Efficient Parallel Out-of-core Matrix Transposition

Sriram Krishnamoorthy Gerald Baumgartner
Daniel Cociorva Chi-Chung Lam
P. Sadayappan
Department of Computer and Information Science
395, Dreese Laboratories, 2015 Neil Avenue
The Ohio State University
Columbus, OH, USA - 43210
{krishnsr,gb,cociorva,clam,saday}@cis.ohio-state.edu

***Abstract*— This paper addresses the problem of parallel transposition of large out-of-core arrays. Although algorithms for out-of-core matrix transposition have been widely studied, previously proposed algorithms have sought to minimize the number of I/O operations and the in-memory permutation time. We propose an algorithm that directly targets the improvement of overall transposition time. The I/O characteristics of the system are used to determine the read, write and communication block sizes such that the total execution time is minimized. We also provide a solution to the array redistribution problem for arrays on disk. The solution to the sequential transposition problem and the parallel array redistribution problem are then combined to obtain an algorithm for the parallel out-of-core transposition problem.**

***Index Terms*— out-of-core, parallel matrix transposition, disk-based array redistribution**

I. Introduction

This paper addresses the problem of parallel out-of-core matrix transposition. The problem is viewed in terms of two sub-problems: disk-based array redistribution, followed by concurrent independent uniprocessor transposition of disk-based arrays. The same algebraic framework is used for both steps. We first address the sequential transposition problem, which has been previously studied.

Consider an $N \times N$ matrix that is stored in disk in row-major order. The system has main memory, which can hold M elements, where $M < N^2$, $M = O(N)$. The problem is to transpose the matrix stored in disk, when only a part of the matrix can be brought into memory at any time. Applications that need to access the elements of a matrix in column-major order transpose the matrix before accessing its elements. Matrix transpose is a key operation in various scientific applications. For example, the multidimensional Fast Fourier transform (FFT) [1], [2] can be implemented as a series of one-dimensional FFTs, one along each dimension. For a matrix stored in disk in row-major order that is too large to fit in memory, the most effective mechanism is to transpose the matrix between the one-dimensional FFTs.

Our primary motivation for addressing the parallel out-of-core matrix transposition problem arises from the domain of electronic structure calculations using ab initio quantum chemistry models such as Coupled Cluster models. We are developing an automatic synthesis system called the Tensor Contraction Engine (TCE) [3], to generate efficient parallel programs from high level expressions for a class of computations expressible as tensor contractions [4]–[7]. Often the tensors (essentially multi-dimensional matrices) are too large to fit in memory and must be disk-resident.

The optimized parallel programs synthesized by the tool often have to take as input large disk-resident tensors created by other software packages, such as the NWChem computational chemistry suite [8]. For efficient execution, the TCE-synthesized program might need to store and access the disk-resident tensors in a very different order than that used by the producer program. Efficient transformation of the data from the available format to the required format is required through transposition and/or re-blocking. In addition, when TCE-synthesized code is used on different machines, different transformations are required on the data produced by packages like NWChem, requiring efficient out-of-core matrix transposition and transformation algorithms.

This problem has been widely studied in the literature. A simple in-place element-wise approach to transpose the matrix is prohibitively expensive. The block transposition algorithm transposes the array in a single pass

Supported in part by the National Science Foundation through the Information Technology Research program (CHE-0121676)

in $O(N^{3/2})$ I/O operations. An in-place transposition algorithm requiring $O(N \log N)$ disk accesses was proposed by Eklundh [9]. This algorithm requires at least two rows to fit in memory. Extensions to the algorithm for rectangular matrices were presented in [10]–[12]. Kaushik et al. [13] proposed an out-of-place algorithm that improves upon these algorithms by reducing the number of read operations. Suh and Prasanna [14] reduced the in-memory permutation time by using *collect* buffers, instead of in-memory permutation, in addition to reducing the number of I/O operations. Their algorithm combines writes and collects the rows to be permuted in subsequent passes.

All these studies use the number of I/O operations as the primary optimization metric. Although the execution time of the solution provided has been improved by all these efforts, the total execution time has not been used as the primary metric for optimization. A reduction in the number of I/O operations, in most cases, translates to larger sizes of I/O blocks. The importance given to reducing the number of I/O operations is due to the fact that the seek time for the disk head is very large (of the order of several milliseconds) compared to the per-byte transfer time (of the order of microseconds or less). If the I/O blocks read/written are relatively small, the total number of I/O operations is indeed a suitable optimization metric. However, as the I/O blocks get larger, the data transfer time becomes significant and begins to dominate the total access time. Since previously proposed algorithms for out-of-core transposition have focused on reducing the number of I/O operations, they can become sub-optimal when large block transfers are involved.

Cormen et al. [15] solve the problem based on the parallel disk model (PDM) [16]. PDM handles the read and write block sizes as equivalent, while the I/O characteristics of reads and writes can differ widely. PDM uses the number of I/O operations as the metric, where the size of each I/O is determined by the layout of data on disk. It does not take into account the effect of read-ahead and request reordering in the I/O subsystem.

All the algorithms in the literature determine the fundamental unit of I/O based on the size of the matrix, i.e., they are data-centric. The basic unit of I/O operation in these algorithms is one row of the matrix or a multiple thereof. They do not adapt to the I/O characteristics of the system. In contrast, the approach proposed here takes into account the empirically determined I/O characteristics of the disk and file system in determining the parameters of the algorithm. The basic unit of I/O is not a row, but is determined by the I/O characteristics and the instance of the problem at hand. The execution time of the algorithm on the system is estimated based on the experimentally observed I/O characteristics. The parameters that minimize the execution time are chosen.

The paper is organized as follows. The I/O characteristics of two systems are discussed in Section II. In Section III the transposition problem is formulated using the matrix-vector product notation. The sequential transposition algorithm is described in Section IV. The algorithm is extended to parallel systems in Section V. Experimental results are presented in Section VI. Section VII concludes the paper.

II. I/O Characteristics

Out-of-core transposition involves reading and writing blocks of data at different strides. To understand the variation in performance of the algorithm with respect to these parameters, we studied the variation of read and write times with changes in size and stride of I/O on the machines in two clusters at the Ohio Supercomputer Center (OSC) [17]. The configuration of the machines in both the clusters is shown in Table I. Fig. 1 and Fig. 2 show the strided read and write times respectively on the IA32 cluster. Fig. 3 and Fig. 4 show the strided read and write times respectively on the Itanium 2 cluster.

For both the systems we observe that beyond a particular block size the stride does not affect the per-byte transfer cost and approximates to the cost of sequential I/O. More importantly, the incremental improvement obtained in the I/O time by increasing the block size decreases and is very small beyond a particular block size. We expect this observation to hold across a wide variety of systems. These block sizes, above which the per-byte read and write times are not affected by the stride of access, will henceforth be referred to as the *read* and *write thresholds* respectively. These parameters vary depending on the system under consideration and the per-byte read and write costs can saturate at different block sizes. The read and write thresholds for IA 32 Cluster are 2MB and 1MB, respectively. For the Itanium 2 Cluster they are 1MB and 1MB, respectively.

An out-of-core algorithm needs to perform I/O on sufficiently large block sizes for good performance. On the other hand, a smaller block size provides more flexibility in accessing data and can improve performance of the algorithm. In the case of out-of-core matrix transposition if the thresholds are smaller than N, the size of the matrix, fractions of a row can be read and written with little additional penalty, irrespective of the stride of access. This might result in a decrease in the number of passes. In the extreme case, if each element is large enough to be read/written individually, a simple single-pass element-wise transposition would be efficient. An out-of-core algorithm that chooses the largest possible I/O block size when I/O on a much smaller block can be

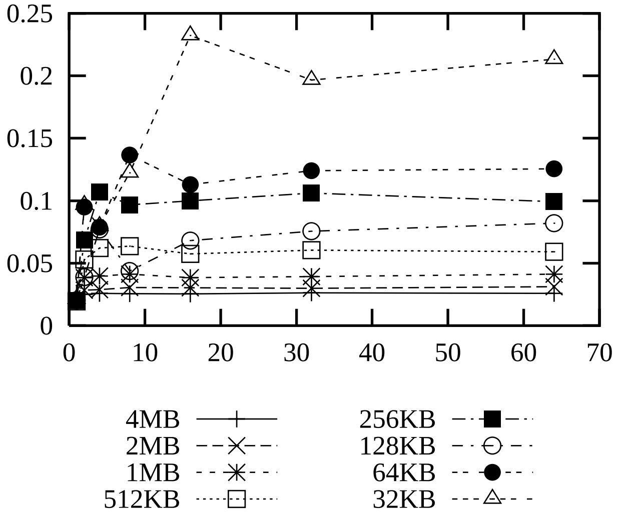

Fig. 1. Strided read times for the IA32 Cluster. x-axis is the stride in number of blocks. y-axis is the per-byte access time in microseconds. Each line corresponds to a block size.

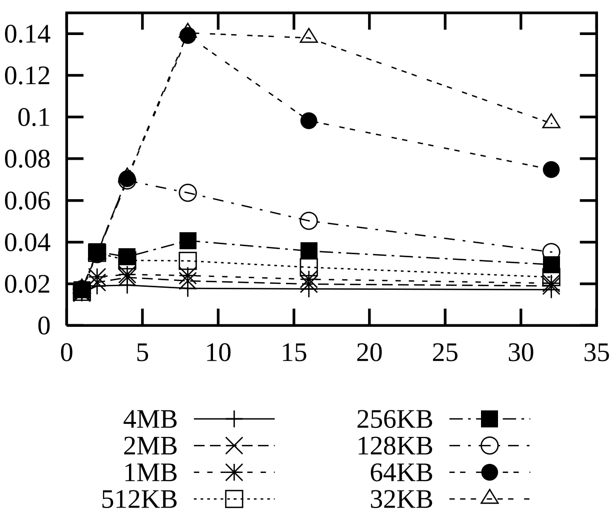

Fig. 3. Strided read times for the Itanium 2 Cluster. x-axis is the stride in number of blocks. y-axis is the per-byte access time in microseconds. Each line corresponds to a block size

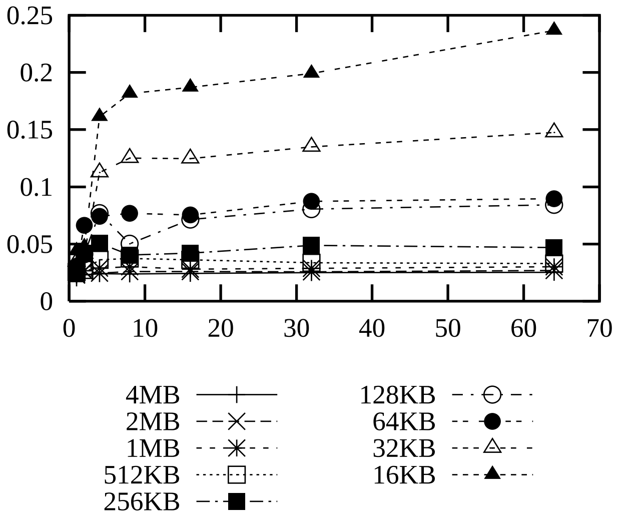

Fig. 2. Strided write times for the IA32 Cluster. x-axis is the stride in number of blocks. y-axis is the per-byte access time in microseconds. Each line corresponds to a block size.

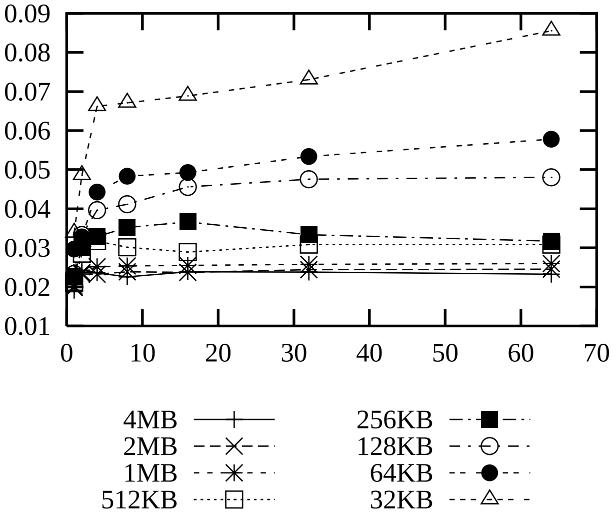

Fig. 4. Strided write times for the Itanium 2 Cluster. x-axis is the stride in number of blocks. y-axis is the per-byte access time in microseconds. Each line corresponds to a block size

performed efficiently may not be optimal. An algorithm might involve more I/O operations but be faster than another algorithm with fewer I/O operations due to this effect.

III. Matrix Vector Product Formulation of Transposition Algorithms

In this section, matrix transposition algorithms are formulated based on the matrix-vector notation used in [18]. This section provides a generic formulation for transposition algorithms.

Transposition of a matrix can be viewed as an interchange of the indices of the matrix.

$$T(i, j) = (j, i)$$

where i is the row index and j is the column index. This is a particular instance of a general class of index transformation algorithms.

Each element of the array on disk has a linear address obtained by concatenating the column index bits to the row index bits. This is the address upon which the permutation is applied. The transformation of the address vector using a permutation matrix corresponds to the permutation of the address vector and hence the matrix. The linear address of an element in the array contains

System	Configuration			
	Processor	Memory	OS	Compiler
IA32 Cluster	Dual AMD Athlon MP (1.533 GHz)	2GB	linux 2.4.20	pgcc 4.0-2
Itanium 2 Cluster	Dual Itanium-2(900 MHz)	4GB	linux 2.4.18	gcc 2.96

TABLE I
CONFIGURATION OF THE SYSTEMS USED FOR I/O CHARACTERIZATION.

$2n(N = 2^n)$ bits and hence the permutation matrix contains $2n$ rows.

The identity of the transformation is $\begin{pmatrix} I_n & 0 \\ 0 & I_n \end{pmatrix}$. Matrix transposition is defined as the permutation of the address vector i

$$i \to Ti$$

where T is the transformation matrix $\begin{pmatrix} 0 & I_n \\ I_n & 0 \end{pmatrix}$.

We use the following notation in the discussion.

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \tag{1}$$

$$L(A, B) = \begin{pmatrix} 0 & B \\ A & 0 \end{pmatrix} \tag{2}$$

Thus, $L(I_n, I_n)$ is the desired permutation. Since the entire array does not fit in memory, $L(I_n, I_n)$ is factorized into a number of permutation matrices such that the transformation effected by each of the matrices can be done with the memory available.

Any out-of-core matrix transposition algorithm consists of three phases — read, permute and write. Each of these phases can be used to permute the linear address. Hence each phase corresponds to a permutation on the linear address and can be represented by a permutation matrix. These phases are repeated on disjoint parts of the array in the different steps of each pass. The algorithm might involve many passes, each operating on the entire array. Thus, out-of-core matrix transformation algorithms are of the form

$$T = L(I_n, I_n) = \prod_{i=t-1}^{i=0} W_i P_i R_i$$

where W_i is the permutation matrix corresponding to a write, R_i is a permutation matrix corresponding to a read and P_i corresponds to in-memory permutation. The algorithms under this formulation read some data, permute it in memory, and write the data to disk before reading data for the next step in the same pass. t specifies the number of passes. Thus each algorithm is defined by the parameters t, W_i, P_i and R_i, where the suffix i is used to refer to the permutations in the ith pass. Each algorithm can also have additional parameters.

Some restrictions apply to the possible values of W_i, P_i and R_i. These restrictions are induced by the memory constraint involved in the algorithm. Each permutation matrix must correspond to a transformation of the given matrix that can be done with the memory available. Thus, each step of the algorithm can operate on at most M elements. In particular, W_i, P_i and R_i must be expressed as

$$R_i = A_{2*n-r} \oplus I_r \quad r \leq m$$
$$P_i = I_{2*n-m} \oplus B_m$$
$$W_i = C_{2*n-w} \oplus I_w \quad w \leq m$$

The algorithm reads $R = 2^r$ elements and writes $W = 2^w$ elements in one I/O operation. The formulation enforces contiguity in these operations and requires the amount of data read to be less than the memory size $M = 2^m$. Henceforth, we use the term read(write) block size to refer to $R(W)$ and the least significant $r(w)$ rows of the permutation matrix, interchangeably. The reference will be clear from the context. Also note that P_i can permute only data corresponding to elements in memory. Given these parameters for an algorithm it can be implemented as

```
Algorithm 1: Generic Transposition Algorithm
for $i = 0$ to $t - 1$
    for $j = 0$ to $2^{2*n-m} - 1$
        Read M elements at address $R_i^{-1}(j)$
        /*Might involve multiple I/O operations*/
        Permute data in memory according to $P_i$.
        Write M elements at address $W_i(j)$
        /*Might involve multiple I/O operations*/
```

The read (write) may involve multiple I/O operations each of size at least $2^r(2^w)$ elements.

For a discussion of performance of different transposition algorithms based on these I/O characteristics, refer to [19].

IV. SEQUENTIAL TRANSPOSITION ALGORITHM

Our algorithm is based on estimating the total transposition time and choosing parameters for the algorithm that optimize it. The observation that an increase in I/O size beyond the threshold does not influence the performance of the algorithm is exploited. There is a trade-off between the I/O size and the number of passes the algorithm requires. The smaller the I/O size, the more the algorithms approach the block-transposition algorithm and hence run in a smaller number of passes. However, reducing the I/O size below the threshold increases the I/O time above the minimum possible.

The transposition time can be written as

$$\mathit{Time}_{\mathit{total}} = \sum_{i=0}^{i=t-1} \mathit{Time}_{\mathit{Read}} + \mathit{Time}_{\mathit{Permute}} + \mathit{Time}_{\mathit{Write}}$$

The read and write times for each pass can be computed from the stride and block size of the I/O operation. Estimating the permutation time is more difficult as it depends on the exact permutation involved. Unlike the I/O characteristics of a system, which can be determined independent of any specific algorithm, the permutation characteristic for each algorithm has to be individually determined. Here, we determine the best parameters for the algorithm that optimize the total I/O time. The characteristics of the algorithm allow for optimizing the in-memory permutation, as will be discussed later.

The algorithm has two parameters, namely the read and write block sizes. They are chosen close to the threshold in order to optimize the total I/O time. The most common case in which the I/O block size is chosen to be smaller than the threshold is when such a choice reduces the number of passes and offsets the additional cost incurred due to the smaller I/O size.

In previous algorithms, the basic unit of I/O is a row. The I/O permutations are of the form $A \oplus I_n$, while the required permutation $L(I_n, I_n)$ involves exchanging the upper and lower n address elements in the address vector. The nature of the I/O permutation prevents any effective permutation from being done in the read and write phases. The I/O phases 'gather' data to be permuted and 'scatter' the result of the permutation. In our algorithm, the I/O block size could be smaller than N, say $B = 2^b$, in which case the exchange $(0 : b-1) \leftrightarrow (n : n+b-1)$ can be done in the read and/or write phases. This reduces the number of address vector elements to be permuted in the permutation phase and might result in a reduction in the number of passes, and hence significantly reduce transposition time.

Our algorithm is formulated as shown below. The unit of each read and write is 2^r and 2^w elements respectively. Except in the first pass, the algorithm reads M elements in each read operation.

Conditions to be satisfied

$$n \geq w$$
$$m \geq r \geq w$$
$$m > w$$

Parameters

$$s_0 = \begin{cases} \min(m-r, w) & \text{if } r < n \\ \min(m-n, w) & \text{if } r \geq n \end{cases}$$
$$t = \begin{cases} 1 & \text{if } s_0 = w \\ 1 + \lceil \frac{w-s_0}{m-w} \rceil & \text{otherwise} \end{cases}$$
$$k = (w - s_0) \bmod (m - w)$$
$$s_i = \begin{cases} k & \text{if } i = t-1 \text{ and } k > 0 \\ m - w & \text{otherwise} \end{cases}$$

First pass $(i = 0)$

Case 1: $r \geq n$

$$R_0 = I_{2n}$$
$$P_0 = I_{n-s_0} \oplus L(I_{s_0}, I_{n-w} \oplus L(I_{w-s_0}, I_{s_0}))$$
$$W_0 = L(I_{n-s_0}, I_{n-w+s_0}) \oplus I_w$$

Case 2: $r < n$

$$R_0 = I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r$$
$$P_0 = I_{2n-(r+s_0)} \oplus L(I_{s_0}, I_{r-w} \oplus L(I_{w-s_0}, I_{s_0}))$$
$$W_0 = L(I_{n-s_0}, I_{n-w+s_0}) \oplus I_w$$

Remaining passes $(0 < i \leq t-1)$

$$sp_i = \sum_{j=0}^{i-1} s_j$$
$$R_i = I_{2n}$$
$$P_i = I_{2n-(w+s_i)} \oplus L(I_{s_i}, L(I_{w-sp_i-s_i}, I_{s_i})) \oplus I_{sp_i}$$
$$W_i = I_{n-w} \oplus L(I_{s_{i-1}} \oplus I_{n-s_{i-1}-s_i}, I_{s_i}) \oplus I_w$$

The formulation requires the read block size to be at least as large as the write block size, which could be relaxed. It is also assumed, as in earlier algorithms, that each row is individually writable. The memory size and read block size do not have any relation to the size of the array. Thus restrictions in earlier algorithms on minimum memory size in terms of number of rows of the array do not hold. The number of passes, t, depends primarily on the write block size and the memory size. The algorithm involves moving the least significant w elements in the linear address beyond the write block size(w). The smaller the write block size, the smaller the number of passes. The larger the memory, the more room there is for permutation and the smaller the number of passes. $s_0, \ldots, s_{t-1}$ specify the number of rows to be permuted out of the write block size in each of the t passes. Additionally, the first pass permutes the address elements in the column index that are beyond the write block size to their target positions. The first pass also optimizes for the case when the read block size is smaller than a row, by reading at a stride.

With increasing memory size, a modification of the I/O parameters provides diminishing improvements, unless it results in a reduction in the number of passes. Greater improvements can be obtained if the additional memory available is used to improve permutation time.

Kaushik et al. perform an in-place in-memory permutation. Suh and Prasanna use collect buffers to collect data to be written in each write operation. The locality of the permutation operation can be improved by optimizations such as blocking.

We use collect operations to perform the permutation, as this was empirically found to take less time than in-memory permutation. The permutation involved in the first pass is similar to transposition. Since the naive element-wise approach or even the collect operation has poor cache performance in the first pass, the permutation was done out-of-place in-memory. The I/O size was further reduced in order to maintain the number of passes.

V. Parallel Out-of-Core Matrix Transposition

In this section, the problem of transposing an out-of-core array distributed among multiple processors is discussed. Each processor has a local disk and the array is distributed among the processors in a row-blocked fashion. The required distribution of the transposed array among the processors is specified.

In the following discussion, we first formulate the representation of an array distributed among multiple processors. Then an algorithm is provided for redistributing out-of-core arrays in a parallel system. To our knowledge the problem of parallel out-of-core array redistribution has not been addressed previously.

The array redistribution mechanism and the sequential transposition algorithm are combined to describe the out-of-core transposition algorithm for arrays distributed among multiple processors.

A. Formulation for Arrays Distributed among Multiple Processors

The arrays are assumed to be distributed in a regular fashion so that some of the elements in the address vector represent the processor identifier. This corresponds to a mapping of the elements of the array to a sequence of processors. A row-blocked distribution is obtained when the most significant elements in the address vector represent the processor identifier. A cyclic distribution is obtained when the least significant elements of the address vector represent the processor identifier.

We define the linear address vector of an element in the array to be the concatenation of the address vector of the element in the local disk to the processor identifier. This view preserves the notion of contiguity of elements which differ in the lower most elements of the address vector, analogous to the sequential formulation. Hence the formulation can represent read and write thresholds in the address vector and the access pattern that can take advantage of prefetching as well.

Given that the uppermost elements in the linear address vector correspond to the processor identifier, the distribution of the array among multiple processors corresponds to choosing a set of elements in the address vector to become the uppermost elements. Hence array distribution among multiple processors can be viewed as a permutation of the linear address space of the array. The identity of array distribution is I_{2n}, which corresponds to a row-blocked distribution. Any other distribution of data among processors is viewed as a permutation on the row-blocked distribution. For example, a cyclic distribution of an array among two processors corresponds to the following permutation:

$$\begin{pmatrix} 0 & 1 \\ I_{2*n-1} & 0 \end{pmatrix}$$

B. Array Redistribution Problem

The array redistribution problem is stated as follows: Given an array distributed among processors, represented by a permutation matrix, achieve a target distribution corresponding to a new permutation.

The array redistribution problem brings with it another cost factor in the form of communication. Communication cost varies linearly and is modeled as $T_s + l * T_b$, where T_s is the startup cost, l the message size and T_b the per-byte transfer cost. Depending on the parameters T_s and T_b of a communication protocol, beyond a message size l, the transfer cost dominates the startup cost and the average per-byte cost converges to a constant. The message size beyond which there is little change in the communication cost is called the communication threshold 2^c. Note that as in the case of the read and write thresholds, the message size chosen for a specific instance of an algorithm may be below the threshold, if it cannot be improved upon. The communication characteristics of various systems have been widely studied and we do not discuss them here. For the following discussion, it is assumed that there are 2^p processors. The uppermost p rows of any permutation matrix correspond to the elements that constitute the processor identifier. The lowermost c elements of the address vector correspond to the communication threshold. The terms read, write and communication thresholds will be used interchangeably to refer to the size of I/O and r, w and c least significant elements in the address vector respectively. The reference will be clear from the context.

The formulation of the parallel redistribution involves four permutation matrices — read, write, in-memory permutation and communication. Extending the template for the formulation of read, write and in-memory permutation discussed in Section III to the parallel domain we get

$$R_i = I_p \oplus A_{2*n-r-p} \oplus I_r \quad r \leq m$$
$$P_i = I_{2*n-m} \oplus B_m$$
$$W_i = I_p \oplus C_{2*n-w-p} \oplus I_w \quad w \leq m$$

which indicates that R_i, W_i and P_i cannot permute the elements corresponding to the processor identifier. Only communication can permute the elements corresponding to the processor identifier. The permutation corresponding to communication is of the form

$$C_i = \begin{pmatrix} D_{2*n-c} & 0 \\ 0 & I_c \end{pmatrix}$$

where D describes the permutations done by communication.

Note that there are some restrictions on C_i similar to those on R_i, W_i and P_i as discussed in Section III. C_i cannot permute between address elements corresponding to in-memory and out-of-memory data (the elements corresponding to the processor identifier are special and will be discussed below). Any permutation except those involving the the processor identifier can be performed by P_i and W_i. Therefore, we place additional restrictions on C, so that it can only involve permutations required to change the processor identifier. In most cases, c is smaller than r and w and we assume the same.

Array redistribution can involve permutations of three kinds. First is the exchange of address vector elements that are part of the processor identifier. This effect is achieved by an exchange of all the data between processors. An equivalent effect could be achieved by relabeling the processors. But this does not obviate the problem as the same situation arises when there are multiple arrays which are aligned with respect to one another. This, or other constraints, might involve such an exchange that cannot be handled by relabeling.

Second is the exchange involved when elements within the communication threshold are to become part of the processor identifier. Any permutation involving the elements beyond the communication threshold is performed by an all-to-all personalized collective communication operation. If the number of elements within the communication threshold that are to become elements corresponding to the processor identifier is greater than $m - c$, then a sequence of in-memory permutation and communication operations are carried out. Each in-memory permutation operation moves as many elements from within the communication threshold to be beyond the threshold as possible. These elements are then made part of the processor identifier by a scatter operation. This process is repeated until there are no more elements in the least significant c address elements that are to be part of the processor identifier.

Thus any element already part of the processor identifier or within the least significant m elements (memory size), that is to be part of the processor identifier can be made part of the processor identifier in a single pass.

A more complicated operation is required when trying to permute the elements corresponding to the processor identifier and those beyond the least significant m elements. This involves a collect operation by each processor. The difference in handling this case and the previous two cases is that in the previous two cases all processors do the same operations throughout each pass. In this case, each processor collects all the data in memory from certain other processors in turn, in different iterations of the loop. But since all the collected data cannot be stored in memory, the data received from every processor is written to disk. This breaks the clear demarcation between the communication and write operations as they become interleaved. Since handling this case essentially involves writing the data to disk, this case is handled last.

But note that this may not be the most efficient way of performing the array redistribution. In handling the last case, each processor might receive data from a different set of processors in different iterations. Each receive is separated by a write to disk. Hence the communication and write times cannot overlap and could lead to very poor execution time especially when the number of processors is large. A more optimal implementation would be to schedule the communication among processors so that they overlap. A simple schedule would be for each processor to operate on data that has to be sent to one processor and then begin processing data to be sent to another processor. Each processor would be sending to and receiving data from a different processor, say as in a ring topology, ensuring overlap of communication and writing of data to disk. But this would modify the read and write access patterns by reordering of the reads and writes. The performance is not significantly impacted as the block size of I/O has been chosen to be large enough.

Hence all communication required to handle array redistribution can be done in a single pass. The implementation of this phase might involve a series of communications as just described. Henceforth we shall refer to C_i as the permutation effected on the linear address by the communication step and not get into the implementation details.

C. Combining Array Redistribution and Sequential Matrix Transposition

In this section, we combine the mechanisms considered until now to derive an algorithm for transposing out-of-core matrices which are distributed in a row-blocked fashion among multiple processors. Row-blocked distribution of data involves a permutation that is similar to transposition. Arbitrary data distribution would correspond to arbitrary permutations. The approach presented applies to arbitrary distributions but we formulate the row-blocked distribution to illustrate the procedure involved.

The parallel version of the algorithm differs from the sequential version only in the first pass. Since array redistribution can be performed in a single pass, it is performed in combination with the first pass of the sequential algorithm. Subsequent passes are identical to running the sequential algorithm on all the processors. The first pass for the parallel algorithm has the following flavor:

- Read as in sequential case (R_i).
- Perform in-memory permutation as in sequential case P_i.
- Perform array distribution, handling the different cases discussed above.
- Perform any permutation need to regroup the data.
- Write data to disk.

The subsequent passes are identical to those in the sequential case. Thus the parallel case does not lead to an increase in the number of passes in the form of additional reads or writes. The formulation for the first pass is shown below.

Conditions to be satisfied

$n \geq w$
$m > r \geq w$
$m > w$

Parameters

$$T = \prod_{t-1}^{0} T_i$$
$$T_i = \begin{cases} W_i P_i R_i & \text{if } i > 1 \\ W_i P_i' H_i P_i R_i & \text{otherwise} \end{cases}$$
$$s_0 = \begin{cases} \min(m - r - 1, w) & \text{if } r < n \\ \min(m - n - 1, w) & \text{if } r \geq n \end{cases}$$
$$s' = \begin{cases} p - (n - w) & \text{if } p > (n - w) \\ 0 & \text{otherwise} \end{cases}$$
$$t = \begin{cases} 1 & \text{if } s_0 + s' = w \\ 1 + \lceil \frac{w - s_0 - s'}{m - w} \rceil & \text{otherwise} \end{cases}$$
$$k = (w - s_0 - s') \bmod (m - w)$$
$$s_i = \begin{cases} k & \text{if } i = t - 1 \text{ and } k > 0 \\ m - w & \text{otherwise} \end{cases}$$

First pass $(i = 0)$

Case 1: $r \geq n$
$R_0 = I_{2n}$
$P_0 = I_{2n}$
$H_0 = L(I_p, L(I_{n-p}, I_p)) \oplus I_{n-p}$
$P_0' = I_{n-s_0} \oplus L(I_{s_0}, I_{n-w} \oplus L(I_{w-s_0}, I_{s_0}))$
$W_0 = I_p \oplus L(I_{n-p-s_0}, L(I_p, I_{n+s_0-p-w})) \oplus I_w$

Case 2: $r < n$ and $p \leq (n - w)$
$R_0 = I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r$
$P_0 = I_{2n-(r+s_0)} \oplus L(I_{s_0}, I_{r-w} \oplus L(I_{w-s_0}, I_{s_0}))$
$H_0 = L(I_p, L(I_{n-p-s_0}, I_p)) \oplus I_{n+s_0-p}$
$P_0' = I_{2n}$
$W_0 = I_p \oplus L(I_{n-p-s_0}, L(I_p, I_{n+s_0-p-w})) \oplus I_w$

Case 3: $r < n$ and $p > (n - w)$
$R_0 = I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r$
$P_0 = I_{2n-(r+s_0)} \oplus$
$\quad L(I_{s_0}, I_{r+p-n} \oplus L(I_{n-p-s_0}, I_{s_0}))$
$H_0 = L(I_n - w \oplus L(I_{p-(n-w)}, I_{n-p-s_0}), I_p) \oplus$
$\quad I_{n-p+s_0}$
$P_0' = I_{2n-w-s_0} \oplus L(I_{p-(n-w)}, I_{s_0}) \oplus I_{n-p}$
$W_0 = I_p \oplus L(I_{2n-p-w-s_0}, I_{s_0}) \oplus I_w$

There are some noticeable differences in the first pass as compared to that in the sequential algorithm. H_0 represents the array redistribution phase. The first pass consists of five phases. There are two in-memory permutation steps, P_0 and P_0', that prepare data for communication and regroup the data before writing to disk. This could involve a series of interleaved permutation and communication steps, where the communication steps satisfy the communication threshold. The amount of memory available should be at least double the read block size chosen. This is because communication requires buffers to store the received data in addition to the data read from disk, which might be sent to another processor in parallel. Increase in the number of processors implies an increase in the total available memory. If the number of processors is large enough, the communication phase can contribute to permuting the address elements within the write threshold. This factor is represented by s'. When the number of processors is large enough to contribute to permutation of the linear address, the communication and in-memory permutations involved are different from when it is not. The formulation handles all the different cases.

The transposition of a 4×4 array is illustrated in Table II. The array is distributed in a row-blocked fashion among 2 processors. The transposed array is also required to be in a row-blocked distribution. In terms of data, the top half of the matrix is stored in the first processor's disk, the bottom half on second processor's disk. The parameters of the algorithm are shown on the left hand side of the table. The actual data layout is shown on the right hand side. The algorithm requires two passes to transpose the array. In the first pass no elements within the write block size are permuted and no in-memory permutation is done. In the illustration these permutations are combined with other phases to simplify the diagrams, as they are just identity transformations. Upon completion of the first pass, the elements have been redistributed to the target processors. In the second pass, each processor permutes the array independently to arrive at the transposed form. Note that the reads and writes conform to the read and write block size thus ensuring a minimum contiguity in I/O.

#procs	Memory size (MB)					
	16	32	64	128	256	512
1	3406	3322	2265	2230	2003	2079
2	1536	1127	962	949	984	1006
4	740	542	484	483	475	474

TABLE III

EXECUTION TIME, IN SECONDS, ON THE ITANIUM 2 CLUSTER. ARRAY SIZE IS 16GB (N=64K).

#procs	Memory size (MB)					
	16	32	64	128	256	512
4	3448	3252	3213	2102	2907	2801
8	1470	1533	1469	921	985	1007

TABLE IV

EXECUTION TIME, IN SECONDS, ON THE ITANIUM 2 CLUSTER. ARRAY SIZE IS 64GB (N=128K).

VI. Experimental Results

In this section, we discuss the results obtained from implementing the parallel transposition algorithm. The transposition times were measured on the Itanium 2 cluster and on the IA32 cluster at the Ohio Supercomputer Center, whose I/O characteristics were discussed in Section II. Both clusters use the Myrinet [20] interconnection network. The implementation was out-of-place and used an auxiliary array.

The transposition times for different memory sizes and numbers of processors were measured. Tables III and IV show the transposition times on the Itanium 2 cluster for array sizes of 16GB (N=64K) and 64GB (N=128K). Tables V and VI show the transposition times on the IA32 cluster for the same array sizes.

In both systems the read threshold was much higher than N. So the execution time was influenced mainly by the write threshold. Increasing the memory decreases the number of I/O operations. If I/O operations were an effective measure of performance, doubling the memory size should halve the execution time. But the execution time improves little with increase in memory size, except when the larger memory size leads to a reduction in the number of passes. Reduction in the number of passes is accompanied by a significant reduction in the total execution time. This can be seen, for example, in the transition from 32MB to 64MB on one processor in Table V. The slight improvement seen with the increase in memory size is due to a reduction in the stride of writes. The write block size is reduced to be below the write threshold if it can reduce the number of passes and hence the total execution time. This is the case for 64MB memory on the one processor in Table V. In certain cases, the stride of write is so large as to wrap around and result in the writing of adjacent blocks before earlier written blocks have been flushed to disk. This leads to larger write block sizes and hence shorter total execution time. This trend can be especially seen in Table V at the transition in the number of passes, when the write block size is reduced to avoid an increase in the number of passes.

The parallel algorithm scales well with an increase in the number of processors. A slightly super-linear speedup can be seen in some cases. This is due to improved locality in I/O. Note that for an array size of 16GB and for four processors, the portion of each array in a processor is 4GB, equal to the memory size in the Itanium 2 cluster. But since there are three arrays the arrays are not fully cached in memory, making the results dependent on the caching mechanism. In some cases, an increase in the number of processors reduces the number of passes thus significantly reducing the execution time. This effect can be observed in Table V for a memory size of 32MB, when the number of processors is increased from one to two.

VII. Conclusions

In this paper, we have addressed the efficient parallel out-of-core transposition of matrices that are too large to

#procs	Memory size (MB)					
	16	32	64	128	256	512
1	7443	7386	3344	4254	4374	4223
2	3865	2098	2179	2253	2333	2207
4	1971	981	1142	1131	1165	1122
8	995	583	549	688	638	560

TABLE V

EXECUTION TIME, IN SECONDS, ON THE IA32 CLUSTER. ARRAY SIZE IS 16GB (N=64K).

#procs	Memory size (MB)					
	16	32	64	128	256	512
4	8122	6365	4948	3959	3855	3923
8	3523	3469	2695	2167	2046	1855

TABLE VI

EXECUTION TIME, IN SECONDS, ON THE IA32 CLUSTER. ARRAY SIZE IS 64GB (N=128K).

Parameters	Data layout
$n = 2,\ r = w = 1,\ m = 2,\ p = 1$ $s = \{0, 1\},\ s' = 0,\ t = 2$	$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$
pass=0 (Case 2) $R_0 = I_4$ $P_0 = I_4$ $H_0 = L(I_1, L(I_1, I_1)) \oplus I_1$ $P'_0 = I_4$ $W_0 = I_1 \oplus L(I_1, I_1) \oplus I_1$	$\overset{R_0P_0}{\Rightarrow} \begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array} \overset{H_0}{\Rightarrow} \begin{array}{cccc} 0 & 1 & 8 & 9 \\ 4 & 5 & 12 & 13 \\ \hline 2 & 3 & 10 & 11 \\ 6 & 7 & 14 & 15 \end{array} \overset{P'_0W_0}{\Rightarrow} \begin{array}{cccc} 0 & 1 & 4 & 5 \\ 8 & 9 & 12 & 13 \\ \hline 2 & 3 & 6 & 7 \\ 10 & 11 & 14 & 15 \end{array}$
pass=1 $R_1 = I_4$ $P_1 = I_2 \oplus L(I_1, I_1)$ $W_1 = I_1 \oplus L(I_1, I_1) \oplus I_1$	$\overset{R_1}{\Rightarrow} \begin{array}{cccc} 0 & 1 & 4 & 5 \\ 8 & 9 & 12 & 13 \\ \hline 2 & 3 & 6 & 7 \\ 10 & 11 & 14 & 15 \end{array} \overset{P_1}{\Rightarrow} \begin{array}{cccc} 0 & 4 & 1 & 5 \\ 8 & 12 & 9 & 13 \\ \hline 2 & 6 & 3 & 7 \\ 10 & 14 & 11 & 15 \end{array} \overset{W_1}{\Rightarrow} \begin{array}{cccc} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ \hline 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{array}$

TABLE II

ILLUSTRATION OF THE PARALLEL ALGORITHM.

fit in main memory. The problem was cast as a composition of two sub-problems: disk-based array redistribution, followed by concurrent independent uniprocessor transposition of disk-based arrays. The same algebraic framework was used for both steps. By viewing the transposition problem as an index permutation on the addresses of matrix elements, effective use was made of available main memory in optimizing the overall transposition time, rather than reducing the number of I/O operations, as previous algorithms have done. A solution to the out-of-core array redistribution problem was then provided using the same algebraic framework, combining to provide an algorithm for parallel out-of-core matrix transposition. Experimental measurements were provided, demonstrating the scalability of the proposed approach and the limited communication overhead. Extensions of this framework are being pursued for efficient index permutation of multi-dimensional arrays on parallel systems.

ACKNOWLEDGMENTS

We would like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

REFERENCES

[1] G. L. Anderson, “A stepwise approach to computing the multidimensional fast Fourier transform of large arrays,” *IEEE Transactions on Acoustics and Speech Signal Processing*, vol. 28, no. 3, pp. 280–284, 1980.

[2] D. H. Bailey, “FFTs in external or hierarchical memory,” *Journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, 1990.

[3] Synthesis of High-Performance Algorithms for Electronic Structure Calculations, http://www.cis.ohio-state.edu/ saday/TCE/index.html. [Online]. Available: http://www.cis.ohio-state.edu/~saday/TCE/index.html

[4] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan, “A high-level approach to synthesis of high-performance codes for quantum chemistry,” in *Proceedings of Supercomputing 2002*, 2003.

[5] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam, “Global communication optimization for tensor contraction expressions under memory constraints,” in *Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS)*, 2003.

[6] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, , and R. Harrison, “Space-time trade-off optimization for a class of electronic structure calculations,” in *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[7] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison, “Towards automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization,” in *Proc. of the Intl. Conf. on High Performance Computing*, 2001.

[8] NWChem, http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html. [Online]. Available: http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html

[9] J. O. Eklundh, “A fast computer method for matrix transposing,” *IEEE Transactions on Computers*, vol. 20, no. 7, pp. 801–803, 1972.

[10] W. O. Alltop, “A computer algorithm for transposing nonsquare arrays,” *IEEE Transactions on Computers*, vol. 24, no. 10, pp. 1038–1040, 1975.

[11] H. K. Ramapriyan, “A generalization of Eklundh’s algorithm for transposing large matrices,” *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1221–1226, 1975.

[12] R. E. Twogood and M. P. Ekstrom, “An extension of Eklundh’s matrix transposition algorithm and its application to digital signal processing,” *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 950–952, 1976.

[13] S. D. Kaushik, C.-H. Huang, R. W. Johnson, P. Sadayappan, and J. R. Johnson, “Efficient transposition algorithms for large matrices,” in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. ACM Press, 1993, pp. 656–665.

[14] J. Suh and V. K. Prasanna, “An efficient algorithm for out-of-core matrix transposition,” *IEEE Transactions on Computers*, vol. 51, no. 4, pp. 420–438, April 2002.

[15] T. H. Cormen, T. Sundquist, and L. F. Wisniewski, “Asymptotically tight bounds for performing BMMC permutations on parallel disk systems,” *SIAM Journal on Computing*, vol. 28, no. 1, pp. 105–136, 1998.

[16] J. S. Vitter and E. A. M. Shriver, “Algorithms for parallel memory I: Two-level memories,” *Algorithmica*, vol. 12, no. 2–3, pp. 110–147, 1994.

[17] Ohio Supercomputing Center, http://www.osc.edu. [Online]. Available: http://www.osc.edu

[18] A. Edelman, S. Heller, and S. L. Johnsson, “Index transformation algorithms in a linear algebra framework,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 12, pp. 1302–1309, 1994. [Online]. Available: citeseer.nj.nec.com/edelman94index.html

[19] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan, “On efficient out-of-core matrix transposition,” School of Computer and Information Science, The Ohio State University, Tech. Rep. OSU-CIRSC-9/03-T52, Sept 2003.

[20] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, “Myrinet: A gigabit-per-second local area network,” *IEEE Micro*, vol. 15, no. 1, pp. 29–36, February 1995.