condition for a conflict since in each step each packet moves to the west (either to the southwest or to the southeast).

Case B to A versus B to D:

Now we contrast the movements from B to A to those from B to D. There is an easy indirect proof of conflict-freeness that leads this case back to the previous one B to A versus B to A. Figure 9 displays a partial problem of the transports from B to A.

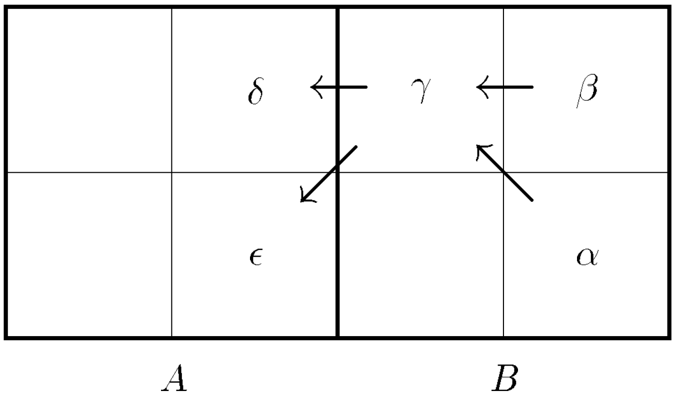


Fig. 9 Transport from B to A via γ.

Consider the five subsquares α, β, γ, δ, and ϵ of ninths A and B. In the transport from B to A the path of packets initially located in α first leads northwest to γ and then leads from γ southwest to the final destination ϵ. At the same time the packets from β travel to δ via γ. That means, after half of the time we have to transport one full layer of packets from γ to δ and one from γ to ϵ. But this is in miniature exactly the same situation we face when we transport one layer from B to A and one from B to D. In this way the proof of conflict-freeness directly follows from the B to A versus B to A case.

Here again the generalization to cases with ninths of even side lengths is easy. As in the previous case a necessary condition for a conflict between two packets is that they start in the same column. But then again in both the movements of the packets exactly the same T-piece is cut out and thus the conflict-freeness follows from the power of two case.

Case B to A versus E to A:

This case is dual to the previous one. Considerations analogous to there yield that the transports from α to γ and from β to γ (Figure 9) reflect in miniature the transports from B to A and from E to A.

The generalization to even side-lengths is fully analogous to the B to A case.

Case B to A versus D to A:
The candidate connection lines for simultaneous use by both transports are northwest diagonals. The prevention of conflicts results from the observation that the northwest diagonals either are exclusively used below (B to A) or are exclusively used beyond (D to A) the main diagonal (Figure 10). So the respective conflict candidates always use different transport lines.

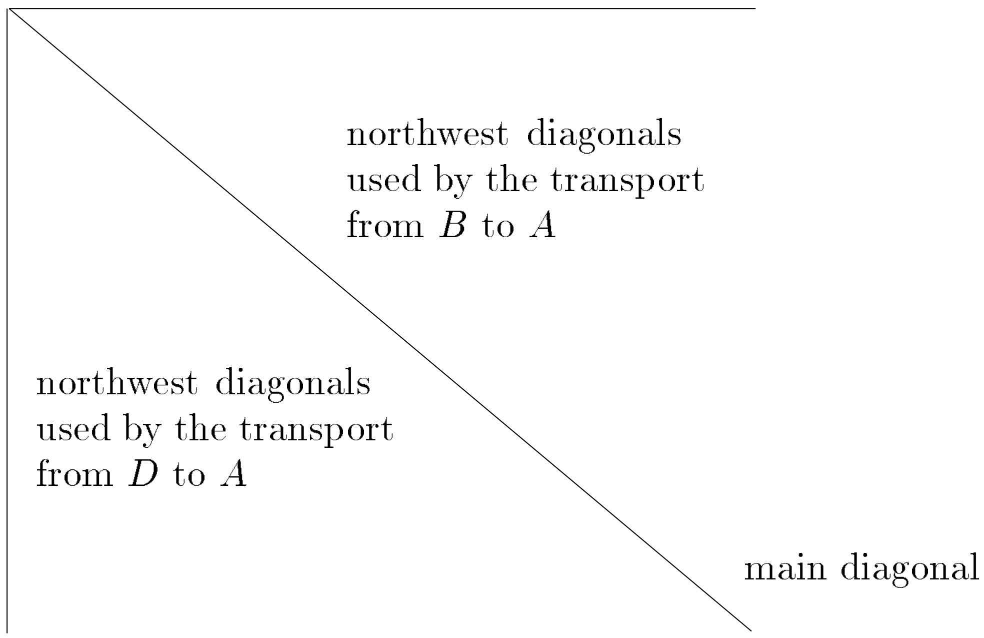


Fig. 10 B to A packets versus D to A ones.

Under the condition of ninths with even side-lengths instead of power of two side-lengths the argumentation is still valid. We omit the straightforward details.

Case A to B versus A to D:
This case works in full analogy to the previous one. The southeast diagonals now just play the part of the northwest ones.

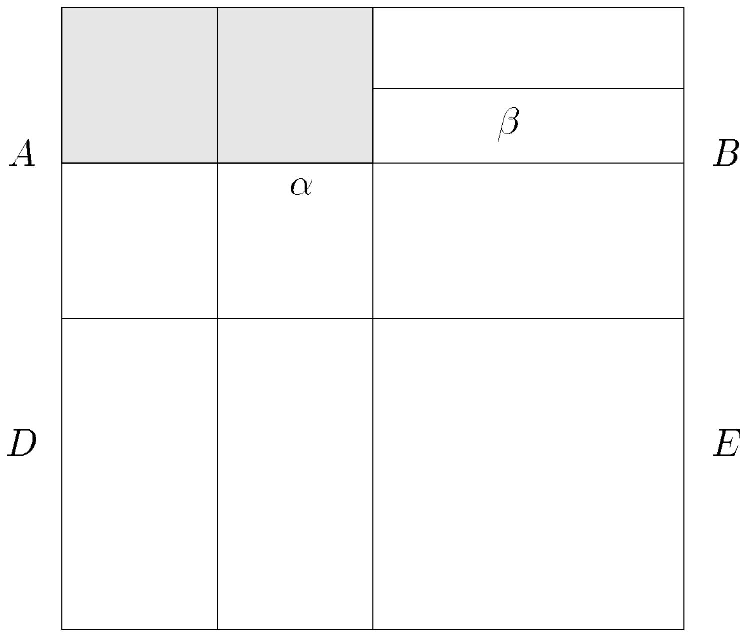


Fig. 11 Packets from B to A with smaller period than packets from A to D.

Case B to A versus A to D:
The only candidate connection lines for conflicts in this case are southwest diagonals. We start with excluding two simple subcases from further consideration.
1. The first subcase deals with the situation when the B to A packet has smaller period and amplitude than the A to D packet. We assume that the A to D packets start in a rectangle α and the B to A packets start in a rectangle β (Figure 11).

Only in the shaded area of Figure 11 conflicts are possible, because this is the intersection of the travel areas of packets starting in rectangles α and β. But all packets from α travel *within* the shaded area without interruption in western direction. This holds because the north-south expansion of the shaded area is at most half the period length of the packets starting in α. So packets starting in β never are able to catch up the packets from α in their western movement and, therefore, no conflicts occur.

The above argument in principle also generalizes to the case of ninths with even side-lengths. Again we use a "run away to the west" argument. We have to consider two situations. First, if the packets from α are those with maximal amplitude, then the first half of its movement still is a run away to the (south)west and the second half uses the uncritical southeast diagonals.

Second, in all other cases exactly the same argument as in the power of two case works.
2. The second easy to handle subcase is when the B to A and A to D packets have same period and amplitude (Figure 12).

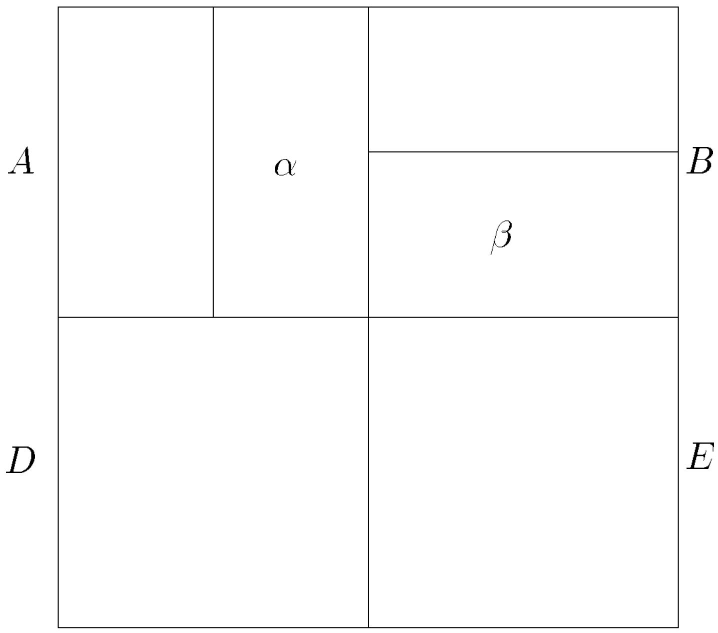


Fig. 12 Packets from B to A have the same period as packets from A to D.

If packets from rectangle α travelling from A to D move southwest, then packets from rectangle β travelling from B to A move northwest, and if packets from α move southeast, then packets from β move southwest. So southwest diagonal connections never are used simultaneously and no conflicts occur.

Clearly, the above reasoning directly transfers from the power of two to the general case without problems.

For the following we now assume that the packets transported from B to A have greater amplitude and period than the packets transported from A to D (Figure 13). Our aim is to show that when a packet from β starts moving southwest, then all packets that started in α are already southern to the packet from β.

What is the worst case that has to be considered? Clearly, the most disadvantageous situation with respect to A to D packets is when these packets start as northern as possible, because those packets form the northern border of the southward movement of A to D packets.

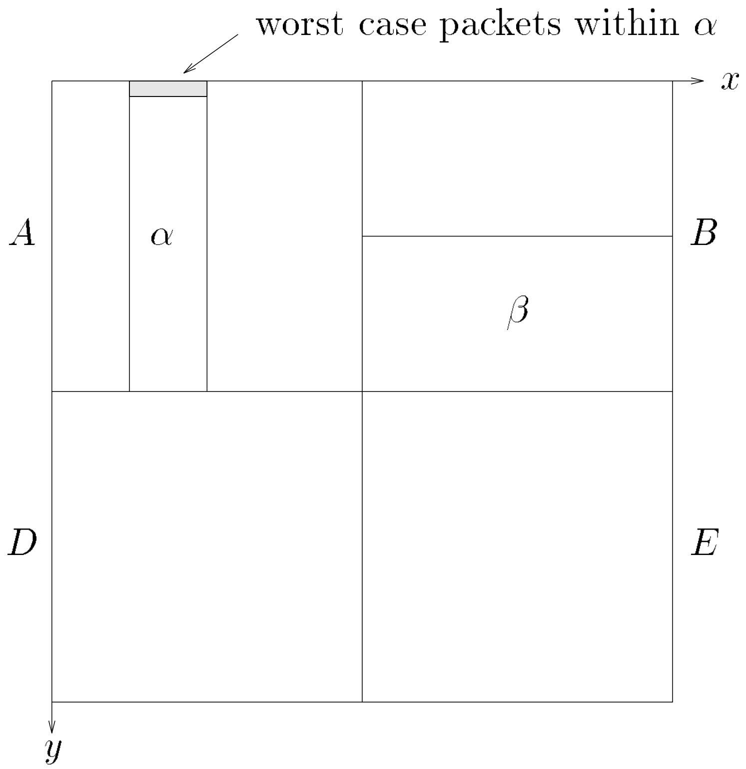


Fig. 13 Packets from B to A with longer period than packets from A to D.

We start with assuming that β is the biggest possible rectangle of transport in B (as we drew it in Figure 13). We need the following two fundamental observations. First, note that packets from α increment in each step their y-coordinate (Figure 13). That is, in each step, they either go southwest or southeast. Second, packets from β decrement their y-coordinate in each step of the first half of their movement. In particular, that means that packets from β, which use only northwest diagonals during the first half of their movement, then have a y-coordinate at least by one bigger than all worst case packets from α. So due to the first observation no conflicts can occur.

Now we consider the cases when β is not the biggest rectangle. Here the second of the above observations changes in the following way. The packets from β decrement their y-coordinate in each step of the first half of a packet's period length. But since the packets now also start at an appropriately smaller depth (that is, with smaller y-coordinate), the above reasoning in essence still remains valid.

To show the correctness of the general case, we can directly transfer the argumentation from the power of two case. This finishes the proof of correctness of generalized shiver shifts.

5. Sorting and routing for loads of one and two elements

The 4-4 and 9-9 all-to-all mappings are the most important algorithms in this paper. Besides the interest in their own, all other algorithms use them as subroutines. We are now going to present algorithms for h-h problems for $h < 4$.

Let us begin with the 1-1 routing and sorting problems. The 1-1 problem is special in the sense that we can solve the routing problem faster than the sorting problem. We handle first the routing.

Theorem 3. *The 1-1 routing problem requires at most* $10/9n + o(n)$ *steps on a mesh with diagonals.*

Proof. Each packet has a source address $(s_1, \ldots, s_m)$ and a target address $(d_1, \ldots, d_m)$ in ninth-coordinates. This address is not the address of the packet's block or brick, but the packet's own address. We start by routing each element to address $(s_1, d_2, s_3, \ldots, s_m)$ by a 2-shift, which adjusts the second component of each element's address. This changes the 1-1 problem into a partial 9-9 problem and needs $2/9n$ steps. Next, we route to address $(s_1, d_2, d_3, \ldots, d_m)$ by using the 9-9 sorting algorithm as an routing algorithm in all eighty-one $n/9 \times n/9$-submeshes. This takes $2/9n$ steps according to Theorem 2. Finally, we perform a 1-shift and route thus from $(s_1, d_2, \ldots, d_m)$ to $(d_1, d_2, \ldots, d_m)$ in $2/3\ n$ steps. Altogether the algorithm needs $10/9n$ steps. ∎

We need a little more time for 1-1 sorting:

Theorem 4. *The 1-1 sorting problem requires at most* $4/3n + o(n)$ *steps on a mesh with diagonals.*

Proof. Our general strategy is to start with a simple algorithm based on an all-to-all mapping and then to refine it in order to get a faster one.

One important ingredient is the algorithm for an all-to-all mapping of a single loaded mesh. Let us divide the mesh into nine equally sized quadratic submeshes called again *ninths*. As usual we assume there are m^2 blocks on the whole mesh and each block contains m^2 bricks. Thus each ninth contains $(m/3)^2$ blocks. We apply an all-to-all mapping on each ninth individually. Within each ninth each block receives *nine* bricks from each block, since there are nine times as many bricks in a block as blocks in a ninth. To complete the all-to-all mapping on the *whole* mesh, it suffices to interchange bricks between blocks in different ninths, but with the same relative position in the ninth. A fixed block in a ninth has to send $8/9m^2$ bricks to the eight other ninths and receives also $8/9m^2$ bricks from them. That completes the all-to-all mapping on the whole mesh, because in this way every block sent one brick to every block.

In order to interchange bricks between blocks of different ninths, we first bring together all nine blocks with the same relative position in the nine ninths by concentrating the border ninths in the central ninth according to Figure 14.

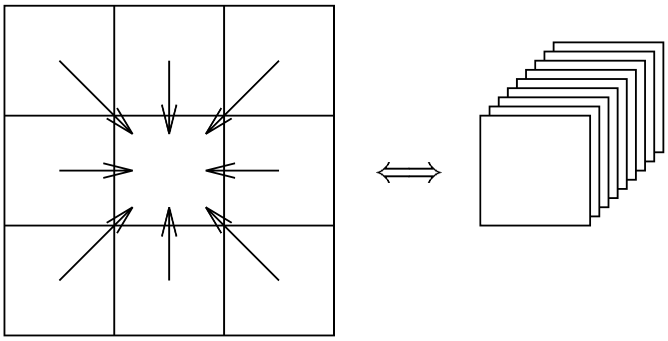


Fig. 14 Concentration and expansion.

After interchanging bricks between the nine layers we have to bring them back to where they belong by shifting eight ninths back to the border of the mesh. In this way the all-to-all mapping takes n steps, since the all-to-all mapping of single ninths takes $n/3$ steps, concentration also $n/3$ steps, and expansion another $n/3$ steps.

We now know we can sort in $2n + o(n)$ steps using the standard method by applying local sorting to blocks, then an all-to-all mapping, again local sorting on blocks, a second all-to-all mapping, and finally again local sorting on pairs of blocks (see Section 3).

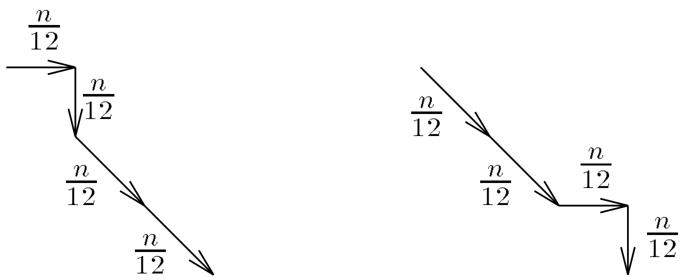
But we can improve this result by making use of the following observation. The second local sorting phase between the two all-to-all mappings may take place a little earlier: Since only single blocks are sorted and no transports between different blocks take place, we can sort them *before shifting the ninths back,* that is, before the all-to-all mapping is finished. Of course, the local sorting of blocks takes more time at a load of nine, but this doesn't hurt because blocks are so small. Please note that after this modification *nothing* happens between the two all-to-all mappings. There is an expansion of ninths followed instantly by a compression back to the central ninth. Obviously, we can leave away this expansion and compression without altering the outcome of the algorithm. This saves $2/3n$ steps. ∎

Theorem 5. *The 2-2 sorting problem requires at most* $5/3n + o(n)$ *steps on a mesh with diagonals.*

Proof. We use essentially the same trick as for the 1-1 sorting problem: The all-to-all mapping involved consists of a local all-to-all mapping in the four *quarters*, followed by a concentration in the middle of the grid that brings together the four quarters, and finally an expansion back into the corners. Again, applying two all-to-all mappings we can leave away the expansion of the first and the concentration of the second all-to-all mapping. The all-to-all mappings in the quarters take $n/2$ steps according to Theorem 1. All that is left to do is to shift the four quarters as fast as possible into the center of the mesh and back.

Each quarter consists of two layers. A straightforward approach transports one layer over diagonals and the second one over vertical and horizontal lines. This is not a very good idea, since one layer travels very fast and the second one two times slower. We can do better by *balancing* both transports, such that both layers take the same time. Let each level travel *two third* of the way on fast diagonals and one third on slow vertical and horizontal lines. In this way each layer uses the fast diagonals half of the time.

The upper left quarter has to travel $n/4$ steps to the southeast. According to Figure 15 (a), one layer goes first $n/12$ steps east, then $n/12$ steps south, and finally $n/6$ steps southeast.



(a) (b)

Fig. 15 Interleaved diagonal transports.

The second layer travels first southeast, then east, and at the end south (see Figure 15 (b)). The used lines are nearly disjoint, only parts of the diagonals are used by packets from both layers. This is not harmful, because both layers use those diagonals at different times. By now we proved that both layers of the upper left quarter can travel to the center in $n/3$ steps without collisions. We still have to specify how the remaining three quarters move and to be sure that there are no lines used simultaneously by packets from different quarters. Table 4 shows in detail how each layer of the four quarters moves.

◰		◳		◱		◲	
layer		layer		layer		layer	
one	two	one	two	one	two	one	two
$\rightarrow$	$\searrow$	$\downarrow$	$\swarrow$	$\uparrow$	$\nearrow$	$\leftarrow$	$\nwarrow$
$\downarrow$	$\searrow$	$\leftarrow$	$\swarrow$	$\rightarrow$	$\nearrow$	$\uparrow$	$\nwarrow$
$\searrow$	$\rightarrow$	$\swarrow$	$\downarrow$	$\nearrow$	$\uparrow$	$\nwarrow$	$\leftarrow$
$\searrow$	$\downarrow$	$\swarrow$	$\leftarrow$	$\nearrow$	$\rightarrow$	$\nwarrow$	$\uparrow$

Table 4 Conflict-free routes to the center.

There are four rows showing the travel directions in the first, second, third, and fourth $n/12$ steps. The columns of the table indicate that each layer indeed reaches the correct position, that is, the center of the mesh. The rows show that at a time only one packet uses each line. ∎

6. Sorting and routing on tori with diagonals

At first glance the *recursive* algorithms we obtained for grids are not suitable for tori. We can divide a grid in subgrids, but cannot divide a torus in subtori. A look at the *nonrecursive* version of the algorithms, however, leads us out of this dilemma. The movement of a brick from source address $(s_1, \ldots, s_k)$ to destination address $(d_1, \ldots, d_k)$ was achieved by "adjusting" the k components of the quad address one after the other. To change the kth component from ◳ to ◰, Algorithm F uses a shift to the left. As long as the source block is not at the right border, a shift to the *right* also changes the kth component from ◳ to ◰. It changes some other components of the quad coordinate, too; but this causes no harm, since only smaller components are affected. Starting with the kth component and working towards the first is always safe.

The only problem are the borders. If there are no borders, we can reverse all arrows in Table 2 without causing any trouble. A torus has no borders, so we can apply this idea. By using the reverse directions as well as the regular ones, we can transport the double amount of data during an all-to-all mapping. In this way we can solve the 8-8 sorting problem in $2n$ steps on a torus. This is not yet good enough, just as Algorithm F is inferior to Algorithm N.

We cannot modify Algorithm N in the same way. In contrast to Algorithm F the "reverse directions" are almost all occupied. Yet there is a simple solution to this problem: Table 3 used for Algorithm N is very complicated; to redesign it for tori let us first build a *naïve* table for subtraction of ninth-coordinates. All paths should be as simple as possible, no detours and no delays. So we replace ↲ by ↙ and •→ by →. Table 3 avoids collisions by hook or by crook, but we want neither hooks, nor crooks, nor delays in our naïve table. Changing all entries of Table 3 accordingly leads us to Table 5 (a).

(a)

−	⌜	⊏	⌞	⊓	⊡	⊔	⌝	⊐	⌟
⌜	·	↑	↑↑	←	↖	↖↑	←←	↖←	↖↖
⊏	↓	·	↑	↙	←	↖	←↙	←←	←↖
⌞	↓↓	↓	·	↙↓	↙	←	↙↙	↙←	←←
⊓	→	↗	↑↗	·	↑	↑↑	←	↖	↑↖
⊡	↘	→	↗	↓	·	↑	↙	←	↖
⊔	↓↘	↘	→	↓↓	↓	·	↓↙	↙	←
⌝	→→	↗→	↗↗	→	↗	↗↑	·	↑	↑↑
⊐	→↘	→→	→↗	↘	→	↗	↓	·	↑
⌟	↘↘	↘→	→→	↘↓	↘	→	↓↓	↓	·

(b)

−	⌜	⊏	⌞	⊓	⊡	⊔	⌝	⊐	⌟
⌜	·	↑	↓	←	↖	↙	→	↗	↘
⊏	↓	·	↑	↙	←	↖	↘	→	↗
⌞	↑	↓	·	↖	↙	←	↗	↘	→
⊓	→	↗	↘	·	↑	↓	←	↖	↙
⊡	↘	→	↗	↓	·	↑	↙	←	↖
⊔	↗	↘	→	↑	↓	·	↖	↙	←
⌝	←	↖	↙	→	↗	↘	·	↑	↓
⊐	↙	←	↖	↘	→	↗	↓	·	↑
⌟	↖	↙	←	↗	↘	→	↑	↓	·

Table 5 Computing the path between two ninth-coordinates for a torus.

Instead of building the dual table to Table 5 (a) by reversing all directions, let us only *replace* some directions by their reverse counterparts. We have to do this in such a way that no collisions will occur after doing this replacement. The resulting table is the only one used in our algorithm. There is another goal to meet: We have to gain something. Since we will not transport *more* data, we have to get *faster*. We can meet both goals by replacing all *long* arrows by their reverse counterpart. Since the reverse direction of a long shift is a short one, only short shifts will survive. Thus we need only half of the time as before. The resulting Table 5 (b) also looks quite simple and there are no conflicts, because from each ninth we reach each of the other eight ninths by a shift in one of the eight directions. The resulting Algorithm T looks as follows. It subtracts ninth-coordinates according to Table 5 (b). The running time is $n/2$.

Algorithm T. (All-to-all mapping for load nine on a torus)
for $i = k$ **downto** 1 **do**
  for all $(s_1, \ldots, s_k)$, $(d_1, \ldots, d_k)$ **do in parallel**
    Shift the brick with source address $(s_1, \ldots, s_k)$ *and destination address* $(d_1, \ldots, d_k)$ *in direction* $d_i - c_i$ *for* $n/3^i$ *steps, where* $(c_1, \ldots, c_k)$ *is the brick's current address.*

In the ith iteration the ith component of a brick's address changes to d_i, while the $(i + 1)$st to kth components remain unchanged. The first $i - 1$ components, however, may change, too. That is the reason why we can't change the ith component by a shift in direction $d_i - s_i$. We change by a shift in direction $d_i - c_i$, where c_i is the brick's ith component after $i - 1$ iterations.

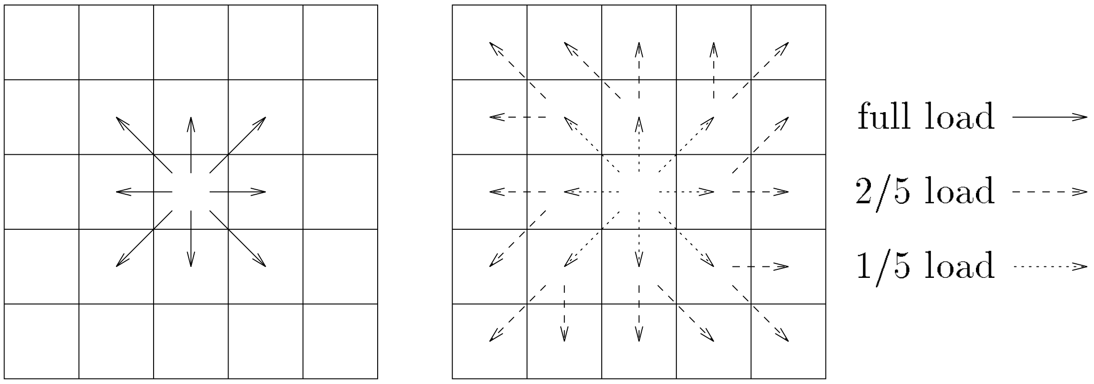
We can solve the 9-9 sorting problem twice as fast as on a grid. While the 9-9 problem is the biggest we can solve on a grid in $2n$ steps, we can improve on the load of nine on a torus keeping the time bound n.

Theorem 6. *The 10-10 sorting problem requires at most* $n + o(n)$ *steps on a torus with diagonals.*

Proof. Instead of a subdivision into nine submeshes we rather use *twentyfive* submeshes. If we transport one layer to each "twentyfifth," we perform an all-to-all mapping for a 25-25 problem. Unfortunately, it is hard to do this fast. We are humble and stick to simple routing scheme and see how much data we can transport within the distance bound. First, in $n/5$ steps we move eight layers in all eight directions (Figure 16 (a)).

Second, in another $n/5$ steps we move $2/5$ of one of the inner layers into each outer layer according to Figure 16 (b). At the same time each inner twentyfifth receives another $1/5$ layer from the central twentyfifth. That fills it out to a $2/5$ layer.

Up to now all algorithms we considered moved *full* layers. At this point we start to consider shifting *partial* layers. Shifting a half layer, for example, means to transport one packet from every second processor. To perform an all-to-all mapping for a 10-10 problem means that every twentyfifth has to send a $2/5$ layer to each other twentyfifth.



(a) (b)

Fig. 16 Standard routing scheme for all-to-all mappings on tori.

■

7. Conclusion

Meshes with diagonals are simple, scalable parallel architectures. Compared to meshes with four-neighborhood the distance bound is two times smaller and the bisection bound is three times smaller. We developed algorithms with small buffer size for sorting and routing on grids and tori with diagonals that come very close to the lower bounds.The algorithmic methods are generalizable to higher dimensional grids.

Some open questions remain. Is there an 1-1 routing algorithm that needs only n steps, matching the distance bound? Is there an 12-12 sorting algorithm that needs only $2n$ steps? More general, is there an sorting algorithm that matches the bisection bound for arbitrary load h?

8. Bibliography

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

[2] M. Kaufmann, S. Rajasekaran, and J. F. Sibeyn. Matching the bisection bound for routing and sorting on the mesh. In *Proceedings of the 3d ACM Symposium on Parallel Algorithms and Architectures*, pages 31–40, 1992.

[3] M. Kaufmann and J. F. Sibeyn. Optimal k-k sorting on meshes and tori. 1993.

[4] M. S. Kaufmann and J. Sibeyn. Optimal multi-packet routing on the torus. In *Proc. of the 3d Scandinavian Workshop on Algorithm Theory*, pages 118–129, 1992.

[5] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[6] M. Kunde. Concentrated regular data streans 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.

[7] 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.

[8] 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.

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

[10] T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 77–96, 1992.

[11] 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.

[12] 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.

[13] A. Park and K. Balasubramanian. Reducing communication costs for sorting on mesh-connected and linearly connected parallel computers. *Journal of Parallel and Distributed Computing*, 9:318–322, 1990.

[14] S. Rajasekaran and R. Overholt. Constant queue routing on a mesh. In C. Choffrut and M. Jantzen, editors, *Proceedings of the 8th Symposium on Theoretical Aspects of Computer Science*, number 480 in Lecture

Notes in Computer Science, pages 444–455, Hamburg, Federal Republic of Germany, February 1991. Springer-Verlag.

[15] 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.

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

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