Minimizing Page Fetches for Permuting Information in Two-Level Storage

Part 2. Design of the Algorithm for Arbitrary Permutations

TAKAO TSUDA* and KEN-ICHI NAKAGAWA*

In light of the generalization of Floyd's model studied in Part 1 of this paper, an algorithm for arbitrary permutations of records between pages of slow memory is constructed. Bounds of page fetches required are estimated; for the extreme case of 'pure' transposition, the algorithm can do with the same number of page fetches as that considered the best possible. Some important applications are discussed.

1. Statement of the Problem

The theoretical framework of this paper has been studied in Part 1, where, using Floyd's idealized model, one finds how to compute the lower bound of the number of page fetches required for a given permutation of records between pages in slow memory.

Consider p' pages, each filled with p records and designate the objective permutation by a $p' \times p'$ matrix $(X(i,j))$, hereafter called an X-matrix. All the records are assumed to have, or bounded by, the same length. The element $X(i,j)$ is either a zero or a positive integer and indicates that those records, $X(i,j)$ in number, of the current i^{th} page should have been moved to the j^{th} page when the final distribution of records is established. The exchange of records between pages of slow memory is a stepwise process, because the number of pages that can reside in main memory at a time is limited to w. The stepwise inter-page data transfer corresponds to the stepwise transformation of the X-matrix.

Given an arbitrary initial distribution of records $(X(i,j))$ $(\equiv X_{\text{initial}})$, we try to have the final distribution of records X_{final}, namely,

$$(X(i,j)) = \begin{pmatrix} p & & & 0 \\ & p & & \\ & & \ddots & \\ 0 & & & p \end{pmatrix} \equiv X_{\text{final}}. \tag{1}$$

The purpose of this paper is to construct an algorithm which disposes the above problem of external permutations with as small a number of page fetches as possible. The algorithm devised can in fact do with the least possible page fetches for the limiting case of full transposition; hence it is best possible. For other cases, we can give the close bounds of page fetches actually required.

Remember that at each step of transformation there is the constraint that

$$\sum_{i=1}^{p'} X(i,j) = \sum_{j=1}^{p'} X(i,j) = p. \tag{2}$$

We first consider the case where $p' = p$, and later the results are extended to cover the case where $p' \neq p$.

In the case of transposition, all the records are shuffled in a regular and foreseeable way as the algorithm dictates. In the 'arbitrary' permutations, however, which will be discussed in this paper, the information of the changing X-matrix need be retained in memory—in a *page-transfer management table.*

In the Floyd model studied in Part 1, successive w 'operations' to yield a set of new w pages in slow memory finally destroy the original w pages they copy. In actual algorithms, however, one may have some of the original copies remain resident for the next new w-page formation. This means that the number of page fetches can be less than the number of operations. Distinct from Part 1, the main concern of Part 2 is the number of page fetches actually required, and it is evaluated by actually constructing the relevant algorithm.

2. The Algorithm for Arbitrary Permutations: p Pages, Each with p Records

As already stated, we first consider p pages, each with p records. The X-matrix is then a $p \times p$ square matrix. Depending on the relationships between parameters p and w, there are the following cases.

A. p is some power of w.

B. p is not a power of w.

Let k, l, w_0 and x denote some positive integers. Case B is further divided into the next three cases:

B–1. $p = k \cdot w^l$ and $k = w_0^x$,
where $2 \leq w_0 < w$ and $k \geq 2$.
Note that case B–1 assumes $w \geq 3$; if $w = 2$, then the case reduces to case B–2 shown next.

B–2. $p = k \cdot w^l$ and $k \neq w_0^x$.
Namely, k and w are mutually prime and $k > w$. This latter condition holds because, if either $k = w$ or $k < w$, this case is included in case A or case B–1 (where $k = w_0$), respectively.

*Department of Information Science, Kyoto University, Kyoto, Japan.

B–3. $p \neq k \cdot w^l$.
Namely, p and w are mutually prime.

We first construct the algorithm for case A, and all the other cases will be covered by modifying this algorithm for case A.

Case A—p is Some Power of w.

Processing is done over $\log_w p$ passes. One pass consists of a preprocessing plus block-diagonalization of the X-matrix. The algorithm is used recursively; parameter s is first initialized as

$$s \leftarrow p.$$

By updating the value of s as

$$s \leftarrow s/w$$

at the end of each pass, the recursion finishes on arriving at $s=w$. The number of passes is thus $\log_w p$.

In the first pass we first initialize as

$$s \leftarrow p.$$

Preprocessing. s ($=p$ for the first pass) columns of the X-matrix is divided into w parts. The first through the s/wth columns are called block # 1, the $(s/w+1)$st through the $2s/w$th columns are called block # 2, $\cdots$, the $[(w-1)s/w+1]$st through the sth columns block # s/w. See Fig. 1. When processing block # 1, the rest of the blocks that are to the right of block # 1, i.e., blocks # 2, # 3, $\cdots$, and # w, are collectively called block # 1′. Similarly, when processing block # 2, those blocks that are to the right of block # 2, i.e., blocks # 3, # 4, $\cdots$, # w, are called block # 2′.

The elements in the ith row of block # j ($j=1, 2, \cdots, w$) are defined to make a *partial row*, their sum given by

$$X\left(i, (j-1)\frac{s}{w}+1\right)+X\left(i, (j-1)\frac{s}{w}+2\right) + \cdots + X\left(i, (j-1)\frac{s}{w}+\frac{s}{w}\right) \equiv S(i,j) \quad (i=1, 2, \cdots, p; j=1, 2, \cdots, w). \tag{3}$$

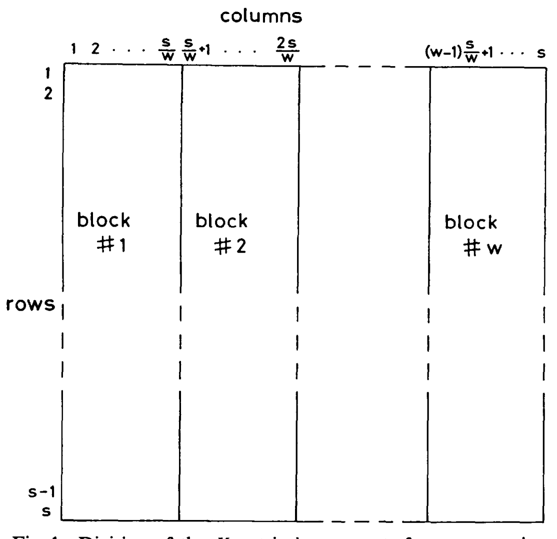

Fig. 1 Division of the X-matrix into w parts for preprocessing.

We call this a *partial sum*. Considering all the partial rows belonging to a block, we therefore have

$$\sum_{i=1}^{s} S(i,j)=ps/w. \tag{4}$$

The purpose of the present preprocessing is to redistribute the elements of X_{initial}, hence to shuffle records between pages correspondingly, in such a way that

$$S(i,j)=p/w \tag{5}$$

for all i and j. Our algorithm is such that (5) is first realized in block # 1, and then in block # 2, $\cdots$; in the end all the blocks meet the equi-distribution condition of (5).

Consider first the processing of block # 1 (i.e., $j=1$ in (5)). Two out of s pages are fetched in turn. If page i' such that

$$S(i', 1)=p/w-\alpha \quad (\alpha>0)$$

exists, then there is always a complementary page i'' such that

$$S(i'', 1)=p/w+\beta \quad (\beta>0)$$

($i' \neq i''$). Fetching these two complementary pages, let page i' be such that

$$S(i', 1)=p/w.$$

Namely, α records of page i'' in block # 1 is moved to page i', so that there results

$$S(i'', 1)=p/w+\beta-\alpha.$$

Note that column numbers of the X-matrix designate the destination page numbers in the final distribution. This means that the inter-page record movement corresponds to the increase or decrease in integers of $X(i,j)$ of the relevant two pages, but this data movement or algebraic adjustment between elements $X(i',j)$ and $X(i'',j)$ ($j=1, 2, \cdots, s/w$) must be in the same column j. Since there is the constraint of (2), the record transfer from page i'' to page i' in block # 1 is simultaneously accompanied by the transfer of the same number of records in the inverse direction in block # 1′, namely, from page i' to page i'' in block # 1′. We thus have $S(i', 1)=p/w$, and then page i' is pushed to slow memory. If eventually $S(i'', 1)=p/w$, then page i'' is also pushed to slow memory. If not, another page having a complementary S-value to that of page i'' is fetched and the two new pages are processed in the similar way, and at least one of them attains $S=p/w$. The entire preprocessing of block # 1 finishes by fetching p pages *at most*.

In this stepwise way, referring to, and then updating, the information of the X-matrix, inter-page exchange of records on main memory proceeds.

When the processing of block # $(w-1)$ is over, block # w (or, equivalently, block # $(w-1)'$) automatically satisfies (5) because of condition (2).

To have relation (5) in all of the blocks, and all of the p pages, $(w-1)p$ page fetches are thus required at most.

In those cases, such as full transposition [1], [2], where every record satisfies (5) from the outset, the whole of preprocessing can be deleted.

Block diagonalization. The preprocessing now having been completed, we then fetch the first, the second, $\cdots$, and the wth pages, and exchange records between these pages in main memory, in such a way that

$$\left.\begin{array}{l} S(1,1)\leftarrow S(1,1)+S(2,1)+\cdots+S(w,1),\\ S(2,2)\leftarrow S(2,1)+S(2,2)+\cdots+S(w,2),\\ \cdots \quad \cdots\\ S(w,w)\leftarrow S(w,1)+S(w,2)+\cdots+S(w,w). \end{array}\right\} \quad (6)$$

The new w pages thus shuffled are then pushed to slow memory. The corresponding transformation of the X-matrix is additions of elements in the same columns, not between distinct columns. This process results in

$$\left.\begin{array}{l} S(1,1)=S(2,2)=\cdots=S(w,w)=p;\\ S(i,j)=0 \quad (i\neq j;\ i,j=1,2,\cdots,w). \end{array}\right\} \quad (7)$$

We then fetch the next w pages, i.e., the $(w+1)$st, $(w+2)$nd, $\cdots$, $2w$th pages, and in the same way as in the above first w pages, we have

$$\left.\begin{array}{l} S(w+1,w+1)=\cdots=S(2w,2w)=p,\\ S(i,j)=0 \quad (i\neq j;\ i,j=w+1,\cdots,2w). \end{array}\right\} \quad (7')$$

Repeating these w-page fetches and the respective processing in main memory, we finally arrive at

$$S(i,j)=\begin{cases} p & ((i-1 \bmod w)+1=j;\\ & \ i=1,2,\cdots,s;\ j=1,2,\cdots,w),\\ 0 & (\text{otherwise}) \end{cases} \quad (8)$$

at which the X-matrix has zero partial sums except those at the shaded portions (see Fig. 2(1)). The X-matrix has now a periodic structure with periods of w rows and s/w columns. Namely, it is an assembly of $w\times s/w$ minor matrices each of which having a non-zero partial row. The above processing can be done by fetching (plus pushing) disjoint w-page sets, so that the block diagonalization can be done by p page-fetches.

Renaming pages (no page-fetch required), we have the X-matrix block-diagonalized as shown in (2) of Fig. 2; namely, the page numbers are redefined as

$$(j-1)\frac{s}{w}+i\leftarrow(i-1)w+j \quad (i=1,2,\cdots,s;j=1,2,\cdots,w). \quad (9)$$

We have thus completed the processing of the first pass. As a result, the X-matrix has w^2 $(s/w\times s/w)$-matrices along its diagonal, all the other off-diagonal elements being zero.

Now, moving on to the second pass, and assigning

$$s\leftarrow s/w,$$

we perform operations, similar to the first pass, on each of the $s/w\times s/w$ matrices obtained in the immediately preceding pass. The second pass that preprocesses and block-diagonalizes all of the w^2 minor matrices of the X-matrix requires at most $(w-1)p$ plus exactly p page fetches.

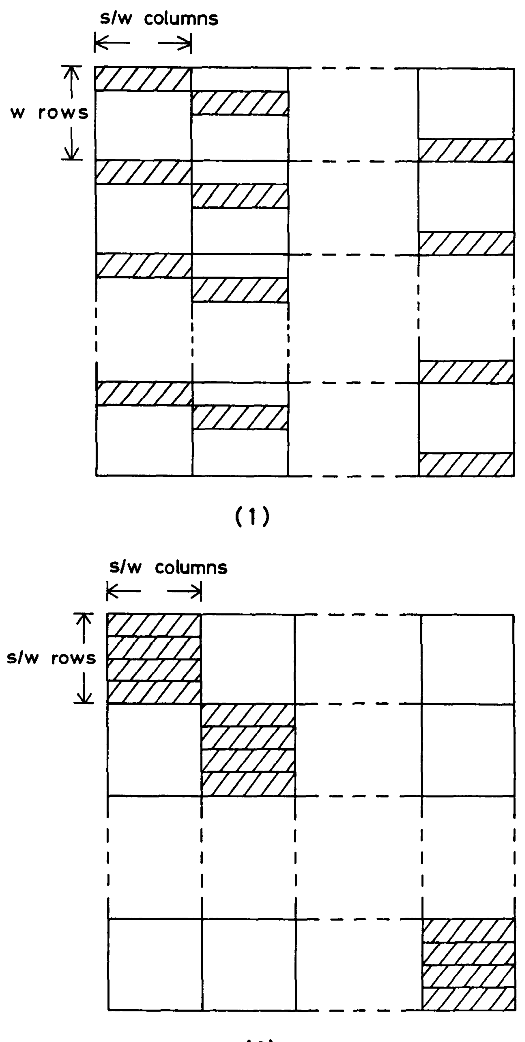

Fig. 2 Block diagonalization of the X-matrix ($s=p$ for the first pass): (1) before renaming pages, (2) after renaming pages.

The above procedures are carried out recursively on further passes until $s=w$, when the X-matrix is a complete diagonal matrix:

$$X(i,j)=p\delta_{ij} \quad (\delta_{ij}=1 \text{ for } i=j, \text{ otherwise zero}).$$

The number of passes is $\log_w p$. The final pass includes the process of reordering records in each page, if this intra-page permutation is necessary. We therefore have:

Theorem. Consider p pages each with p records in slow memory. The total number of page fetches, $F(A)$, required of an arbitrary permutation, is given by

$$p\log_w p\leqq F(A)\leqq wp\log_w p,$$

where the bounds are exact in the sense that the equalities actually hold. □

5	3						
	3	5					
		1	2	2	1	1	1
		1	1	1	2	2	1
		1	2	1	1	2	1
	2		1	1	2		2
1			1	2	1	1	2
2			1	1	1	2	1

(1)

1	3			3	1		
		4		4			
1		1	2	1	1	1	1
2		1	1		1	2	1
1		1	2		1	2	1
	3		1		2		2
1	2		1		1	1	2
2		1	1		1	2	1

(2)

1	3	4					
3		2	3				
1	3	1	3				
3	2	1	2				
				7	1		
				1	2	3	2
					3	2	3
					2	3	3

(3)

1	3	4					
4		1	3				
1	3	1	3				
2	2	2	2				
				3	1	4	
				2	2	2	2
				1	3	1	3
				2	2	1	3

(4)

5	3						
3	5						
		5	3				
		3	5				
				5	3		
				3	5		
						6	2
						2	6

(5)

8							
	8						
		8					
			8				
				8			
					8		
						8	
							8

(6)

Fig. 3 Stepwise changes of the X-matrix for the Case A where $w=2$ and $p=8$. (1) and (6) are X_{initial} and X_{final} respectively. (2) and (3) are the results of the first pass. The second pass yields (4) and (5), while the third and final pass gives (6), the solution. Blank squares indicate zero elements.

The case of 'pure' transposition can be handled with page fetches, given by the lower-bound value of the above theorem, hence the algorithm is in this case best possible. It has therefore been demonstrated that there is yet another best possible algorithm of transposition, that differs from that of [1] and [2]. In the present method, however, the stepwise transformation of the X-matrix must be retained and referred to in main memory at each step of subsequent record manipulations until $(X(i,j))=X_{\text{final}}$. An example of the stepwise transformation of the X-matrix is shown in Fig. 3 for the case that $w=2$, $p=2^3=8$. Since Case A is of primary importance among others, the algorithm for this case is shown in the Appendix in executable Pascal statements.

Case B-1—$p=k\cdot w^l$, $k=w_0^x$ ($2\leqq w_0<w$, $k\geqq 2$).

Divide the X-matrix into w_0 blocks each of which comprises p/w_0 contiguous columns. Taking w_0 for w, apply the Case A algorithm, which then yields w_0 minor square matrices of size $p/w_0\times p/w_0$. Further recursive use of the Case A algorithm to each minor matrix results in w_0^x ($=k$) ($w^l\times w^l$)-matrices, to each of which the Case A algorithm is again applied to complete the permutation. The number of page fetches in this case does not exceed

$$xw_0p+wp\log_w(p/k)=p(xw_0+wl).$$

Case B-2—$p=k\cdot w^l$, $k\neq w_0^x$.

The first step is to divide the X-matrix into w blocks each of which is made up of p/w contiguous columns, and then apply the Case A algorithm. In this case, however, let the algorithm terminate at $s=k$. As the second step, apply the algorithm of Case B-3 below to each of the resulting w^l minor matrices of size $k\times k$. The first step requires at most $wp\log_w w^l$ ($=wpl$) page fetches. In the second step, page fetches required are as follows. Let k' be such that it is the least positive integer satisfying $k''=k'w$ and $k''>k$. The number of page fetches required in the second step does not exceed $w^lk'w(k'\lceil\log_w k'\rceil+1)$. The resultant page fetches through this case do not exceed

$$wpl+k'w^{l+1}(k'\lceil\log_w k'\rceil+1).$$

Case B-3—$p\neq k\cdot w^l$.

Let k' be such that it is the least positive integer satisfying $q=k'w$ and $q>p$. To the X-matrix we add $(q-p)$ rows from below and $(q-p)$ columns from the right in such a way that

$$X(i,j)=1\quad(i,j=p+1,p+2,\cdots,q).$$

Appending these dummy elements to the X-matrix, we correspondingly provide $(q-p)$ extra pages all with blank records in slow memory in addition to the originally given p pages of records. To this enlarged X-matrix there is the constraint that

$$\sum_{i=1}^{q}X(i,j)=\sum_{j=1}^{q}X(i,j)=q,$$

instead of (2) previously shown. This means that, whenever a Case B-3 is anticipated to occur, every page should not be fully packed with records, but must have sufficient capacity to allow for the presently discussed extension. The algorithm is then as follows. First, divide the new X-matrix into k' blocks where each block consists of q/k' contiguous columns. Applying the Case A algorithm, we have k' minor matrices of size $w\times w$. As the second step, the Case A algorithm is used once for each of these minor matrices. The first step requires at most $k'q\lceil\log_w k'\rceil$ page fetches. The second step needs q page fetches. Case B-3 therefore requires not more than $q(k'\lceil\log_w k'\rceil+1)$ page fetches. See Fig. 4, where an example of Case B-3 is shown.

3. The Algorithm for Arbitrary Permutations: p' Pages, Each with p Records

This section considers the most general case of p' pages in slow memory, each page having p records ($p'\neq p$). By slight modification of the algorithms described in Section 2, this general case can be covered.

The X-matrix is now a matrix of size $p'\times p'$; it is subject to the condition given by Eq. (2).

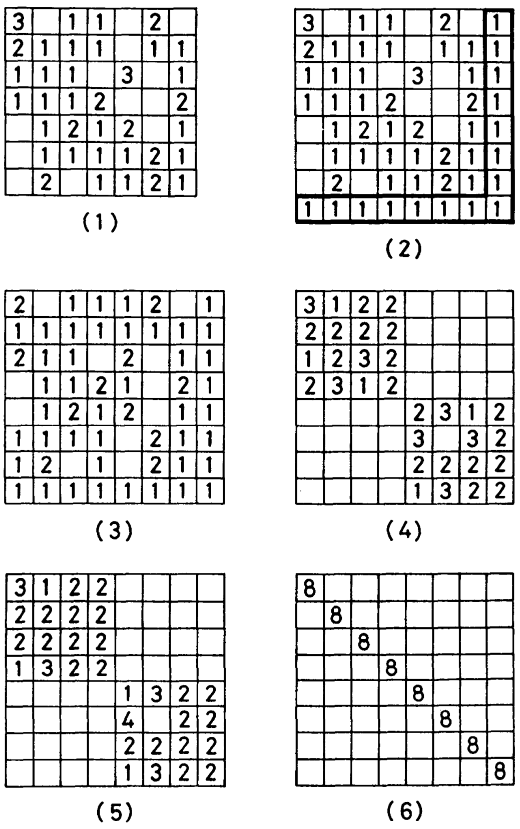

Fig. 4 An example of Case B-3 ($w=2$, $p=7$). (1) is X_{initial}. At (2) an extra page is appended in order to have $p=w^2$. The extended X-matrix thereafter changes following the Case A algorithm as illustrated in (3) through (6).

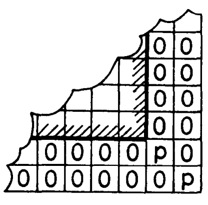

Fig. 5. Extra blank pages appended to the X-matrix (shaded, partly shown) with no increase in p.

We assume that $p'>p$ and p' is divisible by y, where

$$y \in \{w, w_0, k, k'\}.$$

If the given p' is not divisible by y, then we provide extra blank pages in slow memory so that resulting p' is divisible by y. This modification is not accompanied by any increase of p, the number of records per page, since the extension in the X-matrix is such as that shown in Fig. 5 ($p'-p=2$).

We discriminate between two cases:

Case C-1—p is divisible by y.

Preprocessing starts with partitioning the X-matrix into y blocks. At the completion of the preprocessing, each partial sum (as defined in Section 2) has become p/y. Thereafter the algorithms in the preceding section are applied. The number of page fetches required is that given by the relevant formulas of Section 2 where p is replaced by p'.

Typically p may be some power of w; the number of page fetches required, $F(C)$, is then given by

$$p' \log_w p' \leqq F(C) \leqq wp' \log_w p'.$$

Case C-2—p is not divisible by any y.

The X-matrix of size $p' \times p'$ is divided into y blocks, each having p'/y contiguous columns. Let the sum of the elements on each y rows of each block be p by exchanging records between two pages in main memory. It suffices for this part of the whole algorithm that, within each set of y rows (i.e., partial sums) of each block, integer $\lceil p/y \rceil$ is assigned to $(p/y - \lfloor p/y \rfloor)y$ partial sums, while integer $\lfloor p/y \rfloor$ is assigned to the rest of them. See (1) and (2) of Fig. 6. What should be done next is straightforward; see Section 2. See Fig. 6 for the stepwise changes of the X-matrix. Even in this case, the number of page fetches required is that given by the relevant formulas of Section 2 where p is replaced by p'.

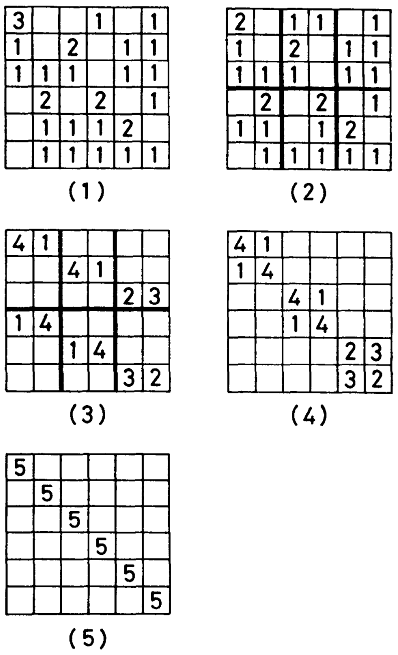

Fig. 6 An example of Case C-2 ($p=5$, $p'=6$, $y=w=3$).

4. Applications

There are two important applications of the results obtained in this paper. These will be discussed in this section.

4.1 Application to Key Sorting

In some cases records are long whereas their keys are short. Key sorting is then applicable; the keys are first sorted, most probably in main memory, and thus we specify where each record is to be moved. This is, in our terminology, none other than the specification of the X-matrix. The subsequent external rearrangement of long records fits well into the algorithmic scheme we have been considering in this paper.

Considering p pages each with p records, it is known [3] that in a demand-paging environment the above rearrangement of records requires, on the average, as many page fetches as

$$p^2\frac{p-w}{p} \simeq O(p^2).$$

On the other hand, the Case A algorithm of Section 2 can dispose the problem with page fetches $F(A)$ bounded as

$$p \log_w p \leqq F(A) \leqq wp \log_w p.$$

Although we have here assumed that $p = w^l$, w being the main memory size in page, this indicates that our algorithm is efficient for a large mass of data.

4.2 Application to Multidimensional FFT

At this moment, or even for several years to come, it seems difficult to solve veritable 3-dimensional scientific problems, such as those of fluid dynamics, despite the progressive increase of speed of operations in computers. The main bottleneck is caused by the limited size of main memory, which cannot meet the requirement of 3-dimensional problems. For example, for a mesh of $300 \times 300 \times 300$ there should be 27M word memory, but actually an order of magnitude larger memory is required because there are several distinct physical quantities to be held in memory for each of the mesh points. If discretization errors are to be suppressed sufficiently low to have, say, an accuracy of more than 3 decimals, then a mesh as fine as $1000 \times 1000 \times 1000$ will be needed. Bulky memory causes slow access. It seems therefore inevitable that the overall efficiency of computation hinges on the presence of memory levels, fast and slow.

Consider, for example, two-dimensional FFT (Fast Fourier Transform) in a paging environment. Data become exponentially more numerous with the increase of the number of dimensions, hence data transfers between fast and slow memories are mandatory. Take w and p for the main memory size in pages and the number of data per page. One-dimensional FFT for data of p pages requires $p \log_w p$ page fetches $(p > w)$. Suppose

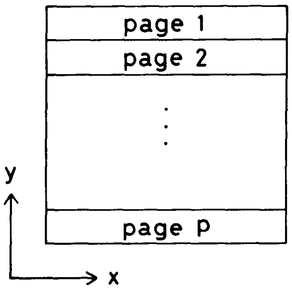

Fig. 7 Two-dimensional data aligned in x-direction over p pages of slow memory.

that there are two-dimensional data, as shown in Fig. 7, where they are aligned along the x-axis over p pages. If we carry out one-dimensional FFT twice, first sweeping along the x-axis over the p-page data, and then along the y-axis, the page fetches that will be incurred are:

p fetches along the x-axis,

p^2 fetches along the y-axis

or $p + p^2$ fetches in total. If, on the contrary, we sweep along the x-axis and then perform a 'transposition' of data to align them along the y-axis, the page fetches required for the two-dimensional FFT are:

p fetches along the x-axis,

$p \log_w p$ fetches for transposition,

p fetches along the y-axis

or $p(2 + \log_w p)$ fetches in total. The second scheme is by far the better, even though the process of transposition adds to CPU cost. There is a more sophisticated algorithm that handles the two-dimensional FFT *per se* with a smaller number of operations, such as multiplications [4]. This algorithm, however, yields as much paging cost as the above repetition of one-dimensional FFT; hence the improvement on the CPU cost is swamped by the cost of data transfers between the fast and thə slow memories.

References

1. Floyd, R. W., Permuting information in idealized two-level storage in *Complexity of Computer Computations* (R. Miller and J. Thatcher, editors), pp. 105–109, Plenum Press, New York (1972).
2. Tsuda, T. and Sato, T., Transposition of Large Tabular Data Structures with Applications to Physical Database Organization. Part 1. Transposition of Tabular Data Structures, *Acta Inf.*, Vol. 19, pp. 13–33 (1983).
3. Brawn, B. S., Gustavson, F. G. and Mankin, F. S., Sorting in Paging Environment, *Comm. ACM*, Vol. 13, No. 8, pp. 483–494 (1970).
4. Nussbaumer, H. J., Fast Fourier Transform and Convolution Algorithms, Chap. 7, Springer-Verlag, Berlin (1981).

APPENDIX

```
PROGRAM PERMUTATION(INPUT,OUTPUT);
  CONST Q=100;           (* MAXIMUM INDEX OF X,X2 MATRIX *)
   TYPE MATRIX=ARRAY (.1..Q,1..Q.) OF INTEGER;
        VECTOR=ARRAY (.1..Q.) OF INTEGER;
    VAR P: INTEGER;      (* NO. OF PAGES, RECORDS/PAGE *)
        W: INTEGER;      (* MAIN MEMORY SIZE *)
        PW: INTEGER;     (* NO. OF RECORDS / PARTIAL ROW *)
        FETCH: INTEGER;  (* NO. OF PAGE-FETCHES *)
        I,J,K: INTEGER;
        X :MATRIX;       (* PAGE-TRANSFER MANAGEMENT TABLE *)
        X2:MATRIX;       (* WORK SPACE TO MANIPUTLATE X-MATRIX *)
        TID: VECTOR;     (* TID(.I.) MEANS THE INITIAL POSITION *)
                         (* OF I-TH ELEMENT OF SORTED ARRAY     *)
(*****************************************************************)
(* THIS PROGRAM MANIPULATES X-MATRIX (I.E.,MANAGEMENT TABLE) IN  *)
(* ORDER TO CARRY OUT PERMUTING P*P TABULAR DATA(I.E.,P PAGES,   *)
(* EACH WITH P RECORDS), WHERE P IS SOME POWER OF W(MEMORY SIZE).*)
(*****************************************************************)

  PROCEDURE QSORT(VAR A:VECTOR; S,T:INTEGER);
  (* THIS PROCEDURE CARRIES OUT SORTING OF ARRAY A(.I.),       *)
  (* WHERE S<=I<=T.                                             *)
    PROCEDURE SORT(S,T: INTEGER);
       VAR I,J,Y1,Y2: INTEGER;
     BEGIN
       I:=S;  J:=T;  Y1:=A(.(S+T) DIV 2.);
       REPEAT
         WHILE A(.I.)<Y1 DO I:=I+1;
         WHILE Y1<A(.J.) DO J:=J-1;
         IF I<=J THEN
         BEGIN
           Y2:=A(.I.);  A(.I.):=A(.J.);  A(.J.):=Y2;
           Y2:=TID(.I.);  TID(.I.):=TID(.J.);  TID(.J.):=Y2;
           I:=I+1;  J:=J-1
         END
       UNTIL I>J;
       IF S<J THEN SORT(S,J);
       IF I<T THEN SORT(I,T);
     END;  (* END OF SORT *)

    BEGIN FOR I:=1 TO T DO TID(.I.):=I;
          SORT(S,T);
    END; (* END OF QSORT *)

  PROCEDURE DOUBLE(S:INTEGER; B:INTEGER);
                    (* S: SIZE OF SUB MATRIX;  B: BASE POSITION *)
    VAR R: INTEGER;   (* SIZE OF SUB MATRIX AT NEXT STAGE *)
        DIF: INTEGER; (* DIFFERENCE OF NO. OF RECORDS BETWEEN   *)
                      (* TWO PAGES IN MAIN MEMORY               *)
        BASE,LBASE,CBASE: INTEGER; (* BASE POSITION *)
        I,J,K,M,Y1,Y2: INTEGER;
        BOX:VECTOR;   (* PARTIAL SUM *)
 (* THIS PROCEDURE MANIPULATES X(.I,J.), WHERE B+1<=(I,J)<=B+S  *)

  PROCEDURE DIST(DIF:INTEGER);
    VAR DIF2: INTEGER;  (* TEMPORARY VARIABLE OF DIF  *)
        IT,LT: INTEGER; (* PAGE NO. IN MAIN MEMORY    *)
        BJ,J: INTEGER;
  (* THIS PROCEDURE MOVE RECORDS IN ORDER TO ADJUST PARTIAL SUM *)
```

```
BEGIN DIF2:=DIF;
     LT:=TID(.M.)+B; IT:=TID(.I.)+B;
     FOR J:=1 TO R DO
     IF DIF2>0 THEN
     BEGIN BJ:=BASE+J;
           IF X(.LT,BJ.)>=DIF2
           THEN BEGIN X(.LT,BJ.):=X(.LT,BJ.)-DIF2;
                      X(.IT,BJ.):=X(.IT,BJ.)+DIF2;
                      DIF2:=0
                END
           ELSE BEGIN X(.IT,BJ.):=X(.IT,BJ.)+X(.LT,BJ.);
                      DIF2:=DIF2-X(.LT,BJ.);
                      X(.LT,BJ.):=0
                END
     END;
     DIF2:=DIF;                  (* COMPLEMENTARY MOVING *)
     FOR J:=R+1 TO P-BASE DO
     IF DIF2>0 THEN
     BEGIN BJ:=BASE+J;
           IF X(.IT,BJ.)>=DIF2
           THEN BEGIN X(.IT,BJ.):=X(.IT,BJ.)-DIF2;
                      X(.LT,BJ.):=X(.LT,BJ.)+DIF2;
                      DIF2:=0
                END
           ELSE BEGIN X(.LT,BJ.):=X(.LT,BJ.)+X(.IT,BJ.);
                      DIF2:=DIF2-X(.IT,BJ.);
                      X(.IT,BJ.):=0
                END
     END
  END; (* END OF DIST *)

BEGIN R:=S DIV W;
  IF R<>1 THEN
  BEGIN                          (* PRE-PROCESSING *)
    FOR K:=0 TO W-2 DO
      BEGIN BASE:=K*R+B;
        FOR I:=1 TO S DO
          BEGIN BOX(.I.):=0;
            FOR J:=1 TO R DO
              BOX(.I.):=BOX(.I.)+X(.I+B,BASE+J.);
          END;
        M:=S; I:=1;
        QSORT(BOX,1,S);
        WHILE BOX(.M.)>PW DO
          BEGIN DIF:=BOX(.M.)-PW;
            IF PW-BOX(.I.)>=DIF
            THEN BEGIN DIST(DIF);
                       BOX(.M.):=PW; BOX(.I.):=BOX(.I.)+DIF;
                 END
            ELSE BEGIN DIST(PW-BOX(.I.));
                       BOX(.M.):=BOX(.M.)-PW+BOX(.I.);
                       BOX(.I.):=PW:
                       I:=I+1; FETCH:=FETCH+1
                 END;
            M:=M-1;
            FETCH:=FETCH+1
          END
      END
  END;
  FOR K:=0 TO R-1 DO             (* PASS *)
```

```
    BEGIN LBASE:=K*W+B;
      FOR M:=1 TO W-1 DO
        FOR J:=1+B TO S+B DO
          BEGIN X(.LBASE+W,J.):=X(.LBASE+W,J.)+X(.LBASE+M,J.);
                X(.LBASE+M,J.):=0
           END;
      FOR M:=1 TO W-1 DO
        BEGIN CBASE:=(M-1)*R+B;
          FOR J:=1 TO R DO
            BEGIN X(.LBASE+M,CBASE+J.):=X(.LBASE+W,CBASE+J.);
                  X(.LBASE+W,CBASE+J.):=0
            END
        END;
        FETCH:=FETCH+W
    END;
    IF R<>1 THEN                (* RENAME PAGE_NO.*)
    BEGIN FOR I:=1 TO S DO
      BEGIN Y1:=(I-1) DIV R;
            Y2:=(I-Y1*R-1)*W+Y1+1;
            FOR J:=1 TO P DO
              X2(.I+B,J.):=X(.Y2+B,J.)
      END;
      FOR I:=1+B TO S+B DO
        FOR J:=1 TO P DO
          X(.I,J.):=X2(.I,J.);
      FOR K:=0 TO W-1 DO        (* GO TO NEXT STAGE *)
      BEGIN BASE:=K*R+B;
            WRITELN;
            LBASE:=0; CBASE:=0;
            DOUBLE(R,BASE)
      END
    END
    ELSE FOR I:=1+B TO W+B DO (* FINAL PASS *)
           BEGIN WRITELN;
                 FOR J:=1+B TO W+B DO
                   WRITE(X(.I,J.));
           END;
 END;  (* END OF DOUBLE *)

BEGIN
      READ(W,P);
      FOR I:=1 TO Q DO
        FOR J:=1 TO Q DO
          BEGIN X(.I,J.):=0;  X2(.I,J.):=0  END;
      FOR I:=1 TO P DO
        FOR J:=1 TO P DO
          READ(X(.I,J.));
      WRITE('X-MATRIX(INITIAL):');
      FOR I:=1 TO P DO
      BEGIN WRITELN;
        FOR J:=1 TO P DO
          WRITE(X(.I,J.))  END;
      WRITELN;
      PW:=P DIV W;
      FETCH:=0;
      WRITE('X-MATRIX(FINAL):');
      DOUBLE(P,0);
      WRITELN('      FETCH=',FETCH)
 END.
```

(Received April 1, 1982)