Matrix Transpose on Meshes: Theory and Practice

Michael Kaufmann
Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen, Sand 13,
72076 Tübingen, Germany
mk@informatik.uni-tuebingen.de

Ulrich Meyer and Jop F. Sibeyn
Max-Planck-Institut für Informatik,
Im Stadtwald,
66123 Saarbrücken, Germany
{umeyer, jopsi}@mpi-sb.mpg.de

Abstract

Matrix transpose is a fundamental communication operation which is not dealt with optimally by general purpose routing schemes. For two dimensional meshes, the first optimal routing schedule is given. The strategy is simple enough to be implemented, but details of the available hardware are not favorable. However, alternative algorithms, designed along the same lines, give an improvement on the Intel Paragon.

1 Introduction

Various models for parallel machines have been considered. Despite of their large diameter, *meshes* are of great importance because of their simple structure and efficient layout. In a d-dimensional mesh, the processing units, *PUs*, form an array of size $n \times \cdots \times n$ and are connected by a d-dimensional grid of communication links. A number of parallel computers with two-dimensional and three-dimensional mesh topology has been built, for instance the Intel Paragon, the CRAY T3E, and the J-Machine of MIT.

Routing. In a *routing problem* packets must be sent to known destinations such that at most one packet passes through any wire during a single step. The quality of a routing algorithm is mainly determined by its *run time* T, the maximum number of steps a packet may need to reach its destination. Several variants of the problem have been studied. In 1-1 *routing*, each PU initially sends and receives a single packet. In k-k *routing*, each PU is the source and destination of k packets.

Models. Our model is a simplification of the store-and-forward model, in which in each step one packet can be routed over each connection. To obtain the simplest possible exposition, we concentrate on the maximum number of packets that has to go in one direction over any connection. Under light conditions this simplification is correct. Considering only the load of the connections, comes close to the wormhole and virtual cut-through model. In these models, which describe the routing in modern parallel computers, the time for a routing step is the sum of a start-up time, data-transfer time, and time for additional hops. If the size of the packets is large enough, the data-transfer time dominates, and we find back our model.

Matrix Transposes. Next to general routing, several permutations have been analyzed in more detail [6, 9, 10, 1, 7, 2]. The considered permutations are for example shuffles and unshuffles, transposes [2], bit-permute complements [6] and affine permutations [9, 1]. The reason is that these permutations are very important in applications like linear algebra systems, and often (not always, see [1]) general-purpose routers perform not good enough for them.

Among the mentioned permutations, the *transposes* are of outstanding importance. They are frequently used in a variety of techniques for the solution of linear equations or partial differential equations in linear algebra. We might even call the transpose problem a benchmark to evaluate efficient permutation routing schemes.

On a two-dimensional mesh a transpose is the permutation under which the packets from PU (x, y) has to be routed to PU (y, x). Here PU (x, y) designates the PU with x-coordinate x and y-coordinate y. For d-dimensional meshes, natural generalizations are the permutations mapping the packets from PU $(x_0, x_1, \ldots, x_{d-1})$ to PU $(x_i, x_{1+i}, \ldots, x_{d-1+i})$, where the numbers in the subscripts are computed modulo d. We call these permutations *transposes* again. There are d transposes, Tr_i, $0 \leq i \leq d-1$, with $Tr_0 = Id$, the identity.

A *uni-axial* algorithm, is an algorithm in which in any given step communication is performed only along connections of a single axis. Transposes are also important because of

Property 1 *When one version of a uni-axial algorithm operates along axis* $(j + i) \bmod d$, *whenever another version operates along axis* j, *then their results can be recombined by performing* Tr_i.

By *recombining* we mean that the corresponding results are brought together. On a d-dimensional mesh we want to perform d uni-axial algorithms at the same time, each operating on a fraction $1/d$ of the packets. So there we wish to perform a permutation $STr = \sum_i Tr_i/d$, under which a fraction $1/d$ is sent with each of the Tr_i. This latter operation is very balanced, and can be performed in a regular way.

The most recent algorithms for matrix transpose on meshes were presented by Ding, Ho and Tsay [2]. Only two-dimensional meshes are considered, and their best algorithm is away from optimal by a factor 1.05.

Overview of this Paper. In this paper we consider various aspects of the matrix transpose problem.

We improve the result of [2] to optimality for, and present novel ideas to perform transposes on higher-dimensional meshes. Our lower bounds show that the transpose operation is not as cheap as expected for higher-dimensional meshes.

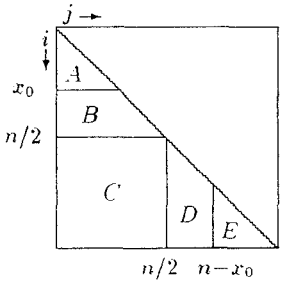
Then in the practical part, we report on our implementations on the Paragon. We present several approaches and analyze the conditions for the best performance of each approach. Even for a small number of processors, our algorithms work very well.

In the third part, we turn to theory again. We illustrate how Property 1 can be applied to the sorting problem on meshes. For two-dimensional meshes, k-k sorting is now performed in less than $k \cdot n$ steps, almost twice as fast as before. Due to a lack of space, we only give results. Details missing in this paper can be found in the full version [4].

2 Transposes on 2-D Meshes

Transposes on two-dimensional meshes have been considered before by Ding, Ho and Tsay [2]. For sufficiently large n they prove a lower bound of $(1 - 1/\sqrt{2}) \cdot k \cdot n \simeq 0.293 \cdot k \cdot n$. Their best algorithm requires $0.306 \cdot k \cdot n$ steps. In this section we show a simple deterministic algorithm which performs only slightly worse: $0.301 \cdot k \cdot n + \mathcal{O}(n/k)$. Then we show how this algorithm can be made optimal.

A basic Scheme. We now describe a simple deterministic algorithm, performing the standard transpose in nearly optimal time. All packets are routed horizontally and then vertically (XY-routing), or vice versa (YX-routing). So, the problem amounts to deciding for each packet in which direction it is routed first: the packets are **colored** white (moving vertically first) or black (moving horizontally first).



$$A : p_h(i,j) = 1$$
$$B : p_h(i,j) = x_0/i$$
$$C : p_h(i,j) = 1/2$$
$$D : p_h(i,j) = 1 - \frac{x_0}{n-j}$$
$$E : p_h(i,j) = 0$$

Figure 1. Subdivisions for the standard transpose. p_h denotes the fraction of the packets per PU, that go horizontally first.

In Figure 1 we give the routing scheme. Because of the symmetry we only consider the packets starting below the diagonal. The other packets are treated analogously. A PU with index (i,j) sends $p_h(i,j) \cdot k$ packets horizontally first, the rest starts to move vertically.

Lemma 1 *The PUs can determine on-line how to color the packets, such that on a two-dimensional $n \times n$ mesh a k-k transpose can be routed in $0.301 \cdot k \cdot n + \mathcal{O}(n/k)$ steps.*

Proof: We have to check that no link transfers more than $T(n,k)$ packets in any direction. We can confine our analysis to the boundary of $B \cup C \cup D$, which is just the critical area constructed in the lower bound proof. Take $x_0 = (\sqrt{2}-1)/2 \cdot n \simeq 0.293 \cdot n$. A horizontal boundary link of B is traversed rightwards by exactly $x_0/i \cdot k \cdot i = x_0 \cdot k$ packets. The right border of $C \cup D$ has the biggest throughput via the lowest x_0 links. Each of them has to transfer $t = k \cdot \sum_{j=0}^{n-x_0} p_h(i,j)$ packets. Applying the definition of $p_h(i,j)$, we get

$$t = k \cdot (n/2 \cdot 1/2 + \sum_{j=n/2}^{n-x_0} (1 - x_0/j))$$
$$\leq k \cdot (3/4 \cdot n - x_0 - x_0 \cdot \int_{x_0}^{n/2} 1/j \, dj) \simeq 0.301 \cdot k \cdot n.$$

The vertical border links are checked in the same manner.

Our scheme ensures that the fraction of white packets per PU either does not change within the PU-rows or it remains fixed within the PU-columns. This allows us to compensate for rounding errors (induced by bad divisibility of k) in one row or column by an appropriate rounding offset in the neighboring rows or columns. The additional link loads can be limited to $\mathcal{O}(1)$. See Figure 2 for a coloring example within the B-area.

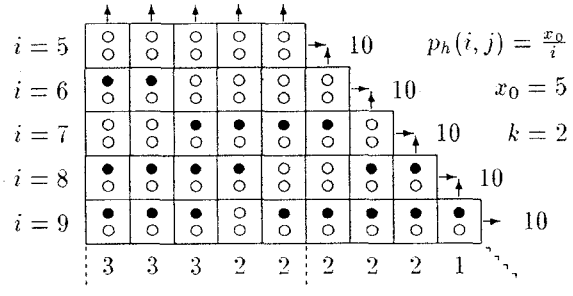


Figure 2. Linked coloring: the idealized fraction of white packets per PU is equal for all PUs within a row. The positions of black packets in row $i+1$ are aligned to those in row i.

Because of the way the packets are colored the links with the biggest throughputs at both ends of the boundary line do not immediately start to transfer packets continuously. Consider e.g. the black packets in Figure 2. An upper bound for the extra time follows from the distance between the border line and the first row / column, in which each PU has at least one packet of the desired color: $x_0/(k-1)$. So, we get $\mathcal{O}(n/k)$ extra time. □

Optimizing the Algorithm. Starting with the basic solution, we now show how to modify the coloring such that the throughput over the border links of $B \cup C \cup D$ becomes balanced. Figure 3 (bottom) shows the areas H and V in which the colors of some packets are flipped.

We take a closer look at the modification in the V-region. It is important that the surpluses in the rows and the slacks in

L_j	80	64	50	40	20	5		
l_j	16	14	10	20	15	5		
	0	0	0	0	1	5	6	6
	0	0	0	0	1	0	1	7
	0	0	5	20	13	0	38	45
	0	0	2	0	0	0	2	47
	0	0	1	0	0	0	1	28
	17	13	2	0	0	0	32	80
							m_i	M_i

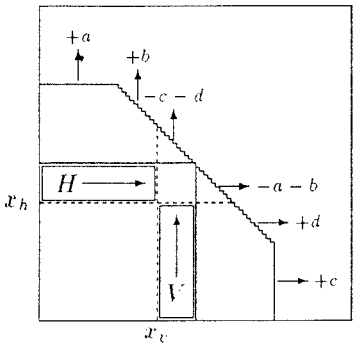


Figure 3. Bottom: Color modification scheme: the surpluses $+a$ and $+b$ are eliminated by coloring more packets white in H. $+c$ and $+d$ are compensated by additional black packets starting from V. Top: Example for an on-line computation of the number of additional black packets per PU. Here the $c_{i,j}$ values in the 6×6 square, are computed with help of (1). E.g., $13 = 20 - 7$, $20 = 40 - 20$, $5 = 45 - 40$ and $2 = 47 - 45$.

the columns coincide in the V-region, such that the optimization can be achieved by local changes. Let l_j denote the free capacity in column j, and let $L_j = \sum_{t=j}^{n/2} l_t$. Analogously, let m_i be the surplus in row i, and $M_i = \sum_{t=x_h}^{i} m_t$, where x_h is the index of the row in which there is neither slack nor surplus. Note that l_j and m_i can be directly computed from the coloring rule, and that an accurate estimate of L_j and M_i can be computed in $\mathcal{O}(1)$ time by integration. The redirection in V is done according to the following rule: in PU (i, j) the number $c_{i,j}$ of white packets that is colored black is given by

$$c_{i,j} = \max\{\ 0, \min\{M_i - L_{j+1}, M_i - M_{i-1}, L_j - L_{j+1}, L_j - M_{i-1}\}\ \}. \qquad (1)$$

So, all PUs in V can perform the recoloring independently in $\mathcal{O}(1)$ time. Figure 3 (right) gives an example of the recoloring. Each of the terms in (1) has a specific interpretation: on the line of PUs with $c_{i,j} > 0$, $M_i - L_{j+1}$ is minimal in the left corners, $M_i - M_{i-1}$ on the vertical and $L_j - L_{j+1}$ on the horizontal legs and $L_j - M_{i-1}$ in the right corners.

Hereby, we have shown, informally, how the throughput over the border links of $B \cup C \cup D$ can be balanced, such that no outgoing connection has to transfer more than $(1 - 1/\sqrt{2}) \cdot k \cdot n$ packets. As the recoloring has not generated new overloaded connections, we conclude

Theorem 1 *The PUs can determine on-line how to color the packets, such that on a $n \times n$ mesh a k-k transpose can be routed in $(1 - 1/\sqrt{2}) \cdot k \cdot n + \mathcal{O}(n/k)$ steps.*

3 Implementations

In contrast to the presentation of the optimal algorithm which we gave for the store-and-forward model, we now show how to proceed on an up-to-date parallel machine using a wormhole routing principle, the two-dimensional Intel Paragon. We first explain why the store-and-forward algorithm cannot immediately be adapted to match the specific requirements for efficient routing under the wormhole model.

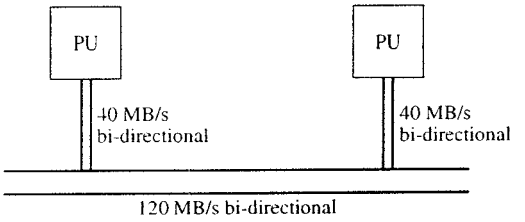


Figure 4. Paragon communication hardware.

Figure 3 shows the communication hardware as it appears to the user: the Paragon has a very powerful network, and relatively slow connections between the PUs and the network. Thus, it is hard to over-saturate the network: mostly the time of a routing is determined by the time a PU needs to pump data into and out of the network (*feeding-time*). Only if many packets are sent over the network the *throughput-time* dominates.

The hardware-supported standard routing invariably starts with the horizontal submovement (trivial XY-scheme) in order to avoid deadlock. If we nevertheless insist on routing (white packets) vertically first, the path has to be divided into two one-dimensional parts explicitly. However, looking at a single packet we find that splitting the path doubles the feeding time, since all the data has to be pumped into and out of the intermediate PU. Even worse, if we try to adjust our optimal algorithm in that manner, $\Theta(n)$ paths are broken at the PUs on the diagonal. Thus, on this machine the feeding time of some PUs on the diagonal dominates the throughput time of the trivial transposition algorithm.

3.1 Two-Phase Routing.

The implementational problems are caused by using few turning points for many packets. This leads to a poor feeding performance. Alternatively we can first split the contents of each PU and redistribute the fractions equally among all PUs of the net before routing them to their final destinations. If we use the standard router for the two phases, the feeding takes only twice as long because each PU is the intermediate host of the amount of data equivalent to only one packet.

Now we merely have to examine the occurring link loads. Due to the regular pattern it is sufficient to examine a horizontal link in one direction per phase. Consider the first j PUs of a row i with indices $(i,0),\ldots,(i,j-1)$ on a $n \times n$ grid. Clearly each of them sends a fraction $(n-j)/n$ of its data to the rightmost $n-j$ PUs, so the relative load for the link between $(i,j-1)$ and (i,j) equals $j\cdot(n-j)/n$. Maximizing over j gives $j = n/2$, and thus the total link load for this algorithm is bounded by $L_{2\text{ph}} = 2\cdot k\cdot(n/2)^2/n = k\cdot n/2$. Our implementation shows that the additional time consumption for the second phase pays back for $n \geq 14$ (see Figure 7).

3.2 Three-Phase Routing.

For bigger mesh sizes we can afford a third routing phase if this leads to a further reduction of the maximum link load. We first present an easy uni-axial scheme with accumulated load $3/2\cdot k\cdot n$ and then we reduce the load to $3/8\cdot k\cdot n$.

Algorithm HOR_BASIC_TRANS(i,j)

1. Send the own packet along the row to column $(n+j-i) \bmod n$.

2. Send the packet received in step 1 along the column to row $(i+j) \bmod n$.

3. Send the packet received in step 2 along the row to column $(n+i-j) \bmod n$.

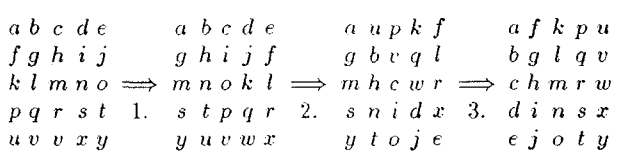


Figure 5. HOR_BASIC_TRANS **on a** 5×5 **mesh.**

The correctness of the algorithm follows by simply tracing the way a packet takes and using the fact that $(a \bmod n + b) \bmod n = (a+b) \bmod n$. Figure 5 gives an example for the routing. In phases one and two the biggest link loads appear at the central PUs where $k\cdot n/2$ packets move in one direction. For step three the crucial loads appear in the highest and lowest rows, also bounded by $k\cdot n/2$.

This weak point of the algorithm is that the maximal loads within each row/column are very unbalanced. We give a second algorithm that intensively uses previously undersaturated links, and vice versa. Overlaying the two algorithms each with half of the data reduces the overall maximum load.

Algorithm HOR_EXT_TRANS(i,j)

1. Send the own packet along the row to column $(\lfloor n/2 \rfloor + n + j - i) \bmod n$.

2. Send the packet received in step 1 along the column to row $(\lceil n/2 \rceil + i + j) \bmod n$.

3. Send the packet received in step 2 along the row to column $(\lfloor n/2 \rfloor + n + i - j) \bmod n$.

In Step 1 and 2 the hot-spots are now located on the outer rows/columns whereas the center is load free. In the third phase the row loads are cyclically shifted by $n/2$ rows in comparison to HOR_BASIC_TRANS.

The running time can be reduced by another factor two by running two orthogonal versions of the above uni-axial algorithms in parallel.

Our implementation shows that the three phase approach is faster than all others for $n \geq 28$ (see Figure 7). Link load patterns are provided in Figure 6.

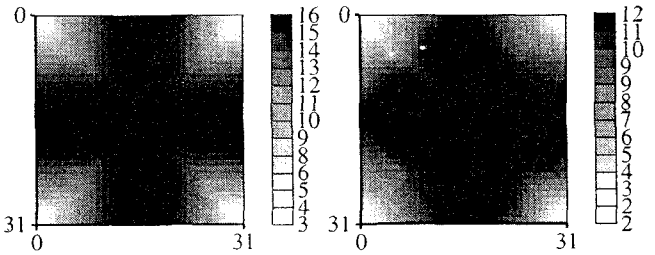


Figure 6. Link loads for the two- (left) and three-phase routing (right) on a 32×32 **mesh.**

3.3 Small Meshes

On small meshes ($n \leq 14$) we cannot allow a second phase of any form that results in doubling the feeding time. In order to beat the trivial approach we nevertheless have to redirect a certain amount of the data that are initially stored in the highest and lowest rows and cause high link loads in the upper-left and lower-right corners.

To demonstrate the new approach we first examine the case where we want to reduce the link loads caused by the first and by the last PU-row. Each PU holds k units of data. On an $n \times n$ mesh we thus have a link load of $L = k\cdot(n-1)$. A reduction of k units can be achieved if the $n-1$ PUs beside the diagonal redirect a fraction $1/(n-1)$ of their initial data. These packets can be sent vertically to intermediate positions within the inner PU-rows which have lower horizontal link loads. From there they are sent directly to their final positions. This is also feasible since the vertical links of the destination columns under consideration are sufficiently low-loaded. The feeding time then increases from t_f to $n/(n-1)\cdot t_f$.

A further reduction of the running time is possible by redirecting more than k data units to different intermediate positions in the inner part of the mesh. For the small mesh sizes we looked at, we found that in our implementation it is not profitable to relieve more than the upper and lower quarter of the mesh. Asymptotically, this leaves us with a throughput load of $3/4\cdot k\cdot n$ being much worse than the $3/8\cdot k\cdot n$ we found for the three-phase algorithm. A summary for the obtained running times is provided in Figure 7.

4 Sorting on Meshes

We now turn our attention to sorting. In a sorting problem, the destination of the packets is not known, but has to be determined by comparison of their keys. Then they have to be arranged with respect to a certain indexing scheme.

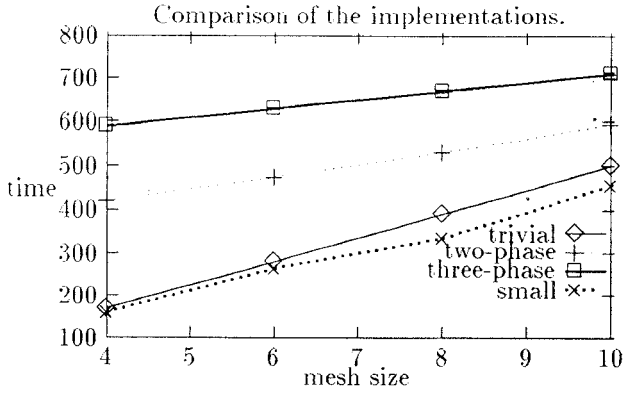


Figure 7. Running times for the different algorithms in milliseconds on the Intel Paragon for an $n \times n$ mesh, $n = 4, 6, 8, 10$. Each PU initially holds 8 Mbytes of data.

Most current communication algorithms strive for $T = \alpha \cdot n + o(n)$, with α as small as possible. The asymptotically optimal algorithms for two-dimensional meshes achieve a performance of $2 \cdot n + o(n)$ steps for 1-1 sorting and $k \cdot n/2 + o(k \cdot n)$ steps for k-k sorting [5, 3]. On d-dimensional meshes the best algorithms for 1-1 routing and sorting need $5/4 \cdot d \cdot n + o(n)$ steps, for any $d \geq 2$ [11]. For k-k problems on d-dimensional meshes the best bounds are $k \cdot n/2 + \mathcal{O}(k^{1-1/(3 \cdot d)} \cdot n^{2/3})$ [3].

However, all these approaches completely neglect the fact that actual meshes and hypercubes tend to be of fairly moderate sizes, for which the 'lower-order' terms may be several times larger than the 'leading' term. Furthermore, most of them work for '(blocked) snake-like' indexings, while we sort with respect to the more natural 'row-major' indexing. This suits the application of sorting as a subroutine better. For k-k sorting on two-dimensional meshes, the best result for practically reasonable values of n is about $2 \cdot k \cdot n$ [8].

To obtain improved running times for meshes of all dimensions, we apply optimized merge sorting algorithms as in [8], but add the idea that the routing power of the network can be used to a larger extent if orthogonal versions of the uni-axial procedure are run in parallel. The obtained results then must be combined by a transpose or orthogonalization, and a final sorting completes the work. Somehow we should prevent that this final sorting may be as expensive as the original problem. In order to achieve this, the packets must initially be sorted in small meshes, after which they can be colored with d 'colors', such that the distribution of the packets in each color is approximately the same.

In [4], we show how to get the following results, refining some of the techniques of [8]. The results are especially good for moderate sizes of n and k. We use the *non-layered* indexing, in which location r in P_i, $0 \leq r < k$, $0 \leq i < n^2$, has index $k \cdot i + r$, assuming a row-major ordering of the PUs.

Theorem 2 *Non-layered k-k-sorting on a $n \times n$ mesh with $n = 3^l$, for some $l > 0$, and $k \geq 15$ can be performed in $(31/40 \cdot k + 15) \cdot (n + \sqrt{n})$ steps.*

So, for $k \geq 250$, and $n \geq 27$, (or $k \geq 125$, and $n \geq 81$) the total routing time lies below $k \cdot n$. For three-dimensional meshes we get

Theorem 3 *Non-layered k-k-sorting on three-dimensional $n \times n \times n$ meshes with $n = 3^l$, for some $l > 0$, can be performed in $1.63 \cdot k \cdot n + \mathcal{O}(n + k \cdot n^{2/3})$ steps.*

Extending the ideas to d-dimensional meshes we achieve

Theorem 4 *If $k \geq d$ and $n \geq d^2$, then on a d-dimensional mesh, non-layered k-k-sorting can be performed in $\mathcal{O}(d \cdot k \cdot n)$.*

References

[1] Cormen T., K. Bruhl, 'Don't be Too Clever: Routing BMMC Permutations on the MasPar MP-2' *Proc. 7th ACM Sym. on Parallel Algorithms and Architectures*, ACM, pp.288–297, 1995.

[2] Ding, K.-S., C.T. Ho, J.-J. Tsay, 'Matrix Transpose on Meshes with Wormhole and XY Routing,' *Proc. 6th Symp. Par. Dist. Proc.*, pp. 656–663, IEEE, 1994.

[3] Kaufmann, M., J.F. Sibeyn, T. Suel, 'Derandomizing Algorithms for Routing and Sorting on Meshes,' *Proc. 5th Symp. on Discrete Algorithms*, pp 669–679 ACM-SIAM, 1994.

[4] Kaufmann, M., U. Meyer, J.F. Sibeyn, 'Matrix Transpose on Meshes: Theory and Practice,' *Techn. Rep. MPI-I-97-10xx*, Max-Planck Institut für Informatik, Saarbrücken, Germany, 1997, to appear.

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

[6] Nassimi, D., S. Sahni, 'An Optimal Routing Algorithm for Mesh-Connected Parallel Computers,' *Journal of the ACM*, 27, pp. 6–29, 1980.

[7] Nesson T., S.L. Johnsson, 'ROMM Routing on Mesh and Torus Networks,' *Proc. 7th Symposium on Parallel Algorithms and Architectures*, ACM, pp. 275–287, 1995.

[8] Sibeyn, J.F., 'Desnakification of Mesh Sorting Algorithms,' *Proc. 2nd European Symposium on Algorithms*, LNCS 855, pp 377–390, Springer, 1994. to appear in SIAM J. Comp.

[9] Sibeyn, J.F., 'Matrix Techniques for Faster Routing of Affine Permutations on a Mesh Interconnection Network,' *Techn. Rep. RUU-CS-90-17*, Utrecht University, 1990.

[10] Stout, Q.F., B. Wagar, 'Intensive Hypercube Communication, Prearranged Communication in Link-Bound Machines,' *Journal of Par. and Dist. Comp.*, 10, pp. 167–181, 1990.

[11] Suel, T., 'Improved Bounds for Routing and Sorting on Multi-Dimensional Meshes' *Proc. 6th ACM Sym. on Parallel Algorithms and Architectures*, ACM, pp.26–35, 1994.