2. Efficient Matrix Transposition

J. O. Eklundh

In this chapter we shall describe a number of different algorithms for transposition of matrices that can be accessed row- or columnwise and discuss their efficiency in terms of required memory and input and output operations. In particular, we shall dwell on matrices whose dimensions are highly composite numbers, e.g., powers of two. In fact, optimal algorithms will be presented in the latter case. We shall also discuss how an arbitrary matrix can be handled, by embedding it into a larger one with desirable properties. Most of the algorithms described require that each row (or column) can be addressed randomly, as in a random access file on disk, but some attention will also be given to the case when then rows are accessible only in sequence, as in a tape file. The first case is, however, of special interest, since then some of the methods allow square matrices to be transposed in-place, which in turn implies that any 2-D separable transform of an externally stored square matrix can be computed in-place as well.

We shall also present the direct approach. A comparison between the two approaches will show that they are essentially equal in performance in the cases that are common in the applications. It will, however, also be observed that important differences exist.

2.1 Background

Integral transforms, for example the Fourier transform, are important computational tools in digital signal processing. In two dimensions their discrete forms are double sums computed by summing iteratively in the two coordinate directions. It is straightforward to compute these double sums when the entire matrix can be stored in high speed memory. A problem arises when this is impossible or inconvenient. In many applications the matrix is stored on a block storage device, e.g., a disk, where the smallest record that can be accessed is an entire row or column. In this environment the direct computation of the transform is expensive because of the time involved in accessing the externally stored matrix.

One way of avoiding this problem is to transpose the matrix after the first summation. Another method due to *Anderson* [2.1] can be derived directly from the definition of the fast Fourier Transform (FFT), the crucial point being

that the FFT is built up by transforms of subarrays and that all these transforms are computed in place.

Both these approaches are based on the separability of the Fourier transform. The 2-D discrete Fourier transform (DFT) of a complex array, $x(m,n)$, $m=0,1,\ldots,M-1$, $n=0,1,\ldots,N-1$ may be defined as

$$X(k,l)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)\,W_M^{km}W_N^{ln}, \quad k=0,1,\ldots,M-1, l=0,1,\ldots,N-1 \tag{2.1}$$

where $W_P=\exp(-2\pi j/P)$ is used for notational convenience. Separability means that the DFT kernel $W_M^{km}W_N^{ln}$ is a product of two functions depending on, respectively, m and n alone. This allows us to compute the transform in two steps, first computing

$$Y(m,l)=\sum_{n=0}^{N-1} x(m,n)\,W_N^{ln} \tag{2.2a}$$

and then

$$X(k,l)=\sum_{m=0}^{M-1} Y(m,l)\,W_M^{km} \tag{2.2b}$$

where the first sum is an N-point 1-D DFT over the *rows* of the matrix $[x(m,n)]$ and the second is an M-point 1-D DFT over the *columns* of the resulting matrix $[Y(m,l)]$. Consequently the 2-D transform can be computed by the use of a 1-D transform and a matrix transposition. This is of particular interest because there is a "fast" algorithm for the 1-D DFT and, moreover, because this algorithm can be implemented in hardware.

On the other hand, the fact that the sum in (2.2b) is also a DFT implies that one can use the iterative summation technique of the FFT mixed with input and output of the transformed rows to compute $[X(k,l)]$ directly.

The transposition approach clearly applies to any separable transform, whereas the direct approach works when, in addition, a "fast" algorithm exists as, e.g., in the case of the Hadamard transform. Both methods also generalize to higher dimensions. (See [2.2] on how the DFT can be computed by use of transpositions).

More important, perhaps, is that also 1-D transforms of large arrays can be broken up into transforms of smaller arrays and matrix transpositions. In the case of the DFT this can be done in the following way (according to [2.2]):

The DFT of the array $x(m)$, $m=0,1,\ldots,M-1$ can be defined as

$$X(k)=\sum_{m=0}^{M-1} x(m)\,W_M^{km} \quad k=0,1,\ldots,M-1. \tag{2.3}$$

Now, if $M=M_0M_1$, we may write

$$m=m_1M_0+m_0 \quad 0\leqq m_0<M_0, 0\leqq m_1<M_1 \tag{2.4}$$

and

$$k = k_0 M_1 + k_1 \qquad 0 \leqq k_0 < M_0, 0 \leqq k_1 < M_1 \tag{2.5}$$

and consider x and X as matrices letting

$$x(m) \Leftrightarrow y(m_0, m_1) \tag{2.6}$$

and

$$X(k) \Leftrightarrow Y(k_1, k_0). \tag{2.7}$$

Since $W_M^{M_0 M_1} = 1$ and $W_M^{M_1} = W_{M_0}$, we get

$$\begin{aligned} Y(k_1, k_0) &= \sum_{m_0=0}^{M_0-1} \sum_{m_1=0}^{M_1-1} y(m_0, m_1) W_M^{(m_0 + m_1 M_0)(k_1 + k_0 M_1)} \\ &= \sum_{m_0=0}^{M_0-1} \sum_{m_1=0}^{M_1-1} y(m_0, m_1) W_M^{(m_0 + m_1 M_0) k_1} W_{M_0}^{m_0 k_0}. \end{aligned} \tag{2.8}$$

Computing the sum in two steps we have

$$Z(m_0, k_1) = \sum_{m_1=0}^{M_1-1} y(m_0, m_1) W_M^{(m_0 + m_1 M_0) k_1}$$

$$m_0 = 0, 1, \ldots, M_0 - 1, k_1 = 0, 1, \ldots, M_1 - 1, \tag{2.9}$$

and

$$Y(k_1, k_0) = \sum_{m_0=0}^{M_0-1} Z(m_0, k_1) W_{M_0}^{m_0 k_0}$$

$$k_0 = 0, 1, \ldots, M_0 - 1, k_1 = 0, 1, \ldots, M_1 - 1. \tag{2.10}$$

Here, given the row of y, we first compute the corresponding *row* of Z by multiplication of the appropriate phase factors and summing. Then, given the *columns* of Z, we compute its DFT, which is the corresponding column of Y. To do this, again, we need to transpose Z.

Other methods have also been proposed for computing one- and multidimensional transforms with limited high-speed storage, some requiring matrix transposition (see, e.g., [2.2, 3]) and some not (see, e.g., [2.4, 5]). The latter approach is generally less efficient.

2.2 Methods for Transposing Externally Stored Matrices

2.2.1 Definition of Performance Criteria

In the next few sections we shall describe a number of methods for transposition of matrices stored on random access devices so that each record contains one row (or column; in the sequel we shall, however, always assume that the matrices are stored in row major order). Generally, all the algorithms achieve their task by reading a limited number of data records into high-speed memory, rearranging the data there, forming new output records and writing these on an output file, which may or may not coincide with the input file. Their performance will be defined in terms of the required number of high-speed memory locations, RM, and the required number of input and output operations (i.e., the number of records that have to be read and written), IO. These measures do not account for all the computational costs involved, but the remaining costs may be neglected or considered invariant when different methods are applied to the same matrix. More specifically, there are two main types of costs that are not included.

The first one is caused by the rearrangement of data in high-speed storage to form the new output records. However, one can create the new records *at output* without actually moving any data in storage. For example, it is possible to use implicit loops in the output statements in *Fortran* (see Sect. 2.7). Consequently, the CPU time needed is small and, moreover, it is proportional to the number of elements in the matrix times the number of passes over the data, which already is captured in the number of input and output statements. It should be noted that although we think of the algorithms as being implemented in this manner, in their descriptions we shall, for simplicity, still talk about transposing matrices and rearranging arrays in high-speed storage.

Secondly, by only counting the number of records that are transferred, we disregard the fact that the record lengths may vary when one or the other method is applied. Hence, it would be more accurate to account also for the number of words that have to be transferred. This cost can, however, be neglected for two reasons. First, the time needed to access one record is normally much higher than the time needed to transfer one word. For a modern high-speed disk the ratio between these two times typically exceeds 1000. Furthermore, a close look at the different methods will show that the variations in record lengths that occur generally are small compared to that ratio.

2.2.2 A Simple Block-Transposition Method

A simple and straightforward technique for transposition of an $M \times N$ matrix is presented in [2.1]. The matrix is transposed by first splitting up the original records of length N into records of length K, where K is chosen such that KM elements can be stored in high-speed storage. Then for $i = 1, \ldots, N/K$ the records

$j=1, i+N/K, \ldots, i+(M-1)N/K$ are read into a $K \times M$ matrix in high-speed storage which is subsequently transposed. K consecutive columns of the desired transpose are then obtained and can be written out.

The number of read and write operations needed for this algorithm is proportional to MN/K. To give good performance, K and hence the required amount of high-speed storage should be large. In fact, the method matches other proposed methods only when almost the entire matrix can be stored in high-speed memory. A similar performance is achieved by another block-transposition technique suggested by *Hunt* [2.3].

2.2.3 Transposition Using Square Partitions

The second algorithm was first proposed by *Eklundh* [2.6]. It is based on successive transposition of square partitions and forms a natural generalization of the method given by *Eklundh* [2.7]. It is also similar to a method used by *Ramapriyan* [2.8].

Let A be an $M \times N$ matrix and suppose that $\bar{M}=m_1 \ldots m_p \geqq M$, where $m_i > 1$. Set $m_0=1$, $M_0=M$, $N_0=N$ and define for $i=0, 1, \ldots, p$

$$P_i=m_1 \cdot \ldots \cdot m_i, \tag{2.11}$$

$$M_i=\left\lceil \frac{M_{i-1}}{m_i} \right\rceil \tag{2.12}$$

and

$$N_i=\left\lceil \frac{N_{i-1}}{m_i} \right\rceil \tag{2.13}$$

where $\lceil\ \rceil$ is defined by

$$\lceil x \rceil = \begin{cases} x, & x \text{ integer} \\ [x]+1 & \text{otherwise,} \end{cases} \tag{2.14}$$

$[x]$ denoting the integer part of x. Then the transpose of A can be constructed in p steps as follows.

Step 1: Form M_1 matrices of size $m_1 \times N$ from consecutive rows of A. Each such matrix can, possibly after padding with zeros [at most $m_1(m_1-1)$], be partitioned into N_1 matrices of size $m_1 \times m_1$. These square matrices are now transposed in place. We obtain a matrix $A^{(1)}$, whose elements are $1 \times P_1$ matrices and the dimension of which is is $P_1 M_1 \times N_1$.

Inductively we define the ith step, when the first $i-1$ steps are completed.

Step i: In step i we start with a $P_{i-1}M_{i-1} \times N_{i-1}$ matrix, $A^{(i-1)}$, whose elements are $1 \times P_{i-1}$ matrices. Then M_i submatrices of size $m_i \times N_{i-1}$ are formed by selecting rows that are P_{i-1} rows apart in the following way: If λP_i rows, $\lambda = 0, 1, \ldots, M_i - 1$, have been processed we pick out m_i rows, P_{i-1} rows apart, starting with each of the ensuing P_{i-1} rows, that is starting with row $\lambda P_i + \mu$, $\mu = 1, 2, \ldots, P_{i-1}$. Hence, a typical submatrix consists of the rows (in order) $\lambda P_i + \mu + \nu P_{i-1}$, where $\nu = 0, 1, \ldots, m_i - 1$ and λ and μ are fixed in the ranges given above. Such a matrix can, if necessary after zero-filling, be partitioned into N_i matrices of size $m_i \times m_i$, whose elements are $1 \times P_{i-1}$ matrices. These square matrices are transposed, and a $P_i M_i \times N_i$ matrix $A^{(i)}$, with $1 \times P_i$ matrices as elements, is obtained. The $1 \times P_i$ matrices are parts of the rows of $\tilde{A}$, the transpose of A.

This description of the algorithm gives an intuitive feeling of how the rows of $\tilde{A}$ are constructed as parts of the records of $A^{(p)}$ and will also be useful in the analysis of the performance. However, it is not immediate that each element will end up in the right place. Therefore, we shall describe the algorithm in a different way without referring to partitions and so be able to prove that the algorithm works, at the same time that we indicate how it can be implemented.

First we look at the case when $M \geqq N$. Let $[k^{(i)}, l^{(i)}]$ denote the position in $A^{(i)}$ (no longer considered as partitioned) of the element $a(k, l)$ of A, $i = 1, \ldots, p$, $k = 0, 1, \ldots, M-1$, $l = 0, 1, \ldots, N-1$. We then observe that k and l can be uniquely represented in the form

$$\begin{aligned} k &= k_{p-1}P_{p-1} + \ldots + k_1 P_1 + k_0 \\ l &= l_{p-1}P_{p-1} + \ldots + l_1 P_1 + l_0 \end{aligned} \tag{2.15}$$

where $0 \leqq k_i < m_{i+1}$, $0 \leqq l_i < m_{i+1}$ and P_i is given by (2.11). The representations (2.15) are evidently obtained from the principal remainders at successive divisions by $m_1, m_2, \ldots, m_p$, as in an ordinary number system representation. Let us simply use the notation

$$\begin{aligned} k &= n(k_{p-1}, \ldots, k_1, k_0) \\ l &= n(l_{p-1}, \ldots, l_1, l_0). \end{aligned} \tag{2.16}$$

Now step one implies that A is partitioned into $m_1 \times m_1$ matrices which are transposed. This means that

$$\begin{aligned} k^{(1)} &= n(k_{p-1}, \ldots, k_1, l_0) \\ l^{(1)} &= n(l_{p-1}, \ldots, l_1, k_0) \end{aligned} \tag{2.17}$$

since $k_1, \ldots, k_{p-1}$ and $l_1, \ldots, l_{p-1}$ just determine which $m_1 \times m_1$ matrix the element belongs to.

Inductively we assume that after $i-1$ steps we have

$$\begin{aligned} k^{(i-1)} &= n(k_{p-1}, \ldots, k_i, k_{i-1}, l_{i-2}, \ldots, l_0) \\ l^{(i-1)} &= n(l_{p-1}, \ldots, l_i, l_{i-1}, k_{i-2}, \ldots, k_0). \end{aligned} \tag{2.18}$$

Step i can then be described in the following way. Consider the matrix as partitioned into $P_i \times P_i$ matrices. Then, inside each such matrix, form $m_i \times m_i$ matrices by picking out m_i rows and the same m_i columns that are P_{i-1} rows (columns) apart in all possible ways. This means that $k_i, \ldots, k_{p-1}$ and $l_i, \ldots, l_{p-1}$ remain unchanged, since they determine the specific $P_i \times P_i$ matrix, and so do $l_{i-2}, \ldots, l_0$ and $k_{i-2}, \ldots, k_0$ because they determine which of the $m_i \times m_i$ matrices the element belongs to. In this particular matrix $a(k, l)$ occurs in row k_{i-1} and column l_{i-1} before transposition and so

$$\begin{aligned} k^{(i)} &= n(k_{p-1}, \ldots, k_i, l_{i-1}, l_{i-2}, \ldots, l_0) \\ l^{(i)} &= n(l_{p-1}, \ldots, l_i, k_{i-1}, k_{i-2}, \ldots, k_0). \end{aligned} \tag{2.19}$$

It follows that $[k^{(p)}, l^{(p)}] = (l, k)$, proving that $a(k, l)$ has the right position in $A^{(p)}$. We observe that the transposition is performed by a sequence of swaps over the digits in the mixed radix representation (2.15). In particular, if $m_i = 2$ for all i, we get the algorithms described in [2.7, 9].

It only remains to establish the relation between $A^{(p)}$ and $\tilde{A}$. If $\bar{M} = M$, the first N rows of $A^{(p)}$ obviously are the rows of A, whereas the last $\bar{M} - N$ rows contain only zeros. In fact, they need never be created. If, on the other hand, $\bar{M} > M$, then the rows of $A^{(p)}$ also contain $\bar{M} - M$ zeros at the end, which like the zero rows, never have to be written out.

The case $M < N$ can be treated in a similar way, if in (2.15) we write

$$l = l_p P_p + l_{p-1} P_{p-1} + \ldots + l_0 \tag{2.20}$$

with $0 \leqq l_i < m_{i+1}$ for $i = 0, 1, \ldots, p-1$ and $0 < l_p$. The final position of $a(k, l)$ will now be $[k^{(p)}, l^{(p)}]$, where

$$k^{(p)} = l_{p-1} P_{p-1} + \ldots + l_0 \tag{2.21}$$

$$l^{(p)} = l_p P_p + k_{p-1} P_{p-1} + \ldots + k_0 . \tag{2.22}$$

In this case the rows of $A^{(p)}$ contain one ($\bar{M} \geqq N$) or several ($\bar{M} < N$) rows of $\tilde{A}$. Writing $A^{(p)}$ as partitioned into N_p $\bar{M} \times \bar{M}$ matrices, where N_p is defined by (2.13),

$$A^{(p)} = (B_1 | B_2 | \ldots | B_{N_p}), \tag{2.23}$$

we obtain $\tilde{A}$ as

$$\left(\begin{array}{c} B_1 \\ \hline B_2 \\ \hline \vdots \\ \hline B_{N_p} \end{array}\right) = \left(\begin{array}{c|c} \tilde{A} & 0 \\ \hline 0 & 0 \end{array}\right). \tag{2.24}$$

It should be observed that this operation as well as the cancelling of the zeros is performed at output in step p and without additional data handling.

If $M \neq N$ then, at least for some products $\bar{M}$, the algorithm will first transpose $K \times K$ matrices, where $K = \min(M, N)$ and then concatenate ($M < N$) or split up ($M > N$) the rows.

If the rows of $A^{(i)}$ are created at output, then the performance in terms of required high-speed storage locations is measured at step i by

$$\mathrm{RM_S} = \max_{1 \leqq i \leqq p} \{m_i N_{i-1} P_{i-1}\}. \tag{2.25}$$

M extra storage locations would allow us to actually form the rows of A in high-speed storage, which is of interest when 2-D transforms are computed. This can, however, also easily be done by transpositions in main storage using no extra storage if $m_p \leqq N_p$ and using $m_p N_p$ additional locations if $m_p > N_p$ (see Sect. 2.7). It will turn out that for optimal embeddings generally $m_p N_p \ll M$.

One can note that performing the transpositions in main storage at each step, instead of just writing out the words in the right order, not only is slower but also requires more memory. More precisely, one gets

$$\mathrm{RM_{S'}} = \max_{1 \leqq i \leqq p} \{m_i N_i P_i\}. \tag{2.26}$$

We shall later (Sect. 2.3.1) show that $N_i P_i$ increases with i, which implies that $\mathrm{RM_{S'}} \geqq \mathrm{RM_S}$.

If the zero rows of $A^{(p)}$ are not written out, the expression for the required number of input and output operations will be

$$\mathrm{IO_S} = M + 2 \sum_{i=1}^{p-1} M_i P_i + N. \tag{2.27}$$

If $\bar{M} = M$, the algorithm presented here has the property that all the matrices $A = A^{(0)}, A^{(1)}, \ldots, A^{(p-1)}$ contain M rows, while $A^{(p)}$, if the zeros are discarded, contains N rows. This implies that the number of input and output operations depends only on the number of factors of M. If we, as mentioned earlier, disregard the varying (increasing) row lengths, we can optimize the algorithm for a given M and p by minimizing the required number of high-speed storage locations. It will be shown how a minimizing factorization of M can be

characterized under certain conditions. If these conditions do not hold, the optimum can still be determined easily in any particular case. Moreover the dependence of the solution on p will be demonstrated. If $\bar{M} > M$, a factorization minimizing the memory requirements will no longer minimize the required number of input and output operations. However, it will be shown that the variation in number of IO operations between different factorizations with the same number of factors is fairly small. This suggests a simple algorithm, which iterates over increasing $\bar{M}$ and quickly finds the factorization minimizing the memory requirements for each p. Because $p \leqq \lceil \log_2 M \rceil$ it is then easy to determine which of these solutions is cheapest in terms of memory *and* input/output operations. In general one can in this way determine a strategy which is close to the global optimum. The approach will be outlined in Sect. 2.4.

2.2.4 Floyd's Algorithm

The square partition algorithm has the property that, if all the factors of $\bar{M}$ are equal to 2 or powers of 2, then it can be implemented using only shift and masking operations. This follows from the description in Sect. 2.2.3 (see also [2.7]). Another algorithm, presented by *Floyd* [2.10], also has this property. It coincides with the square partition algorithm if $M = N = 2^m$ and it can be applied to any square matrix directly (without zero-filling).

An $M \times M$ matrix A can be transposed in $p = \log_2 M$ passes over data using $2M$ main storage locations in the following way:

1) Form $A^{(1)}$, where $a^{(1)}(k, l) = a(-k, l)$.
2) Form $B^{(0)}$, where $b^{(0)}(k, k+l) = a^{(1)}(k, l)$.
3) Recursively, for $i = 1, \ldots, p$, form $B^{(i)}$ from $B^{(i-1)}$ by shifting the columns for which the ith binary digit [l_{i-1} in (2.15)] equals $1\,2^i$ steps.
4) Form $\tilde{A}$ from $B^{(p)}$, setting $\tilde{a}(k, l) = b^{(p)}(k, k+l)$.

Observe that neither $A^{(1)}$ nor $B^{(0)}$ or $B^{(p)}$ actually have to be created. A proof of the algorithm is given in [2.10].

Using, say, $2^m M$ high-speed locations, one can execute m steps of the algorithm in one pass, as in the square partition algorithm. However, this algorithm then requires some additional bookkeeping, since the cycling of the elements spans over *all* the rows and therefore cannot be completed inside a given slice of 2^m rows.

Summarizing, we see that when

$$\mathrm{RM_F} = 2^m M \tag{2.28}$$

high-speed locations are used, the number of IO operations is

$$\mathrm{IO_F} = 2M \lceil \log_2 M/m \rceil . \tag{2.29}$$

In particular, it is shown in [2.10] that if M is a power of 2, then the right-hand side of (2.29) is also a lower bound on the number of IO operations needed to transpose an $M \times M$ matrix with the given memory. Hence, this algorithm as well as the square partition algorithm and the algorithm presented in the next section all minimize IO for given RM. It can be observed that the same lower bound on IO was derived by *Stone* [2.11], who also derived an algorithm attaining that bound. However, his algorithm is not directly applicable to this problem, neither can it be implemented using only shift and masking operations.

Equation (2.29) gives only an upper bound on the number of IO operations required. The optimal algorithm may in fact do strictly better. For instance, the optimal algorithm will for a 5×5 matrix with $m = 1$ give $\mathrm{IO} = 28$ compared to the upper bound that is 30 [2.10]. No algorithm that gives minimal IO for given RM for *all* square matrices has, however, yet been presented. As it is described here, Floyd's algorithm will in general require more IO operations than the other algorithms we describe. However, these generally use somewhat more high-speed storage or external storage. An example of this will be given in Sect. 2.5.

Floyd's method can obviously also be generalized to rectangular matrices using the approach described in Sect. 2.2.3.

2.2.5 Row-in/Column-out Transposition

This algorithm is a generalization of an algorithm for transposing matrices stored on sequential devices proposed by *Knuth* [2.12].

The original algorithm transposes an $M \times M$ matrix stored sequentially (one record being one row) in $\lceil \log_2 M \rceil$ passes over the data, using $2^{\lceil \log_2 M \rceil + 1}$ high-speed storage locations and four additional sequential files as follows.

At each step two files are created which at the next step are processed synchronously:

Step 1:

File 1: $a_{11}a_{21}a_{12}a_{22} \ldots a_{1,M/2}a_{2,M/2} \ldots a_{1M}a_{2M}a_{51}a_{61}a_{52}a_{62} \ldots a_{M-3,M}a_{M-2,M}$

File 2: $a_{31}a_{41}a_{32}a_{42} \ldots a_{3,M/2}a_{4,M/2} \ldots a_{3M}a_{4M}a_{71}a_{81}a_{72}a_{82} \ldots a_{M-1,M}a_{M,M}$

Step 2:

File 3: $a_{11}a_{21}a_{31}a_{41}a_{12}a_{22}a_{32}a_{42} \ldots a_{1,M/4}a_{2,M/4}a_{3,M/4}a_{4,M/4} \ldots a_{4,M}a_{91} \ldots a_{M-5,M}$

File 4: $a_{51}a_{61}a_{71}a_{81}a_{52}a_{62}a_{72}a_{82} \ldots a_{5,M/4}a_{6,M/4}a_{7,M/4}a_{8,M/4} \ldots a_{8,M}a_{13,1} \ldots a_{M,M}$

$\vdots$

by reading one record from each input file and writing out the data in the new order as two records on one of the output files, letting these alternate. The

transpose will obviously be obtained after $\lceil \log_2 M \rceil$ passes or after $M\lceil \log_2 M \rceil$ read and write operations.

Some slight modifications of the record sizes are necessary. In fact, we must increase (some of) the records so that at step i, $i=1,\ldots,\lceil \log_2 M \rceil - 1$, they contain $k2^i$ elements where k is an even number. This means that (some of) the input records at the last step contain $2^{\lceil \log_2 M \rceil}$ elements.

A closer look at this algorithm shows that the ith step actually corresponds to a sequence of perfect shuffles (see [2.11]) of pairs of arrays, whose elements are 2^{i-1} element subarrays. In this sense Knuth's algorithm is indeed similar to the one proposed by *Stone* [2.11], but the shuffling is performed at output and the rearranged arrays of step i are not created in high-speed storage during that step. This suggests a slight modification of the algorithm, in which the last step transposes 2×2 matrices with elements that are $2^{\lceil \log_2 M \rceil - 1} \times 1$ matrices. In this way we obtain the rows of the transpose in high-speed storage *before* they are written out, which is useful if the algorithm is applied in connection with, e.g., the FFT.

This algorithm can evidently be generalized to rectangular matrices using the method given in Sect. 2.2.3. This is an efficient method for transposing sequentially stored matrices. For instance, it is generally more efficient than the approach suggested by *Schumann* [2.14, 15] (see [2.6] for a proof).

Let us now instead suppose that A is stored on a random access device. The algorithm then has a generalization that works exactly like the square partition algorithm except in the following respect. Given $A^{(i-1)}$, still built up out of $1\times P_{i-1}$ partitions that are parts of the rows of $\tilde{A}$, the rows of $A^{(i)}$ are formed by writing out m_i consecutive columns of the submatrices in high-speed storage. The rows of $A^{(i)}$ will then consist of parts of the rows of $\tilde{A}$ of size P_i. Usually there is no need to append any dummy data at output.

Note that, as can be seen in the binary case, this method and the square partition method are not identical. However, they demand the same amount of resources, that is

$$\mathrm{RM_{RC}} = \mathrm{RM_S} = \max_{1\le i\le p} \{m_i N_{i-1} P_{i-1}\} \tag{2.30}$$

and

$$\mathrm{IO_{RC}} = \mathrm{IO_S} = M + 2\sum_{i=1}^{p-1} M_i P_i + N\,. \tag{2.31}$$

They differ, though, in some respects that sometimes may be important. In-place swapping is no longer as simple as transposition of square matrices. But generally we are not concerned about this until the last step, and then the two methods will again work in the same way (see the examples in Sect. 2.5). A more important point can be made about the requirements on external storage. Applied to square matrices stored as random access files, both algorithms can

work in-place on one single file. This is also true for the rectangular partition method presented below, which can then be forced to work with squares ($m_i = n_i$). On nonsquare matrices the three methods are no longer alike.

The original input file can always be a sequential file that is left unchanged. Aside from that, the square method can be applied in-place until step p, whereupon $\tilde{A}$ is stored in a second random access area. The rectangular method needs two random access areas to host $A^{(2i)}$ and $A^{(2i+1)}$, $i=0, 1, \ldots$. These files will moreover vary in size. The row-in/column-out method, finally, can be implemented like the square method. If, however, at the last step, the order in which the groups of rows of $A^{(p-1)}$ are processed is altered (an example is given in Sect. 2.6), the rows of $\tilde{A}$ can be generated sequentially. Furthermore, the method works on sequential files if $\max_{1 \leqq i \leqq p} \{m_i\}$ auxiliary files are available.

There is, of course, also a rectangular version of the algorithm presented in this section. We leave the details to the reader.

2.2.6 The Rectangular Partition Algorithm

The last algorithm we shall present was originally proposed by *Delcaro* and *Sicuranza* [2.13]. Their algorithm is built up of successive transpositions of partitioned matrices of order $m_i \times n_i$, where m_i and n_i are factors of M and N, respectively. Later, *Ramapriyan* [2.8] generalized the algorithm to the case in which $\bar{M} = m_1 \cdot \ldots \cdot m_p \geqq M$ and $\bar{N} = n_1 \cdot \ldots \cdot n_p \geqq N$, where p, $m_1, \ldots, m_p$ and $n_1, \ldots, n_p$ are determined by numerical optimization of the required computation time.

The algorithm works as follows. Let A be an $M \times N$ matrix and suppose that $\bar{M} = m_1 \cdot \ldots \cdot m_p \geqq M$ and $\bar{N} = n_1 \cdot \ldots \cdot n_p \geqq N$, where p is any integer, $p \geqq 2$. Set $m_0 = 1$, $n_0 = 1$, $M_0 = M$, $N_0 = N$ and define the following arrays of integers for $i = 0, 1, \ldots, p$.

$$\begin{aligned} P_i &= m_0 \cdot m_1 \cdot \ldots \cdot m_i \\ Q_i &= n_0 \cdot n_1 \cdot \ldots \cdot n_i \\ M_i &= \lceil M_{i-1}/m_i \rceil \\ N_i &= \lceil N_{i-1}/n_i \rceil . \end{aligned} \tag{2.32}$$

Then at step i, $i = 1, 2, \ldots, p$ the input matrix $A^{(i-1)}$, where $A^{(0)} = A$, is considered as an $M_{i-1} \times N_{i-1}$ matrix, whose elements are $Q_{i-1} \times P_{i-1}$ matrices which in fact are submatrices of $\tilde{A}$. This partitioned matrix is embedded in an $M_i \times N_i$ matrix from which $m_i \times n_i$ matrices are selectively read into high-speed storage, transposed and written out on $A^{(i)}$. The rows are chosen as in [2.13] that is Q_{i-1} rows apart. $A^{(i)}$ will then be built up by $Q_i \times P_i$ blocks and at step p an $\bar{N} \times \bar{M}$ matrix containing $\tilde{A}$ is generated.

Alternatively one can look upon $A^{(i-1)}$ as being built up by $1 \times P_{i-1}$ partitions, since the $Q_{i-1} \times P_{i-1}$ are never actually formed. The rows of $A^{(i)}$ will then contain $1 \times P_i$ partitions, which form parts of the rows of $\tilde{A}$.

If $\bar{M}=M$ and $\bar{N}=N$ the algorithm coincides with the algorithm of [2.13]. Otherwise, the matrix $A^{(i-1)}$ is at step i embedded in the minimal matrix for which the algorithm [2.13] works, given $m_1, \ldots, m_p$ and $n_1, \ldots, n_p$.

The required memory can be written as

$$\mathrm{RM_R} = \max_{1 \leqq i \leqq p} \{m_i N_{i-1} P_{i-1} + P_i N_i\}, \tag{2.33}$$

where the second term could be dropped except for $i=p$.

The number of input and output operations are

$$\mathrm{IO_R} = M + 2 \sum_{i=1}^{p-1} Q_i M_i + N. \tag{2.34}$$

2.3 Optimization of the Performance of the Algorithms

2.3.1 Two Lemmas

All the algorithms presented in the previous sections, except Floyd's binary oriented algorithm, involve an embedding of the original matrix in a larger matrix. We shall now discuss how these embeddings should be chosen in order to optimize the performance of the algorithms. First we need two crucial lemmas.

Let $\bar{M}$ and N be arbitrary integers, $\bar{M}>1$, $N>1$, and assume that for some $p>1$ $\bar{M}=m_1 \cdot \ldots \cdot m_p$, where $m_i>1$, $i=1, \ldots, p$. Define as before $m_0=1$, $p_i=m_0 \cdot \ldots \cdot m_i$, $i=0, 1, \ldots, p$ and $N_0=N$, $N_i=\lceil N_{i-1}/m_i \rceil$, $i=1, \ldots, p$. We then have

Lemma 1. The array $(N_i P_i)_{i=0}^{p}$ is increasing.

Proof. By definition $N_i=\lceil N_{i-1}/m_i \rceil \geqq N_{i-1}/m_i$, that is $N_i m_i \geqq N_{i-1}$. Hence, multiplication by P_{i-1} gives $N_i P_i \geqq N_{i-1} P_{i-1}$. Q.E.D.

This lemma simply states that the length of the rows increases in the square algorithm.

Lemma 2. $N/P_i \leqq N_i < N/P_i + 1$, $i=1, \ldots, p$.

Proof. We use induction on the number of factors of P_i. We have

$$N_1 = \begin{cases} N/m_1 \text{ if } m_1 \text{ is a divisor of } N \\ [N/m_1]+1 \text{ otherwise.} \end{cases}$$

In the first case the proposition is immediate, in the second we only need to note that $[N/m_1] < N/m_1$.

Assume then that the inequality is true for $i-1$ factors, that is

$$\frac{N}{P_{i-1}} \leqq N_{i-1} < \frac{N}{P_{i-1}} + 1 . \tag{2.35}$$

Write $N_{i-1} = q_i m_i + r_i$ where $0 \leqq r_i < m_i$. Then $q_i = [N_{i-1}/m_i]$ and

$$N_i = \begin{cases} q_i & \text{if } r_i = 0 \\ q_i + 1 & \text{if } 0 < r_i < m_i . \end{cases} \tag{2.36}$$

Using the left-hand side of (2.35) we find

$$N_i \geqq \frac{N_{i-1}}{m_i} \geqq \left(\frac{N}{P_{i-1}}\right) \Big/ m_i = \frac{N}{P_i} . \tag{2.37}$$

On the other hand, if $r_i = 0$, then

$$N_i = \frac{N_{i-1}}{m_i} < \left(\frac{N}{P_{i-1}} + 1\right) \Big/ m_i = \frac{N}{P_i} + \frac{1}{m_i} < \frac{N}{P_i} + 1 . \tag{2.38}$$

Furthermore, if $1 \leqq r_i < m_i$ then we may write

$$\begin{aligned} N_i = q_i + 1 &= \frac{N_{i-1}}{m_i} + \frac{m_i - r_i}{m_i} < \frac{N_{i-1}/P_{i-1} + 1}{m_i} + \frac{m_i - r_i}{m_i} \\ &= \frac{N}{P_i} + \frac{m_i - (r_i - 1)}{m_i} \leqq \frac{N}{P_i} + 1 . \quad \text{Q.E.D.} \end{aligned} \tag{2.39}$$

We immediately get the following corollary:

Corollary. Since N_i is an integer, N_i and $N_i P_i$ are invariant under re-factorization of $P_i = m_1 \cdot \ldots \cdot m_i$. In particular $m_1, \ldots, m_i$ can be permuted.

Let us now introduce some new and more general notation. Denote finite arrays of positive integers by $[\alpha]$, $[\beta]$, ... or by enumeration $(m_1, m_2, \ldots, m_p)$. For a given array $[\alpha] = (m_1, m_2, \ldots, m_p)$, write its subarrays $[\alpha_j] = (m_1, \ldots, m_j)$ and set $N_{[\alpha_0]} = N, \ldots, N_{[\alpha_j]} = \lceil N_{[\alpha_{j-1}]}/m_j \rceil$ and $P_{[\alpha_0]} = 1$, $P_{[\alpha_j]} = m_1 \cdot \ldots \cdot m_j$, $j = 1, \ldots, p$.

2.3.2 The Square Partition Algorithm

Set $R_{[\alpha_j]} = m_j N_{[\alpha_{j-1}]} P_{[\alpha_{j-1}]}$ and $R[\alpha] = \max\limits_{1 \leqq j \leqq p} \{R_{[\alpha_j]}\}$. We then have

Theorem 1. Let $\bar{M}$ and N be given integers, $\bar{M} > 1$, $N > 1$, and suppose that $\bar{M} = m_1 \cdot \ldots \cdot m_p$, where $m_1 \geqq m_2 \geqq \ldots \geqq m_p > 1$. Let $[\alpha] = (m_1, \ldots, m_p)$ and assume that $\pi[\alpha]$ is an arbitrary permutation of $[\alpha]$. Then $R[\alpha] \leqq R[\pi[\alpha]]$.

Let us before we prove the theorem make some observations. A look back at (2.25, 27) reveals that the theorem shows that the minimum memory requirements for the square partition methods, given the factorization, is obtained if the factors are ordered in a decreasing sequence. It is then assumed that the rows of $\tilde{A}$ are never formed (as consecutive arrays) in high-speed storage. But this requires at most $m_p N_p$ additional storage locations at step p (see [2.6]), and we shall see that the decreasing order will give the minimum then too. The theorem is also true if we consider $R'_{[\alpha_j]} = m_j N_j P_{[\alpha_j]}$, which is obtained if at each step we form the rows; in fact, a similar proof can be given.

The following examples show that there indeed is an order dependence.

Examples. If $N = 22$ and $\bar{M} = 5 \cdot 3 = 15$ we have $R_{(5)} = 110$ and $R_{(5,3)} = 75$, while $R_{(3)} = 66$ and $R_{(3,5)} = 120$, that is $R(3,5) = 120 > 110 = R(5,3)$. Setting $\bar{M} = 6 \cdot 5 = 30$ we see that $R(6,5) = 132$ while $R(5,6) = 150$, that is, this property does not depend on whether $\bar{M} < N$ or not. In these two examples, the maximum is attained for the maximal factor. This is not always so if $\bar{M} > N$, e.g., if $N = 22$ and $\bar{M} = 7 \cdot 6$ then $R_{(7)} = 154$ and $R_{(7,6)} = 168$. If $\bar{M} \leqq N$, however, $R_{\max}$ will always be attained for the last occurring maximum factor (see below). This proposition is however not true for the array $m_j N_j P_j$, $j = 1, \ldots, p$, obtained when swapping is done at each step.

Proof. Suppose that m_i is a factor for which the maximum $R[\alpha]$ is attained. Let $\pi[\alpha]$ be an arbitrary permutation of $[\alpha]$ and consider the shortest subarray at the beginning of $\pi[\alpha]$ containing all the elements $m_1, \ldots, m_i$. Denote this array by $[\bar{\beta}]$. If $[\bar{\beta}]$ has j elements, then $i \leqq j \leqq p$, and its last element is m_k, where $1 \leqq k \leqq i$. Let $[\beta]$ be the subarray obtained from $[\bar{\beta}]$ by deletion of m_k. Now reorder $[\bar{\beta}]$ to $[\bar{\gamma}]$ as follows. If $k = i$, let $[\bar{\gamma}] = [\bar{\beta}]$, but if $k < i$ $[\bar{\gamma}]$ is obtained from $[\bar{\beta}]$ by changing places between m_i and m_k. As before let $[\gamma]$ be the $j-1$-element subarray at the beginning of $[\bar{\gamma}]$.

We now observe that since $P_{[\gamma]} \geqq P_{[\beta]}$ it follows from Lemma 2 and its corollary that $N_{[\gamma]} \leqq N_{[\beta]}$. (Note that Lemma 2 limits N_i to an interval containing exactly one integer). Moreover, $m_i P_{[\gamma]} = m_k P_{[\beta]}$, so that $m_k P_{[\beta]} N_{[\beta]} \geqq m_i P_{[\gamma]} N_{[\gamma]}$.

Let $[\delta]$ be an array obtained from $[\gamma]$ by putting $m_1, \ldots, m_{i-1}$ at the beginning (in order). Then, again using the corollary of Lemma 2, we get $P_{[\delta]} N_{[\delta]} = P_{[\gamma]} N_{[\gamma]}$. Applying Lemma 1 to $[\delta]$ we then have

$$R[\pi[\alpha]] \geqq m_k P_{[\beta]} N_{[\beta]} \geqq m_i P_{[\gamma]} N_{[\gamma]} = m_i P_{[\delta]} N_{[\delta]} \geqq m_i P_{i-1} N_{i-1} = R[\alpha].$$

Q.E.D.

This theorem does not account for the extra storage locations needed to form the rows of $\tilde{A}$ at step p. This can be done with $m_p N_p$ additional words at step p (see [2.6]). We then have

Corollary. Theorem 1 is true also for the array $(T_i)_{i=1}^p$,

$$T_i = m_i P_{i-1} N_{i-1}, i = 1, \ldots, p-1; T_p = m_p P_{p-1} N_{p-1} + m_p N_p.$$

Proof. Let $m'_1, \ldots, m'_p$ be a reordering of $m_1, \ldots, m_p$ and denote the corresponding array to be maximized by $(T'_i)_{i=1}^p$. Suppose that $\max_{1 \leqq i \leqq p} T'_i = T'_j$. We then observe that $m'_p \geqq m_p$, according to the assumptions, so that $P'_{p-1} \leqq P_{p-1}$ and hence $N'_{p-1} \geqq N_{p-1}$. That is, since $P'_p = P_p = \bar{M}$ and $N'_p = N_p$ we get $T'_p = P'_p N'_{p-1} + m'_p N'_p \geqq P_p N_{p-1} + m_p N_p = T_p$. This proves the corollary, since

$$\max_{1 \leqq i \leqq p-1} T_i \leqq \max_{1 \leqq i \leqq p} T'_i \text{ according to the theorem.} \quad \text{Q.E.D.}$$

Next we prove the nonsurprising fact that the best factorization into $p+1$ factors requires less high-speed storage than any factorization into p factors. Define for a given $\bar{M}$

$$\Omega_p(\bar{M}) = \{[\alpha] = (m_1, \ldots, m_p); m_i > 1, m_1 \cdot \ldots \cdot m_p = \bar{M}\} \tag{2.40}$$

and set

$$S_p(\bar{M}) = \min\{R_{[\alpha]}; [\alpha] \in \Omega_p(\bar{M})\}. \tag{2.41}$$

Theorem 2. If $\bar{M}$ can be factored into $p+1$ nontrivial factors then $S_{p+1}(\bar{M}) \leqq S_p(\bar{M})$.

Proof. Let $[\alpha] = (m_1, \ldots, m_p)$ be an optimal factorization of $\bar{M}$ giving $S_p(\bar{M})$. At least one factor, say m_j, can then according to the assumptions be writen $m_j = st$, where s and t are integers $s > 1$, $t > 1$. Set $[\beta] = (m_1, \ldots, m_{j-1}, s, t, m_{j+1}, \ldots, m_p)$, $[\sigma] = (m_1, \ldots, m_{j-1}, s)$ and $[\tau] = (m_1, \ldots, m_{j-1}, s, t)$. From the corollary to Lemma 2 we know that $N_{[\tau]} = N_j$. Moreover, the same lemma shows that $N_{[\sigma]} \leqq N_{j-1}$. Since $sP_{j-1} < P_j$ we then get

$$\begin{aligned} S_{p+1}(\bar{M}) \leqq R_{[\beta]} &= \max\{P_1 N_0, \ldots, P_{j-1} N_{j-2}, sP_{j-1} N_{j-1}, P_j N_{[\sigma]}, \\ &\quad \cdot P_{j+1} N_j, \ldots, P_p N_{p-1}\} \\ &\leqq \max\{P_1 N_0, \ldots, P_{j-1} N_{j-2}, P_j N_{j-1}, P_{j+1} N_j, \ldots, P_p N_p^{-1}\} = S_p(\bar{M}). \end{aligned} \tag{2.42}$$

Q.E.D.

This theorem is still true if the memory requirements at step i are set to $m_i P_i N_i$. The proof is then somewhat more complicated.

What is missing now is a theorem characterizing the optimal p-factor solution. One result in this direction is given by the following theorem.

Let us introduce

$$r_p(\bar{M}) = \min\{\|[\alpha]\|_\infty; [\alpha] \in \Omega_p(\bar{M})\}, \tag{2.43}$$

where $\|(m_1, \ldots, m_p)\|_\infty = \max_{1 \leqq i \leqq p} \{m_i\}$. We then have

Theorem 3. Suppose that $\bar{M} \leqq N$. Then $R[\alpha] = S_p(\bar{M})$ only if $\|[\alpha]\|_\infty = r_p(\bar{M})$. In fact, if there are several elements in $\Omega_p(\bar{M})$ having minimum norm, then those with the fewest occurrences of this value, $r_p(\bar{M})$, will minimize $S_p(\bar{M})$.

Proof. According to Theorem 1, we only have to consider decreasing arrays. Let $[\alpha]=(m_1,\ldots,m_p)\in\Omega_p$. Then, since $\bar{M}\leqq N$, Lemma 2 gives

$$m_iP_{i-1}N_{i-1}<P_i(N/P_{i-1}+1)=m_iN+P_i\leqq(m_i+1)N. \tag{2.44}$$

This implies that $m_i=m_1$ must hold for the maximum. If $[\alpha]$ and $[\alpha']$ are two decreasing arrays in $\Omega_p(\bar{M})$ such that $r_p(\bar{M})=m_1<m_1'$ then evidently

$$m_iP_{i-1}N_{i-1}<(m_1+1)N\leqq m_1'N\leqq R[\alpha'] \tag{2.45}$$

so that $R[\alpha]<R[\alpha']$. The latter part of the theorem can be seen from (2.25) and Lemma 1, if the decreasing order is used. Q.E.D.

This theorem implies that the memory requirements (for fixed p) are minimum when the sizes of the factors vary as little as possible. It is, however, not true if $\bar{M}>N$. If, for instance, $\bar{M}=9\cdot9\cdot4\cdot4=6\cdot6\cdot6\cdot6$ and $N=243$ then $R(9,9,4,4)=2187$, whereas $R(6,6,6,6)=2592$. Nonetheless, the theorem can be used to simplify the search for the optimal embedding, as will be shown in Sect. 2.4. It can be noted that the theorem does not hold if one considers l^1-norms or the array $m_iP_iN_i$ (see [2.6]). One can also observe that even if $M<N$, the best factorization, $\bar{M}$, could satisfy $\bar{M}>N$.

Unfortunately, the required number of input and output operations [see (2.27)] will not be minimized if the factors are ordered in decreasing sequence. If, e.g., $M=52$ then $\bar{M}=5\cdot4\cdot3$, $3\cdot4\cdot5$, and $4\cdot5\cdot3$ give $\mathrm{IO_S}=52+230+N$, $52+228+N$, and $52+224+N$, respectively. If $\bar{M}=M$ then $\mathrm{IO_S}=(2p+1)M+N$ independently of the order of the factors. Generally, we know from Lemma 2 that $\bar{M}\leqq M_iP_i\leqq\bar{M}+P_i-1$, that is

$$M+N+2p\bar{M}\leqq\mathrm{IO}_{\mathrm{S}}^{(p)}\leqq M+N+2p\bar{M}+2\sum_{i=1}^{p-1}(P_i-1). \tag{2.46}$$

The required number of additional input/output operations caused by increasing M to $\bar{M}$ is, since $m_i>1$, thus not greater than

$$2\sum_{i=1}^{p-1}(P_i-1)=2\left[\frac{\bar{M}}{m_p}\left(\frac{1}{m_2\cdot\ldots\cdot m_{p-1}}+\ldots+\frac{1}{m_{p-1}}+1\right)-(p-1)\right]<2\left[2\frac{\bar{M}}{m_p}-(p-1)\right]. \tag{2.47}$$

This corresponds to less than $2/m_p$ of a data pass.

Using the fact that some rows contain dummy elements from step 2 on, one can of course decrease this number further. Such a procedure will, however, complicate the implementation of the algorithm.

A special case worth mentioning is the one in which $\bar{M}$ and N can be written $\bar{M}=m_1\cdot\ldots\cdot m_p$, $N=m_1\cdot\ldots\cdot m_{p-1}\cdot k$. This ordering will then result in exactly

$2(p-1)\bar{M}+N$ input/output operations, independently of the magnitude of the factors. If $k \geqq m_p$, that is if $N \geqq M$, no extra storage is needed at step p, even if the rows of $\tilde{A}$ are formed. From Theorem 3 it then follows that this ordering will require an optimal $\max_{1 \leqq i \leqq p} m_i N$ storage locations. At the same time the number of input/output operations needed will in general be less than what is obtained using the decreasing order. If $k < m_p$ the latter statement is still true, but the memory requirements will increase slightly.

Examples. Let $M=60$ and $N=72$. Then factoring $M=5\cdot 4\cdot 3=3\cdot 4\cdot 5$ gives in both cases a required memory of 360, while the numbers of input/output operations are 442 and 372, respectively. If, on the other hand, $N=24$, we get RM = 126 and 130, while IO = 214 and 180, respectively.

2.3.3 The Rectangular Partition Algorithm

We are now ready to look at the rectangular partition algorithm. This means that from now on $\bar{N}=n_1\cdot \ldots \cdot n_p$, $N_0=N$ and $N_i=\lceil N_{i-1}/n_i \rceil$. Disregarding the additional storage locations needed to form the rows at each step (also step p) we have

Theorem 4. If $(m_i)_{i=1}^p$ increases and $(n_i)_{i=1}^p$ decreases then $(P_i N_{i-1})_{i=1}^p$ cannot have a strict maximum in the interior. In particular, this ordering will minimize the memory requirements and satisfy

$$\max_{1 \leqq i \leqq p} \{P_i N_{i-1}\} \leqq \max \left\{ \min_{1 \leqq i \leqq p} \{m_i N\}, \min_{1 \leqq i \leqq p} \{n_i+1\}\bar{M} \right\}.$$

Proof. Assume that a strict maximum is obtained for $P_i N_{i-1}$, where $1<i<p$. Then $P_{i+1}N_i < P_i N_{i-1}$ so that according to the definition of N_i

$$0 < P_i N_{i-1} - P_{i+1} N_i = P_i (N_{i-1} - m_{i+1} N_i) \leqq P_i \left(N_{i-1} - m_{i+1} \frac{N_{i-1}}{n_i} \right)$$
$$= P_i N_{i-1} \left(1 - \frac{m_{i+1}}{n_i} \right), \tag{2.48}$$

that is $m_{i+1} < n_i$. The assumptions give $m_i \leqq m_{i+1} < n_i \leqq n_{i-1}$, implying $m_i + 1 \leqq n_{i-1}$ (since we are considering integers).

We also have $P_i N_{i-1} > P_{i-1} N_{i-2}$. From Lemma 2 we know $N_{i-1} < N_{i-2}/n_{i-1} + 1$, that is $(N_{i-1}-1)n_{i-1} < N_{i-2}$. This means that

$$0 < P_i N_{i-1} - P_{i-1} N_{i-2} = P_{i-1}(m_i N_{i-1} - N_{i-2})$$
$$< P_{i-1}\{m_i N_{i-1} - (N_{i-1}-1)n_{i-1}\} \leqq P_{i-1}\{m_i N_{i-1} - (N_{i-1}-1)(m_i+1)\}$$
$$= P_{i-1}(m_i - N_{i-1} + 1). \tag{2.49}$$

Since we deal with integers we conclude that $N_{i-1} \leqq m_i \leqq n_i$. But then $N_i = 1$, so that $P_{i+1}N_i = P_{i+1} = m_{i+1}P_i$, and hence since $N_{i-1} \leqq m_i$ we have $P_iN_{i-1} \leqq P_im_i \leqq P_im_{i+1} = P_{i+1}N_i$, contradicting the assumption.

We observe that if $P_{i+1}N_i < P_iN_{i-1}$, Lemma 2 will yield strict inequality in (2.49) even if $P_iN_{i-1} = P_{i-1}N_{i-2}$. This means that the array can have no interior plateaus, because then $P_{i+1}N_i < P_iN_{i-1}$ must hold for some i, $1 < i < p$.

Obviously, we always have $\max\limits_{1 \leqq i \leqq p} \{P_iN_{i-1}\} \geqq \max\{(m_1N, \bar{M}N_{p-1})\}$, with equality if the ordering proposed is used. Moreover, this bound is minimal if m_1 and N_{p-1} are minimal. But we have earlier noted that N_{p-1} is a minimum if Q_{p-1} is a maximum, that is if $n_p = \min\limits_{1 \leqq i \leqq p} \{n_i\}$. Finally we infer from Lemma 2 that

$$N_{p-1} < N/Q_{p-1} + 1 \leqq \bar{N}/Q_{p-1} + 1 = n_p + 1. \quad \text{Q.E.D.}$$

Examples. The following examples show that the array can have a minimum in the interior. Moveover, an interior maximum may occur if the order is reversed. Consequently there is no converse theorem giving an upper bound to the memory requirements.

(m_1, m_2, m_3)	(n_1, n_2, n_3)	N	(P_1N, P_2N_1, P_3N_2)
(2, 4, 6)	(4, 3, 2)	19	(38, 40, 96), increasing
(2, 3, 5)	(6, 4, 2)	47	(94, 48, 60), interior min.
(2, 3, 4)	(8, 5, 3)	115	(230, 90, 72), decreasing
(10, 9, 8)	(2, 3, 4)	19	(190, 900, 2160), increasing
(6, 4, 2)	(2, 3, 4)	19	(114, 240, 144), interior max.
(2, 3, 4)	(4, 5, 6)	119	(238, 180, 144), decreasing.

To create the rows of $\tilde{A}$ at step p, m_pN_{p-1} additional storage locations are needed if $m_p > N_{p-1}$ (see [2.6]).

The orderings of Theorem 4 will then no longer guarantee a minimum. If, for instance, $N = 19$, $(n_1, n_2, n_3) = (4, 3, 2)$, then the required number of storage locations is $96 + 12 = 108$ if $(m_1, m_2, m_3) = (2, 4, 6)$ but $96 + 8 = 104$ if $(m_1, m_2, m_3) = (2, 6, 4)$. From Theorem 4, since $N_{p-1} \leqq n_p$, we can conclude that, for given $\bar{M}$ and N,

$$\min\{\mathrm{RM_R}\} \leqq \max\{\underline{m}N, (\underline{n}+1)\bar{M} + \bar{m}\underline{n}\} \tag{2.50}$$

where

$$\underline{m} = \min_{1 \leqq i \leqq p} \{m_i\}, \quad \bar{m} = \max_{1 \leqq i \leqq p} \{m_i\} \text{ and } \underline{n} = \min_{1 \leqq i \leqq p} \{n_i\}.$$

An important property of the rectangular partition method is that there is a direct trade-off between the number of memory allocations and the input and output required. This is seen from the following theorem.

Theorem 5. The sum $M+2\sum_{i=1}^{p-1} M_iQ_i+N$ is a minimum if $(m_i)_{i=1}^p$ is decreasing and $(n_i)_{i=1}^p$ increasing, and a maximum if $(m_i)_{i=1}^p$ is increasing and $(n_i)_{i=1}^p$ decreasing.

Proof. We prove the first part by induction on p. Let $p=2$ and $m_1 \geqq m_2$, $n_1 \leqq n_2$. Introduce $M_1^{(1)}=\lceil M/m_1 \rceil$ and $M_1^{(2)}=\lceil M/m_2 \rceil$. Evidently $M_1^{(1)} \leqq M_1^{(2)}$ so we get $M_1^{(1)}n_1 \leqq M_1^{(1)}n_2$, $M_1^{(1)}n_1 \leqq M_1^{(2)}n_1 \leqq M_1^{(2)}n_2$, proving the assertion for $p=2$.

Now suppose the assertion is true for $p-1$ factors and consider arbitrary factorizations $\bar{M}=m_1 \cdot \ldots \cdot m_p$, and $\bar{N}=n_1 \cdot \ldots \cdot n_p$. First reorder $m_1, \ldots, m_{p-1}$ to $m_1' \geqq \ldots \geqq m_{p-1}'$ and $n_1, \ldots, n_{p-1}$ to $n_1' \leqq \ldots \leqq n_{p-1}'$ and put $m_p'=m_p$ and $n_p'=n_p$. Then, according to Lemma 2, $M_{p-1}'=M_{p-1}$. Obviously $Q_{p-1}'=Q_{p-1}$, so that the induction assumption gives $\sum_{i=1}^{p-1} M_i'Q_i' \leqq \sum_{i=1}^{p-1} M_iQ_i$.

If $m_p' \leqq m_{p-1}'$ the m_i''s are now decreasing. Otherwise set $m_p''=m_{p-1}'$ and $m_{p-1}''=m_p'$ and $m_i''=m_i'$, $i=1, \ldots, p-2$. This gives $M_{p-1}'' \leqq M_{p-1}'$ while $M_i''=M_i'$, $i=1, \ldots, p-2$, so $\sum_{i=1}^{p-1} M_i''Q_i' \leqq \sum_{i=1}^{p-1} M_i'Q_i'$. Finally, if $(m_i'')_{i=1}^{p-1}$ is not decreasing we can again apply the theorem for $(p-1)$ factors to obtain $m_1''' \geqq \ldots \geqq m_{p-1}'''$, $m_p''' \geqq m_p''$, where $\sum_{i=1}^{p-1} M_i'''Q_i' \leqq \sum_{i=1}^{p-1} M_i''Q_i'$.

Correspondingly, if $n_p' < n_{p-1}'$ set $n_i''=n_i'$, $i=1, \ldots, p-2$, $n_{p-1}''=n_p'$ and $n_p''=n_{p-1}'$. Then $Q_i''=Q_i'$, $i=1, \ldots, p-2$ and $Q_{p-1}'' < Q_{p-1}'$, so $\sum_{i=1}^{p-1} M_i'''Q_i'' < \sum_{i=1}^{p-1} M_i'''Q_i'$. Using the assertion for $(p-1)$ factors once more we get $n_1''' \leqq \ldots \leqq n_{p-1}'''$ and $n_p'''=n_p''$, where $\sum_{i=1}^{p-1} M_i'''Q_i''' \leqq \sum_{i=1}^{p-1} M_i'''Q_i'' \leqq \sum_{i=1}^{p-1} M_iQ_i$ and $(m_i''')_{i=1}^p$ is decreasing and $(n_i''')_{i=1}^p$ is increasing.

The second part of the theorem follows from a similar argument. Q.E.D.

2.3.4 On the Advantages of Inserting the Factor 1

In the theorems above concerning the square partition algorithm, we have assumed that the factors $m_1, \ldots, m_p$ are all greater than 1. This assumption is in fact not necessary. It is easy to see that if any unit factor is deleted, the required memory will be the same, while the number of input/output operations will decrease.

The rectangular partition algorithm does not possess this property unless the unit factor occurs in the interior of the array. We have more precisely

Proposition 2. Let $\bar{M}=m_1 \cdot \ldots \cdot m_p$, and $\bar{N}=n_1 \cdot \ldots \cdot n_p$. Then, if $m_k=1$, for some k, $1<k \leqq p$, there exist factorizations $m_1' \cdot \ldots \cdot m_{p-1}'=\bar{M}$ and $n_1' \cdot \ldots \cdot n_{p-1}'=\bar{N}$ for which

$$\max_{1 \leqq i \leqq p-1} \{P_i'N_{i-1}'\} \leqq \max_{1 \leqq i \leqq p} \{P_iN_{i-1}\} \quad \text{and} \quad \sum_{i=1}^{p-2} M_i'Q_i' \leqq \sum_{i=1}^{p-1} M_iQ_i.$$

The same assertion can be made if $n_k=1$ and $1 \leqq k < p$.

Proof. If $m_k=1$ the new factorizations are given by $m'_i=m_i$, $i=1,\ldots,k-1$, $m'_i=m_{i+1}$, $i=k,\ldots,p-1$, $n'_i=n_i$, $i=1,\ldots,k-2$, $n'_{k-1}=n_k n_{k-1}$, $n'_i=n_{i+1}$, $i=k,\ldots,p-1$. Because $N'_{k-1}=N_k$ the memory requirements will then be $\max_{\substack{1\leqq i\leqq p\\ i\neq k}}\{p_i N_{i-1}\}$. In the sum for the number of input/output operations the term $M_k Q_k$ will be deleted.

If $n_k=1$ we set $m'_k=m_k m_{k+1}$ and keep the other factors to get the same result. Q.E.D.

2.4 Optimizing the Square Partition and the Row-in/Column-out Algorithms

The results of Sect. 2.3.2 can now be used to help us find the optimal way to transpose an arbitrary matrix with the square partition algorithm or the row-in/column-out algorithm.

More precisely, we shall minimize the memory requirements for a given number of data passes. This does not always minimize the number of IO operations, but the deviation from the minimum is small (see Sect. 2.3.2). Moreover, if the optimum with respect to memory is obtained without insertion of zero rows ($\bar{M}=M$), then the IO requirements are also minimized (Sect. 2.2.3). Finally, since the number of data passes needed for an $M\times N$ matrix is bounded above by $\lceil\log_2 M\rceil$ ($M\leqq N$) or $\lceil\log_2 N\rceil+1$ ($M>N$), the (approximate) optimum with respect to memory *and* IO can be found, given the relative costs of storage and IO operations.

Consider first the case $M\leqq N$. Then, for given p, we can proceed as follows:

1) Set $m=\lceil M^{1/p}\rceil$. (Then $m=\min\{k;k^p\geqq M\}$.)

2) Find the smallest product $\bar{M}=m_1\cdot\ldots\cdot m_p\geqq M$, with $m_1\geqq\ldots\geqq m_p$, using $m_1=m$ because $(m-1)^p<M$ according to 1.

3) Compute the number of high-speed locations needed, $\mathrm{RM_S}(\bar{M})$, according to (2.25).

4) Search for a better product $\bar{M}'=m'_1\cdot\ldots\cdot m'_p\geqq M$, $m'_1\geqq\ldots\geqq m'_p$, for which $m'_1<\mathrm{RM_S}(\bar{M})/N$. [We only have to consider these products since $\mathrm{RM_S}(\bar{M}')\geqq m'_1 N$]. Note that $m'_1\geqq m_1$ because $\bar{M}'\geqq\bar{M}$.

5) If such an $\bar{M}'$ is found, replace $\bar{M}$ with $\bar{M}'$ and repeat 4.

The set of products in 4 is preferably searched in (increasing) lexicographic order. Moreover, from the choice of $\bar{M}$ and Theorem 3 it also follows that we only have to consider products $\bar{M}'>N$; that is, the search in 4 is over the set

$$\Omega=\{M'=m'_1\cdot\ldots\cdot m'_p;m'_1\geqq\ldots\geqq m'_p,\bar{M}>N,m'_1\leqq\mathrm{RM_S}(\bar{M})/N\}\,. \tag{2.51}$$

The case $M>N$ is slightly more complicated. First, one considers the matrix as partitioned into $N\times N$ matrices and computes the optimal p product, $\bar{M}$, for these. If $\bar{M}\geqq M$ one has the best p-step factorization. If, on the other hand $\bar{M}<M$, an additional step is needed to concatenate the rows into rows with M

Table 2.1. Memory requirements for transposition of a 620×1000 matrix. $RM_S(\bar{M})$ is always optimal

p	$\bar{M}$	$RM_S(\bar{M})$	Ω
2	25^2	25,000	$\emptyset$
3	$9^2 \cdot 8$	9,072	$\emptyset$
4	5^4	5,000	$\emptyset$
5	$4^4 \cdot 3$	4,096	$\{4^5\}$
6	3^6	3,645	$\emptyset$
7	$3^4 \cdot 2^3$	3,078	$\{3^{7-k}2^k; k=0, \ldots 2\}$
8	$3^3 \cdot 2^5$	3,024	$\{3^{8-k2k}; k=0, \ldots, 4\}$
9	$3 \cdot 2^8$	3,000	$\{3^{9-k}2^k; k=0, \ldots, 7\}$
10	2^{10}	2,048	$\emptyset$

elements. Discarding all zero elements one then needs $\max\{M, RM_S(\bar{M})\}$ storage locations for the $(p+1)$ step transposition. However, a better result can possibly be obtained if one directly applies a $(p+1)$ step transposition.

For instance, if $M=27$, $N=25$, and $p=2$, then $\bar{M}=5 \cdot 5$ and 125 storage locations are needed using 3 passes. But a 27×25 matrix can be transposed in 3 steps using 81 storage locations ($M=3 \cdot 3 \cdot 3$). Observe that by Theorem 2, transpositions of the $N \times N$ matrices in $(p-1)$ steps followed by concatenation of rows never can give better results if $\bar{M} \geqq M$ above.

In Table 2.1 we give an example of how this method works for a 620×1000 matrix.

2.5 An Example

To illustrate how the 4 algorithms presented in Sects. 2.2.3–6 work we apply them to a 6×6 matrix.

1) Floyd's method. RM = 12, IO = 24. Note that $A^{(1)}$, $B^{(0)}$, and $B^{(3)}$ are not constructed.

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 53 & 55 \end{bmatrix}$$

a) $A = A^{(0)}$

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ 50 & 51 & 52 & 53 & 54 & 55 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 10 & 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

b) $A^{(1)}$, where $a^{(1)}_{i,j} = a_{-i,j}$

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ 51 & 52 & 53 & 54 & 55 & 50 \\ 42 & 43 & 44 & 45 & 40 & 41 \\ 33 & 34 & 35 & 30 & 31 & 32 \\ 24 & 25 & 20 & 21 & 22 & 23 \\ 15 & 10 & 11 & 12 & 13 & 14 \end{bmatrix}$$

c) $B^{(0)}$, where $b^{(0)}_{i,i+j} = a^{(1)}_{i,j}$

$$\begin{bmatrix} 00 & 10 & 02 & 12 & 04 & 14 \\ 51 & 01 & 53 & 03 & 55 & 05 \\ 42 & 52 & 44 & 54 & 40 & 50 \\ 33 & 43 & 35 & 45 & 31 & 41 \\ 24 & 34 & 20 & 30 & 22 & 32 \\ 15 & 25 & 11 & 21 & 13 & 23 \end{bmatrix}$$

d) $B^{(1)}$, columns with $j_0 = 1$ shifted 1 step

$$\begin{bmatrix} 00 & 10 & 20 & 30 & 04 & 14 \\ 51 & 01 & 11 & 21 & 55 & 05 \\ 42 & 52 & 02 & 12 & 40 & 50 \\ 33 & 43 & 53 & 03 & 31 & 41 \\ 24 & 34 & 44 & 54 & 22 & 32 \\ 15 & 25 & 35 & 45 & 13 & 23 \end{bmatrix}$$

e) $B^{(2)}$, columns with $j_1 = 1$ shifted 2 steps

$$\begin{bmatrix} 00 & 10 & 20 & 30 & 40 & 50 \\ 51 & 01 & 11 & 21 & 31 & 41 \\ 42 & 52 & 02 & 12 & 22 & 32 \\ 33 & 43 & 53 & 03 & 13 & 23 \\ 24 & 34 & 44 & 54 & 04 & 14 \\ 15 & 25 & 35 & 45 & 55 & 05 \end{bmatrix}$$

f) $B^{(3)}$, columns with $j_2 = 1$ shifted 4 steps

$$\begin{bmatrix} 00 & 10 & 20 & 30 & 40 & 50 \\ 01 & 11 & 21 & 31 & 41 & 51 \\ 02 & 12 & 22 & 32 & 42 & 52 \\ 03 & 13 & 23 & 33 & 43 & 53 \\ 04 & 14 & 24 & 34 & 44 & 54 \\ 05 & 15 & 25 & 35 & 45 & 55 \end{bmatrix}$$

g) A^1, where $a'_{i,j} = b^{(3)}_{i,i+j}$

2) The square method: $\bar{M} = 3 \cdot 2$, RM = 18, IO = 24. We have indicated how the rows are grouped together and shown how one group of rows maps in step 2.

$$\left[\begin{array}{cccccc} 00 & 01 & 02 & 03 & 04 & 05 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \end{array}\right] \rightarrow \left[\begin{array}{cccccc} 00 & 10 & 20 & 03 & 13 & 23 \\ 01 & 11 & 21 & 04 & 14 & 24 \\ 02 & 12 & 22 & 05 & 15 & 25 \\ 30 & 40 & 50 & 33 & 43 & 53 \\ 31 & 41 & 51 & 34 & 44 & 54 \\ 32 & 42 & 52 & 35 & 45 & 55 \end{array}\right] \rightarrow \left[\begin{array}{cccccc} 00 & 10 & 20 & 30 & 40 & 50 \\ & & & - & & \\ & & & - & & \\ 03 & 13 & 23 & 33 & 43 & 53 \\ & & & - & & \\ & & & - & & \end{array}\right]$$

3) The rectangular method: $\bar{M} = 2 \cdot 3$, $\bar{N} = 3 \cdot 2$, RM = 18 (RM = 12 also possible, see [2.6]), IO = 30.

$$\left[\begin{array}{cccccc} 00 & 01 & 02 & 03 & 04 & 05 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \end{array}\right] \rightarrow \left[\begin{array}{cccc} 00 & 10 & 03 & 13 \\ 01 & 11 & 04 & 14 \\ 02 & 12 & 05 & 15 \\ 20 & 30 & 23 & 33 \\ 21 & 31 & 24 & 34 \\ 22 & 32 & 25 & 35 \\ 40 & 50 & 43 & 53 \\ 41 & 51 & 44 & 54 \\ 42 & 52 & 45 & 55 \end{array}\right] \rightarrow \left[\begin{array}{cccccc} 00 & 10 & 20 & 30 & 40 & 50 \\ & & & & & \\ & & & & & \\ 03 & 13 & 23 & 33 & 43 & 53 \\ & & & & & \\ & & & & & \end{array}\right]$$

4) The rectangular method: $\bar{M}=3\cdot 2$, $\bar{N}=2\cdot 3$, RM = 18, IO = 20.

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \end{bmatrix} \rightarrow \begin{bmatrix} 00 & 10 & 20 & 02 & 12 & 22 & 04 & 14 & 24 \\ 01 & 11 & 21 & 03 & 13 & 23 & 05 & 15 & 25 \\ 30 & 40 & 50 & 32 & 42 & 52 & 34 & 44 & 54 \\ 31 & 41 & 51 & 33 & 43 & 53 & 35 & 45 & 55 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 00 & 10 & 20 & 30 & 40 & 50 \\ & & & - & & \\ 02 & 12 & 22 & 32 & 42 & 52 \\ & & & - & & \\ 04 & 14 & 24 & 34 & 44 & 54 \\ & & & - & & \end{bmatrix}$$

5) The row-in/column-out method: $\bar{M}=3\cdot 2$, RM = 18, IO = 24.

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \end{bmatrix} \rightarrow \begin{bmatrix} 00 & 10 & 20 & 01 & 11 & 21 \\ 02 & 12 & 22 & 03 & 13 & 23 \\ 04 & 14 & 24 & 05 & 15 & 25 \\ 30 & 40 & 50 & 31 & 41 & 51 \\ 32 & 42 & 52 & 33 & 43 & 53 \\ 34 & 44 & 54 & 35 & 45 & 55 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 00 & 10 & 20 & 30 & 40 & 50 \\ 01 & 11 & 21 & 31 & 41 & 51 \\ & & & - & & \\ & & & - & & \\ & & & - & & \\ & & & - & & \end{bmatrix}$$

2.6 Anderson's Direct Method for Computing the FFT in Higher Dimensions

An alternative solution to the entire problem of computing the DFT of a matrix stored rowwise on a random access device is suggested by *Anderson* [2.1]. Because the outer sum, (2.2b), is a DFT for a fixed l, one can obviously use the FFT to compute it. The basic idea behind the FFT is that the DFT of an array can be computed as an iterated sum, where each sum in the iteration is itself a DFT. If, in particular, the array size, M, is a power of 2, each partial sum can be a 2 point DFT. Then (see, e.g., [2.16] or [2.17]) at step i, $i=1, \ldots, \log_2 M$, one transforms the two points with indices m and m', for which the binary

expansions differ exactly at position i. The final array will then appear in bit-reversed order, but the initial array can be rearranged so that the resulting values are in natural order. (See also [2.18].)

Observing that the sum in (2.2b) is computed for all l, we can proceed as follows. At step i, the row pairs for which the indices differ precisely at binary position i are successively read into high-speed storage, where the two point DFT is computed in place for all the columns. These modified rows are then written back to their old positions. After $\log_2 M$ steps we obtain the desired transform, but with the bit-reversed order. This is no problem because one can use a proper order key at input or during future processing. Irrespective of how the bit reversal is handled, one can observe that the addressing of the rows is completely analogous to the addressing in the square partition method (and in Floyd's algorithm) for transposition.

The analogy between the two methods extends to the possible generalizations that exist. In both cases, one can reduce the the IO time by treating more than two rows simultaneously. In fact, if the matrix contains $M \times N$ elements and one can store 2^m rows in high-speed memory, then the number of IO operations will be

$$IO_A = 2 \cdot M \cdot \left\lceil \frac{\log_2 M}{m} \right\rceil. \tag{2.52}$$

If M and N are powers of 2, then IO_A coincides with IO_S and IO_F. Furthermore, the method generalizes to arbitrary factorizations of M in the same manner as the FFT and the analogy with the transposition methods described in Sect. 2.2 still holds. Consequently, the performances measured in terms of IO operations are the same in this case as well.

The direct approach is conceptually simple, in that it is based on properties of the fast algorithm (e.g., the FFT) itself. It also has the advantage that it works in place on arbitrary matrices. However, there are also disadvantages with the method, that will be discussed in the next section.

2.7 Discussion

Two approaches for computing separable 2-D transforms for matrices accessible row- or columnwise have been presented. One of the methods is based on matrix transposition, the other on properties of "fast" transforms, such as the FFT. The methods have been shown to be essentially equivalent in performance when the matrix dimensions are highly composite numbers. (We assume that no actual transposition is done in high-speed storage, so that the processing times are approximately the same.) In such cases the direct approach can be used for in place transformation of arbitrary matrices. It is, however, less modular in its design and therefore the more general matrix transposition

approach often gives better performance in other cases, as will be shown shortly. The modularity of the transposition method also implies that it can be combined with the use of fast transform hardware.

The advantage of the modularity is especially evident in the following case. Suppose that we want to compute the FFT of an $M \times N$ matrix and that M contains a large prime factor. The summations in the FFT will then be the same for the two methods, but the required amount of high-speed storage may be drastically reduced if transposition is done using a suitable embedding. Padding the arrays with zeros in the FFT computations in the direct method may not be desirable, because it changes the transform values. Consider for instance, the example in Table 2.1. The direct approach will require 2 data passes and 31,000 high-speed storage locations, which can be compared to the 25,000 locations needed to transpose in 2 passes. (Observe that the 1-D FFT computations can be intermingled with the transposition so that 2 passes are sufficient.) Futhermore, Table 2.1 shows that the transposition method can be applied with much less memory, at the expense of increased IO time. Conversely, there are also cases where memory can be traded for IO time.

Let us mention some other, less important differences between the methods. Assume that we want to compute the transform and the inverse transform of an $M \times N$ matrix and that M and N are highly composite, so that nothing is gained by padding the matrix with zeros. Assume that KN elements fit into high-speed storage and that KN/M is integer, then,

$$IO_S = 2\left(\left\lceil \frac{\log_2 M}{\log_2 K} \right\rceil M + \left\lceil \frac{\log_2 N}{\log_2 KN/M} \right\rceil N\right) \tag{2.53}$$

and

$$IO_A = 4\left\lceil \frac{\log_2 M}{\log_2 K} \right\rceil M. \tag{2.54}$$

It can be shown from these expressions that $IO_S < IO_A$ if $M < N$ and that $IO_S > IO_A$ if $M > N$. If, for example, $M = 2^{10}$, $N = 2^8$ and $K = 2^3$ then $IO_S = 48 \cdot 2^8$ whereas $IO_A = 64 \cdot 2^8$, but if $M = 2^8$ and $N = 2^{10}$ then $IO_S = 22.2^8$ and $IO_A = 12 \cdot 2^8$. It can also be noted that the minimum amount of storage required to compute a transform pair, regardless of the IO time, is smaller for the direct approach if $M > N$.

The four algorithms for matrix transposition that have been presented are all optimal in the case of a matrix whose dimensions are powers of 2. In this case they can also all be implemented using shift and masking operations only. However, they differ slightly when applied to arbitrary rectangular matrices.

Floyd's algorithm can handle any square matrix using only shifts and masking and extends to rectangular matrices in the manner described in Sect. 2.2.3. However, the other three algorithms do not have this "binary" property,

because they utilize expansions of the form (2.15). On the other hand they can perform the transposition in fewer IO operations, using slightly more memory.

These three algorithms also require an optimization step, which for the square partition and the row-in/column-out algorithms is very simple, but somewhat more complicated for the rectangular partition algorithm.

The latter algorithm is also conceptually more complex, because the number of rows as well as the row lengths vary during the transposition, even when the matrix dimensions are composite numbers.

In general, the square partition and the row-in/column-out algorithms, which are simpler, give (almost) as good performance. They are therefore preferable, unless the implementation has to be done with shifts and maskings, when *Floyd*'s algorithm can be used.

References

2.1 G.L.Anderson: IEEE ASSP-**28**, 280–284 (1980)
2.2 N.M.Brenner: IEEE AU-**17**, 128–132 (1969)
2.3 B.R.Hunt: Proc. IEEE **60**, 884–887 (1972)
2.4 R.C.Singleton: IEEE AU-**15**, 91–98 (1967)
2.5 H.L.Buijs: Appl. Opt. **8**, 211–212 (1969)
2.6 J.O.Eklundh: "Efficient Matrix Transposition with Limited High-Speed Storage"; FOA Reports, Vol. 12, No. 1, National Defence Research Institute, Stockholm, Sweden (1978)
2.7 J.O.Eklundh: IEEE C-**21**, 801–803 (1972)
2.8 H.K.Ramapriyan: IEEE C-**24**, 1221–1226 (1976)
2.9 R.E.Twogood, M.P.Ekstrom: IEEE C-**24**, 950–952 (1976)
2.10 R.W.Floyd: "Permuting Information in Idealized Two-Level Storage", in *Complexity of Computer Computations*, ed. by R.E.Miller, J.W.Thatcher (Plenum Press, New York 1972) pp. 105–109
2.11 H.S.Stone: IEEE C-**20**, 153–161 (1971)
2.12 D.E.Knuth: *The Art of Computer Programming*, Vol. 3 (Addison-Wesley, Reading, MA 1973) pp. 7, 573
2.13 L.G.Delcaro, G.L.Sicuranza: IEEE C-**23**, 967–970 (1974)
2.14 U.Schumann: Angew. Informatik **14**, 213–216 (1972)
2.15 U.Schumann: IEEE C-**22**, 542–543 (1973)
2.16 J.W.Cooley, J.W.Tukey: Math. Comput. **19**, 297–301 (1965)
2.17 W.T.Cochran et al.: IEEE AU-**15**, 45–55 (1967)
2.18 G.E.Rivard: IEEE ASSP-**25**, 250–252 (1977)