Matrix Transposition on a Mesh with Blocking Transmissions

Micha Hofri
Dept. of Computer Science

David L. Thomson
Dept. of Mechanical Engineering

Rice University, Houston TX 77005
e-mail: {hofri,thomson}@rice.edu

February 7, 1997

Abstract

A time-optimal procedure to transpose in situ a matrix stored over a distributed memory 2-dimensional mesh connected parallel computer is shown. The matrix need not be square. Only nearest-neighbor blocking communications is used, and small bounded buffer space is required.

Key words: Parallel computing; 2-dimensional mesh connections; Blocking transmissions; Matrix transposition; Minimal time; Rearrangement.

1 Introduction

We consider a parallel computer with an $N \times M$ two-dimensional mesh connections: processing elements (PEs) are assumed to reside in the nodes of such a grid, one per node. We also assume that local memories are the only fast (main) storage used by the computer. This is a commonly implemented architecture, as well as the natural connectivity for most matrix algorithms. See [5] for a detailed discussion of this architecture – called there an 'array.'

The mesh-connectivity implies that direct inter-node communications are only possible with grid-adjacent nodes (hence each PE has 2, 3, or 4 communicating partners). For the purposes we consider below, each communication lasts essentially the same time, and this duration is taken as our time unit. The communications is via *blocking* messages. This has the result that a PE can only transmit to (or receive from) one of its neighbors at each time unit (it may, however, transmit and receive simultaneously).

One of the basic sub-problems in parallel algorithms is that of transposing a matrix *in situ*. The most common application may be as a step in the two-dimensional FFT, but many others exist. Accordingly, this problem has attracted considerable attention. See [4, §5.2] for a discussion of several general approaches.

The treatments of transposition on mesh computers that we found in the literature fall into two classes: (1). The computer is assumed to have capabilities as above, and transposition is only treated as a particular case of matrix permutation. This is typically solved by assigning to the matrix elements tags corresponding to their desired final position, and sorting the entire array by tag value. Leighton shows in [5, §§1.6.2-3] two such sorting algorithms: Shearsort performs alternate odd-even transposition sorts on rows and columns, and for a square $n \times n$ matrix requires $\Theta(n \log n)$ steps; a more elaborate algorithm is then shown, with an asymptotic duration in $3n(1 + o(1))$. Moreover, it is proved there that no faster sorting method is possible with these communication constraints. While this method performs *arbitrary* permutations, we did not find

in the literature any specific method for transposition, that uses its special properties, one of which is that it can be done in $2(n-1)$ steps.
(2). In the second class, more aggressive communications abilities are assumed, such as direct non-adjacent transmissions, or non-blocking `send` and `receive` primitives, and other relaxations. The constructions of [2, 3] are typical, and they provide additional references.

The most noteworthy method there is probably [6], which allows nodes to communicate with several neighbors simultaneously. They conjecture—and bring substantial simulation results in support—that their heuristic is near-optimal in general, and show that it transposes a matrix in $2(n-1)$ steps. However, the need for simultaneous transmissions is nonremovable there.

2 The Algorithm

Figure 1 presents the idea of a new algorithm to transpose a square matrix under the above assumptions. The matrix elements travel in a snake-like fashion. Diagonal elements do not move.
The two drawings show the minor difference in the lower right corner that distinguishes between matrices of odd and even order.
The following discussion assumes the matrix is $n \times n$ and that we use 0-origin numbering for the rows and columns. An element is denoted by its position as $(row, column)$.

The snake "takes" each element only as far as it needs to reach its new (transposed) location: the term from location $(1,0)$ moves twice only, and at step $2(n-1)$ the only element in the snake that still moves is its 'tail,' the one that came from location $(n-1,0)$, and now moves to its final position $(0,n-1)$.

As is evident from the diagrams, only half the elements can get home in this way. The rest, elements which originate in even-indexed rows above the diagonal, or in odd-numbered columns below the diagonal, are brought each to a location which is in the correct final row or column, respectively, but also one column to the right or one row below its final position, respectively. E.g., the element from $(0,1)$ arrives after $2(n-1)-1$ steps to $(1,1)$. All these elements make their final move in the very last time unit (most could move earlier, but there is no gain in doing so).

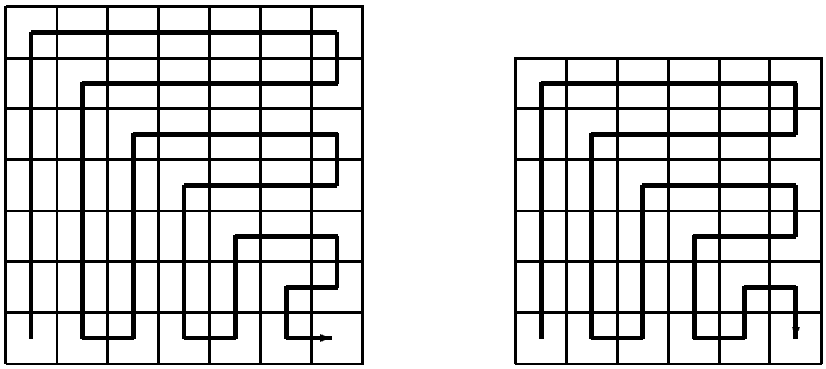


Figure 1: Data movement for transposition in square matrices of odd and even order

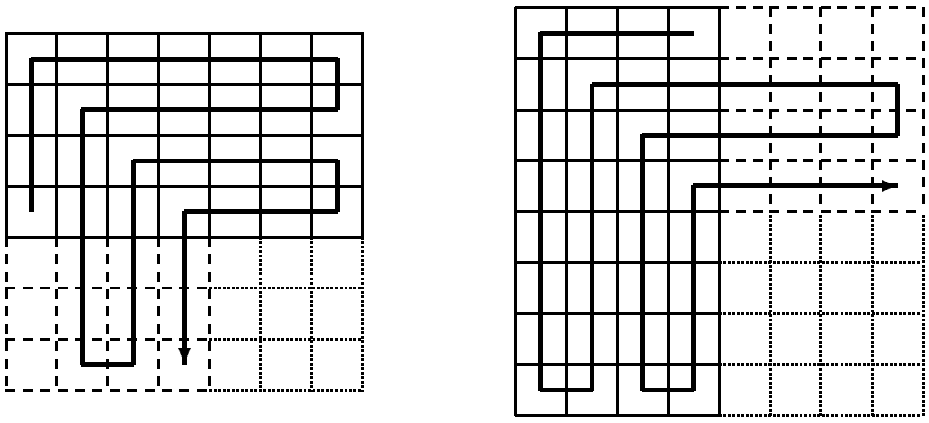
There are several possible ways to implement this data movement; the C-like procedure at the end shows one possibility, assuming only one important convention: the `forall` statement forces a synchronization of all PEs. Initially the matrix elements are in the local variables a. Each PE defines a structure A for one

tagged element. It has three fields which can be pre-computed: $A.a$ is a matrix element; $A.c$ is the number of times this element needs to be moved; $A.t$ is 0 for an element that is brought by the snake to its correct place, it is $+1$ if $A.a$ needs to be moved up one row in the final step, and it is -1 if it needs to be moved then one column to the left. The variable $current$ is 1 when the PE has an element that needs to be moved on, and zero otherwise. Each PE—except $(n-1, n-1)$—has a neighbor: nbr denotes the coordinates of the next processor along the snake. The symbol $\Longrightarrow$ denotes a blocking `send` operation.

This implementation and diagrams for the algorithm assume that the matrix is stored as one element per PE. If the local memories hold instead submatrices (assumed all of the same dimensions), these blocks are communicated *in toto* at each step, and at some stage need to be transposed locally, without requiring any further communications. This is the only scenario where elements in the diagonal PEs are affected.

There is a degree of freedom beyond what Figure 1 shows: we could use a similar snake that starts at the upper right corner and proceeds in a (nearly) symmetrical way about the diagonal. Naturally, both the neighbor defined for each PE and the arrangement of delayed elements would change.

It turns out that this degree of freedom makes no useful contribution above, but it is useful when nonsquare matrices need to be transposed. Then it is needed to maintain *optimal* performance (for sometimes-nonoptimal, but trivial implementation, one can just embed such a matrix in a square one). There are two cases to distinguish. Let the matrix be $n \times m$. The cases differ according to the parity of the smaller dimension, and whether it is the row or column count. The equivalent of Figure 1, given in Figure 2(a), holds for an even n, when $n < m$ or odd m, when it is the smaller dimension. The original 4×7 is transposed into the 7×4 partly-dashed area. The doted PEs are there to complete the grid only, and do not take part in this computation. The parity of the larger dimension never matters. Figure 2(b) shows one (m is smaller and even) of the two cases that require the other snake.



(a) (b)

Figure 2: Data movement for transposition of a nonsquare matrix

3 Properties

The procedure is transparent enough to claim all its characteristics are evident by inspection, but we collect the most important as follows:

Theorem:

1. *The algorithm of §2 performs the transposition in $2(n-1)$ steps and satisfies the connectivity and communications constraints (CCC).*
2. *The algorithm requires each PE to allocate temporary storage for one element only.*

Proof: Since data move during the first $2(n-1)-1$ steps only along the snake, the CCC are then satisfied. In the last step they are satisfied since (a) each PE harbors at most one delayed element; (b) the only snake motion then is in between a pair that does not take part in moving the delayed elements.

It is easy to see, that once a PE receives a delayed element, it moves no other elements till the last step, hence the sufficiency of a single temporary storage location. □

Acknowledgement

We are grateful to Hristo Djidjev for pointing out to us the relevance of shearsort—a two-dimensional adaptation of the odd-even transposition sort—to the matrix transposition problem. In addition, he showed that it is possible to use a much simpler approach, which costs exactly one time unit more than our method: partition the (square) matrix as shown in Figure 3, and reverse the order of the elements in each strip using the odd-even transposition sort.

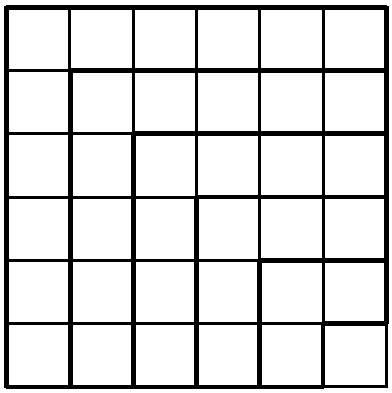


Figure 3: An alternative approach to near-optimal simple transposition

References

[1] Christopher Calvin, Denis Trystram, Matrix transpose for block allocations on torus and de Bruijn networks. *Journal of Parallel and Distributed Computing*, **34** (1), 36–49 (1996).

[2] J. Choi, J. Dongarra, D. Walker, Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers, *Parallel Computing* **21**, 1387–1405 (1995).

[3] Michel Cosnard, Denis Trystram, *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.

[4] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, *Introduction to Parallel Computing – Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.

[5] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[6] Fillia Makedon, Antonios Symvonis, An Efficient Heuristic for Permutation Packet Routing on Meshes with Low Buffer Requirements, *IEEE Trans. Par. and Distr. Sys.* **4**, #3, 270–276 (1993).

```
forall P_jk,  0 ≤ j,k < n                --- initialize ---
  { if (j = k) { current = 0;  A.c = A.t = 0;
                 if (j is odd) nbr = (j+1, k);
                   else        nbr = (j, k+1);
               }
    else       {A.a = a; current = 1;
               if (j < k  ) { if (j is odd) { A.t = 0; A.c = 2*(k-j); nbr = (j, k-1); }
                                    else    { A.t = -1; A.c = 2*(n-j)-3;
                                              if (k < n-1) nbr = (j, k+1);
                                                 else      nbr = (j+1, k);
                                              }
                            }
               if (j > k  ) { if (k is even) { A.t = 0; A.c = 2*(j-k); nbr = (j-1, k); }
                                    else { A.t = 1; A.c = 2*(n-k)-3;
                                           if (j < n-1) nbr = (j+1, k);
                                              else      nbr = (j, k+1);
                                           }
                            }
               }
    }
for l = 1  to  2*(n-1)-1                          --- move ---
    {forall P_jk,  0 ≤ j,k < n
        { if (current = 1) { current = 0;
                             (A,1) ⟹ [nbr](A, current);
                           }
        }
     forall P_jk,  0 ≤ j,k < n
         { if  (current = 1)  { A.c = A.c - 1;
                                if  (A.c = 0)  { current = 0;   if  (A.t = 0)   a = A.a; }
                              }
         }
    }
forall P_jk,  0 ≤ j,k < n                --- last step ---
      { if  (j = 0 and k = n-2)  A.a ⟹ [nbr] a;          /*Or ask if current = 1. */
        if (A.t = 1)             A.a ⟹ [(j-1, k)] a;
        if (A.t = -1)            A.a ⟹ [(j, k-1)] a;
      }
```

A procedure for square matrix transposition in situ