Faster Sorting and Routing on Grids with Diagonals

Manfred Kunde Rolf Niedermeier[1] Peter Rossmanith[1]

Fakultät für Informatik, Technische Universität München,
Arcisstr. 21, 80290 München, Fed. Rep. of Germany
kunde/niedermr/rossmani@informatik.tu-muenchen.de

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 the on 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], 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 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.

A two-dimensional *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 $\max\{|r - r'|, |c - c'|\} = 1$. 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.

[1] Work supported by the Deutsche Forschungsgemeinschaft, SFB 342, A4 "KLARA."

For an h-h *routing problem* each processor contains at most h packets initially. Each packet has a destination address specifying the processor to which it must travel. Each processor is destination of at most h packets. The routing problem is to transport each packet to its destination address. For the sorting problem we need some more details. 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 $\{ (P, j) \mid P \text{ is a processor } \}$ 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 $\{0, \ldots, hn^2 - 1\}$. Then 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 where a processor contains exactly h packets initially 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], 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.

For two-dimensional $n \times n$ meshes without diagonals 1-1 problems have been studied for more than twenty years. The so far fastest solutions for 1-1 problems and for h-h problems with small $h \leq 9$ are summarized in Table 1. In that table we also present our new results on grids with diagonals and compare them with those for grids without diagonals. The table contains results obtained by deterministic as well as by randomized algorithms that work with high probability (the latter ones are marked with an asterisk). We omit all sublinear terms, which are of no importance for the asymptotic complexity.

On meshes with diagonals for h-h problems we have $hn/2$ steps as a simple lower bound, the bisection bound. Recently this lower bound was asymptotically reached by randomized algorithms [2, 3] as well as by deterministic algorithms [7]. These results are optimal 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 $h > 9$.) Note that the bisection bound for meshes with diagonals is $hn/6$.

For wrap-around meshes (or tori) without diagonals the respective bisection bound is $hn/4$, which was also matched by algorithms recently: with high probability [3] and deterministically [7]. If 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.

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

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 algorithm

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 that is 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.

Due to the lack of space several details and proofs had to be omitted. A full paper is available from the authors.

2 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. 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 (see for example [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 Fig. 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].

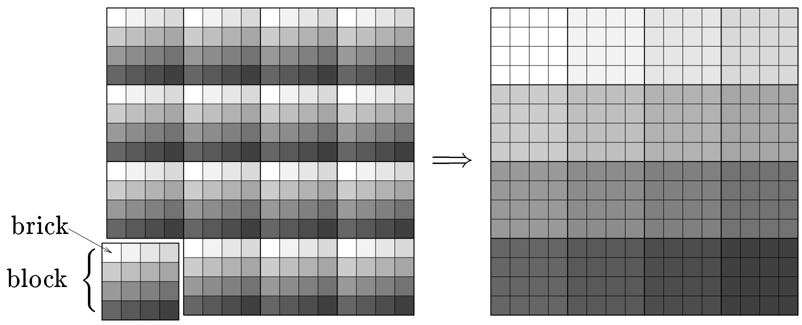


Fig. 1. The standard all-to-all mapping on a mesh with 16 blocks

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 any given continuous block indexing.

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. For partial h-h routing, say with a load of 75 percent, the above method also works, provided some additional tricks are used (see Kunde [7]). Algorithms for partial routing are used as subprocedures for algorithms in this paper. See the full paper for a detailed discussion.

3 Sorting for Loads of Four and Nine Elements

In this section we start with an algorithm for the 4-4 sorting problem to present the basic ideas.

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, called 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 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. □

Before we attack the more complicated 9-9 problem, let us look at the 4-4 sorting 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 $(x, y) \in \{1, \ldots, m\} \times \{1, \ldots, m\}$, where x denotes the column and y the row. 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. We use a *quaternary* number system to address blocks using the four symbols ◲, ◱, ◳, and ◰ as digits. An example reveals best how to apply this number system.

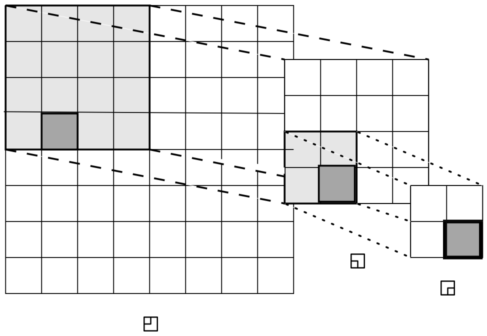


Fig. 2. The quad coordinates are (◰,◱,◲)

In Fig. 2 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?

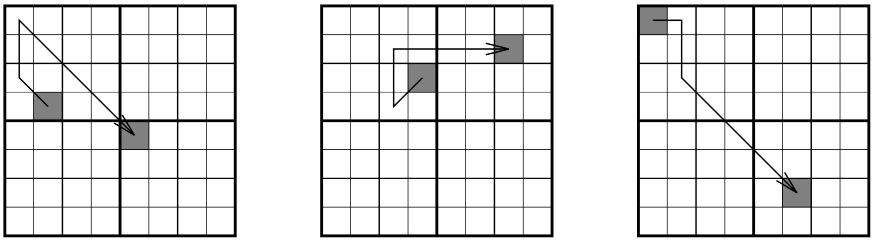


Fig. 3. Three examples of paths on which bricks travel to their destination

Figure 3 contains 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.

Table 2. Computing the path between two quad coordinates.

−	◲	◳	◱	◰
◲	·	↓	→	↘
◳	↑	·	↗	→
◱	←	↙	·	↓
◰	↖	←	↑	·

We compute the paths in Fig. 3 as (◰, ◱, ◲) − (◲, ◰, ◰) = (↘, ↑, ↖), (◰, ◲, ◳) − (◳, ◳, ◱) = (→, ↑, ↗), and (◰, ◰, ◰) − (◲, ◱, ◳) = (↘, ↓, →).

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.*

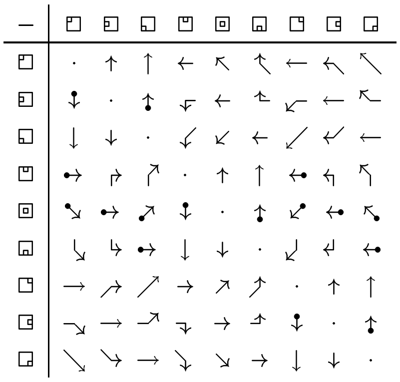
By an *i-shift* we understand to route all bricks in direction $d_i - s_i$, subtracting according to Table 2. The distance of the transport is 2^{n-i} for a $2^n \times 2^n$ grid. An i-shift changes exactly the ith component of a quad coordinate from source to destination. We realize an all-to-all mapping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

The ideas of the above discussion will now be used to solve the 9-9 problem. After Theorem 2 we will sketch the main algorithmic ideas.

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

Divide the grid into *nine* subgrids, called *ninths*. Transport one layer from each ninth to each ninth in order to mix data among them. Then complete the all-to-all mapping by applying this procedure recursively on each ninth, just like Algorithm F did. We can describe this method by *ninth-coordinates* (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* $(d_1, \ldots, d_k)$. 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 (·, ↑, ↗, →, ↘, ↓, ↙, ←, ↖) from Algorithm F, Algorithm N uses some more directions, directions that *change* in exactly the middle of the journey: ↱, ↙•, ←·, and so on. Direction ↱, for example, means "travel the first half of the time upwards and the second half of the time to the northeast." There is also ↙•, which means "stay were you are during the first half of time and travel then to the southwest" and ←·, 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.

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 $d_i - s_i$ 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.

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 Fig. 4. We obtain the same figure also for the second $n/3^i$ steps.

You can read Fig. 4 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

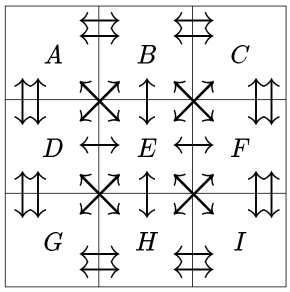


Fig. 4. All shifts at once

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 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 first 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 Fig. 5.

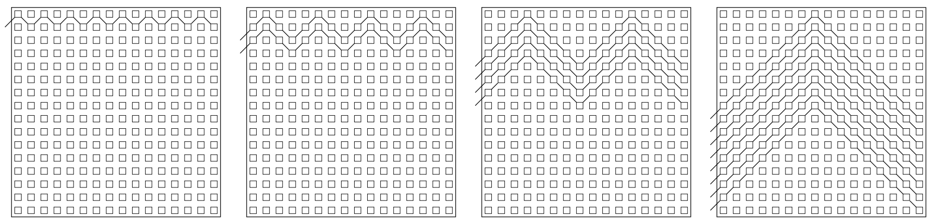


Fig. 5. A generalized shiver shift that transports one layer horizontally to the left at the upper border of a (sub)grid

The zig-zag movement has different amplitude and period-length for different packets. In Fig. 5 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.

Figure 5 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. See the full paper for details and the proof of correctness.

4 Sorting and Routing for Loads of One and Two Elements

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, in ninth-coordinates, a source address $(s_1, \ldots, s_m)$ and a target address $(d_1, \ldots, d_m)$. These addresses consist of processor coordinates, i.e., are on the lowest level. 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. □

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

Proof. We divide the mesh into nine squares and move the outer squares into the middle. Then we sort the middle square in $2/3n$ steps (Theorem 2) and move eight squares back to their outer positions. □

There is another algorithm for the 1-1 sorting problem that does not build upon the 9-9 sorting of Theorem 2. It has the same time complexity as above and can be found in the full paper.

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, but we divide the mesh into only four squares. All that is left to do is to shift the four quarters as fast as possible into the center of the mesh and back.

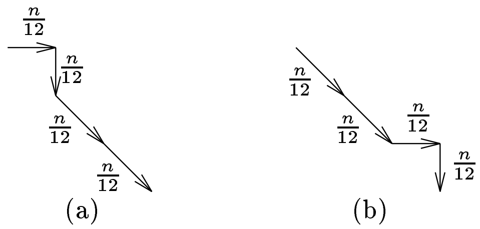


Fig. 6. Interleaved diagonal transports

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

The second layer travels first southeast, then east, and at the end south (see Fig. 6 (b)). The other squares behave symmetrically. □

5 Sorting and Routing on Tori with Diagonals

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 as in Theorem 2, 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 (Fig. 7 (a)).

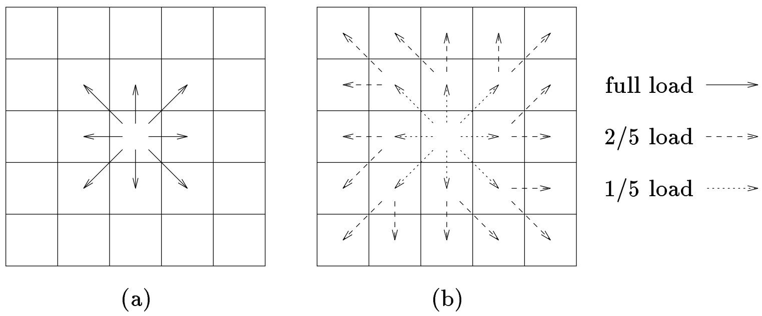


Fig. 7. Standard routing scheme for all-to-all mappings on tori

Second, in another $n/5$ steps we move $2/5$ of one of the inner layers into each outer layer according to Fig. 7 (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. The situation is symmetric for all twentyfives, because a torus has no "center." □

6 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 constant buffer size for sorting and routing on grids and tori with diagonals that come close to the lower bounds. The algorithmic methods are generalizable to higher dimensional grids.

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

References

1. C. Kaklamanis and D. Krizanc. Optimal sorting on mesh-connected processor arrays. In *Proc. of 3d SPAA*, 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 *Proc. of 3d SPAA*, 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 *Proc. of 32d FOCS*, 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, *Proc. of 1st ESA*, number 726 in Lecture Notes in Computer Science, pages 272–283, Bad Honnef, F.R.G., September 1993. Springer.
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 *Proc. of 24th STOC*, 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 *Proc. of 1st SPAA*, 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 *Proc. of 27th FOCS*, 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, *Proc. of 8th STACS*, number 480 in Lecture Notes in Computer Science, pages 444–455, Hamburg, F.R.G., February 1991. Springer.
15. C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. In *Proc. of 18th STOC*, 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.

This article was processed by the author using the TEX macro package from Springer-Verlag.