On Efficient Out-of-core Matrix Transposition*

Sriram Krishnamoorthy Gerald Baumgartner

Daniel Cociorva Chi-Chung Lam

P. Sadayappan

School of Computer and Information Science

The Ohio State University

{krishnsr,gb,cociorva,clam,saday}@cis.ohio-state.edu

Abstract

This paper addresses the problem of 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 algorithm proposed decouples the algorithm from the matrix dimensions and associates it with the I/O characteristics of the system. The I/O characteristics of the system are used to determine the read and write block sizes. These I/O block sizes are chosen in order to optimize the total execution time. Experimental results are provided that demonstrate the

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

better performance of the new algorithm compared to previous known transposition algorithms in the literature.

1 Introduction

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 $M < N^2$, $M = O(N)$. Each element of the matrix is too small to read from disk and written back independently. The problem is to transpose the matrix stored in disk, when only a portion of the matrix can be brought into memory at any time. Matrix transpose is a key operation in various scientific applications. For example, the two-dimensional Fourier transform [2, 3] can be implemented as a one-dimensional Fourier transform along the rows, followed by a one-dimensional Fourier transform along the columns. 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 before the second pass.

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), to generate efficient parallel programs from high level expressions, for a class of computations expressible as tensor contractions [4, 6–8]. Often the tensors (essentially multi-dimensional matrices) are too large to fit in memory and must be disk-resident. Further, the desired layout of these tensors may be different for different programs that process them - the most efficient representation for the producer of a tensor object may not be the same as the layout for another program that consumes it. In such a case, the need to transpose the

disk-resident tensor object arises.

We are developing this algorithm to be used as part of a computational chemistry project at the Ohio State University . The project aims at developing a tool, TCE [13], for automatic synthesis of efficient parallel programs for a computation specified in high-level form by the user. The input for the tool is a set of tensor contractions and data on disk, obtained from another chemistry package, NWChem [12]. For a given problem, the solution is obtained by first running NWChem, and then TCE. For efficient execution TCE accesses the array in a certain fashion. This requires efficient transformation of the data from NWChem to the required format. Transposition is one of the possible transformations. Efficient transposition algorithms are critical in the improving overall execution time. In addition, when TCE is used on different machines, different transformations are required on the data from NWChem, which again requires efficient out-of-core matrix transpose algorithms.

This problem has been widely studied in the literature. A simple in-place element-wise approach to transpose the matrix is prohibitively expensive as long as each element is not large enough to be read (written) from (to) disk efficiently. The block transposition algorithm transposes the array in a single pass in $O(N^{3/2})$ I/O operations, where a pass is defined as accessing each element from disk exactlyonce. An in-place transposition algorithm requiring $O(N \log N)$ disk accesses was proposed by Eklundh [10]. This algorithm requires at least two rows to fit in memory. Extensions to the algorithm for rectangular matrices were presented in [1, 14, 16]. Kaushik et al. [11] improved upon these algorithms by reducing the number of read operations. Suh and Prasanna [15] 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. 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 disk access time, seek time plus latency, is very large (on the order of several milliseconds) compared to the per-byte transfer time (on 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, when the I/O blocks get large, the data transfer time becomes significant and can dominate the total access time. In such a situation smaller block sizes can be read/written without any additional I/O cost. But this might reduce the number of passes involved, thus improving performance. 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.

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. The parameters of the algorithm are determined based on the empirically measured I/O characteristics. 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. In Section 2, some of the existing matrix transposition algorithms for out-of-core matrices are discussed. The I/O characteristics of two systems are discussed in Section 3. In Section 4 the transposition problem is formulated using the matrix-vector product notation. Section 5 formulates the algorithms detailed in Section 2 and infers their performance, based on the I/O characteristics discussed in Section 3. The new algorithm is described in Section 6. Experimental results are presented in Section 7 . Section 8 concludes the paper.

2 Matrix Transposition Algorithms

In this section, we discuss some of the out-of-core matrix transposition algorithms from the literature. The pseudo-code for the algorithms is given with focus on the I/O operations performed in each algorithm. These algorithms are formalized in the next section.

Consider a square matrix of dimension $N = 2^n$. Let the number of elements that can be brought into memory at any time be $M = 2^m$. The memory can hold $B = 2^b$ rows, say, of the input matrix, i.e., $B = M/N$. Each algorithm runs in a certain number of passes. Each pass involves reading the entire array from disk and writing it back. In each pass, the algorithm goes through a sequence of steps, each of which involves three phases–reading data into memory, permuting the read data and writing it back to disk. All algorithms proceed as a sequence of steps in each pass. A step is defined as the operations performed between reading a portion of data into memory and writing it back to disk, including the read and write operations. All algorithms in the literature work on disjoint ranges of data in each step. Note that the algorithms discussed can be employed to transpose matrices whose size is not a power of 2. We capture the basic idea of each algorithm and provide a formulation for the out-of-core matrix transposition problem. This formulation is used

to arrive at a better algorithm.

The block-transposition algorithm is a single-pass algorithm for matrix transposition. The algorithm blocks the input matrix into smaller matrices and recursively transposes the embedded matrices.

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1k} \\ A_{21} & A_{22} & \dots & A_{2k} \\ \vdots & \vdots & & \vdots \\ A_{k1} & A_{k2} & \dots & A_{kk} \end{pmatrix}$$

$$A^T = \begin{pmatrix} A_{11}^T & A_{21}^T & \dots & A_{k1}^T \\ A_{12}^T & A_{22}^T & \dots & A_{k2}^T \\ \vdots & \vdots & & \vdots \\ A_{1k}^T & A_{2k}^T & \dots & A_k^T \end{pmatrix}$$

The block transposition algorithm is shown below. Each step of the algorithm involves $\sqrt{M}$ read and write operations, each of $\sqrt{M}$ elements. The algorithm reads and writes at different locations in the matrix in any given step, thus requiring the destination array to be different from the source array, i.e., the algorithm is out-of-place.

Algorithm 1: Block Transposition Algorithm

(1) **for** $i = 0$ **to** $N/\sqrt{M} - 1$

(2) **for** $j = 0$ **to** $N/\sqrt{M} - 1$

(3) Read data in region $[i*\sqrt{M} : (i+1)*\sqrt{M}-1][j*\sqrt{M} : (j+1)*\sqrt{M}-1]$.

(4) Transpose in memory.

(5) Write data to region $[j*\sqrt{M} : (j+1)*\sqrt{M}-1][i*\sqrt{M} : (i+1)*\sqrt{M}-1]$.

Eklundh's algorithm [10] does the transposition in-place in n/b passes. This algorithm requires that $n \bmod b = 0$. The algorithm is shown below. Each step of the algorithm involves M/N read and write operations, each involving N elements.

Algorithm 2: Eklundh's Algorithm

(1) **for** $i = 0$ **to** $n/b - 1$

(2) **for** $j = 0$ **to** $N^2/M - 1$

(3) Read M/N rows into memory starting with the $(\lfloor (j/B^i) \rfloor * B^{i+1} + j\%B^i)$th row at a stride of B^i rows.

(4) Permute in memory.

(5) Write to the rows from which the data was read.

Kaushik et al. [11] improve upon Eklundh's algorithm by combining the reads. The algorithm is shown below. It is an out-of-place algorithm. In each step of the algorithm, one read of M elements and M/N writes, each of N elements, are performed. In the most general case, N is factorized into $s_0 * \ldots * s_{t-1}$ such that for any s_i, s_i rows fit in memory. The algorithm runs in t passes. Kaushik et al. provide a solution when only one row fits in memory, which cannot be handled by Eklundh's algorithm. They also provide a mechanism to use the maximum available memory.

Algorithm 3: Kaushik et al.'s Algorithm
Input: t, $\{s_0, \ldots, s_{t-1}\}$

Output: -

(1) **for** $i = 0$ **to** $t - 1$

(2) **for** $j = 0$ **to** $N^2/M - 1$

(3) Read M/N contiguous rows starting at $(j * M/N)$th row.

(4) Permute in memory.

(5) **for** $k = 0$ **to** $M/s_i - 1$

(6) Write s_i rows starting at $(k * s_i)$th row in memory to the array in disk starting at the $(j * (M/N)/s_i + k)$th row at stride N/s_i rows.

Suh and Prasanna's algorithm improves further upon Kaushik's algorithm in two ways. It reduces the in-memory permutation time by replacing in-place permutation by a series of collect operations, in which the data to be written is collected into a buffer. The algorithm also reduces the number of I/O operations by 'chunking' the writes. The writes that would have been done at different offsets are done contiguously. This increases the write size and reduces the number of writes. Each write operation in the ith pass writes $z_i * N$ elements instead of the N element written in Kaushik et al.. In the subsequent pass, the data that should have been written contiguously is 'collected' by performing a sequence of reads. Thus the number of reads is increased from one in Kaushik et al.. This mechanism balances the number of reads and writes. The optimal value for z_i was determined to be $\sqrt{s_i}$, at which point the number of writes equals the number of reads and the total number of I/O operations is minimum. In the algorithms discussed so far, each element is read into memory exactly once in each pass. On the other hand, each pass i in this algorithm performs redundant reads to first collect the rows, that have been separated by z_{i-1} rows by the

previous write, into memory and then performs the permutation. Thus this algorithm reduces the total number of I/O operations at the expense of a potential increase in the amount of data to read.

Algorithm 4: Suh and Prasanna's Algorithm
Input: t, $\{s_0, \ldots, s_{t-1}\}$,$\{z_0, \ldots, z_{t-1}\}$

Output: -

(1) **for** $i = 0$ **to** $t - 1$

(2) **for** $j = 0$ **to** $N^2/M - 1$

(3) Collect M/N rows that have been separated by z_{i-1} rows in the previous pass.

(4) Permute in memory.

(5) Write the permuted data to disk with z_i rows in each I/O operation.

3 I/O Characteristics

In this section, the I/O characteristics of two systems are discussed. The characteristics of the two systems will be used to derive the parameters for our algorithm.

We studied the variation of read and write times with change in the size and stride of I/O. The I/O characteristics of a Pentium II PC (henceforth referred to as *PC*) and an HP zx6000 workstation were studied. Their configuration is shown in Table 1. The PC is an ordinary end-user system. The HP zx6000 is a high-end system used as part of a cluster at the Ohio Supercomputer Center (OSC) [5].

The read and write characteristics of the PC are shown in Fig. 1 and Fig. 2, respectively, and those of the HP zx6000 system are shown in Fig. 3 and Fig. 4, respectively. A stride of one

System	Configuration			
	Processor	Memory	OS	Compiler
Pentium II PC	Pentium II(300 MHz)	128MB	linux 2.4.18-3	gcc version 2.96
HP zx6000	Dual Itanium-2(900 MHz)	4GB	linux 2.4.18	gcc version 2.96

Table 1: Configuration of the systems used for I/O characterization.

corresponds to sequential access.

For the PC, the figure shows that for reads of block size above 1MB and writes of block size above 64KB, stride has little effect on the per-byte access time. Similar observations are made for the HP zx6000, for read and write block sizes of 1MB.

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.

An important consequence of this observation is that if the thresholds are smaller than N, fractions of a row can be read and written without any additional penalty. This reduction in the read and write block size in turn decreases the amount of work involved in transposing an array and will be explained later. In the extreme case, if each element is large enough to allow efficient I/O of individual elements, a simple single-pass element-wise transposition would be possible. Thus the unit of I/O depends on the system I/O characteristics and not on the matrix size.

This observation also shows that for I/O sizes above the threshold the number of I/O operations does not reflect the actual performance of the algorithm. An algorithm might involve more I/O operations but be faster than another algorithm with fewer I/O operations due to this effect.

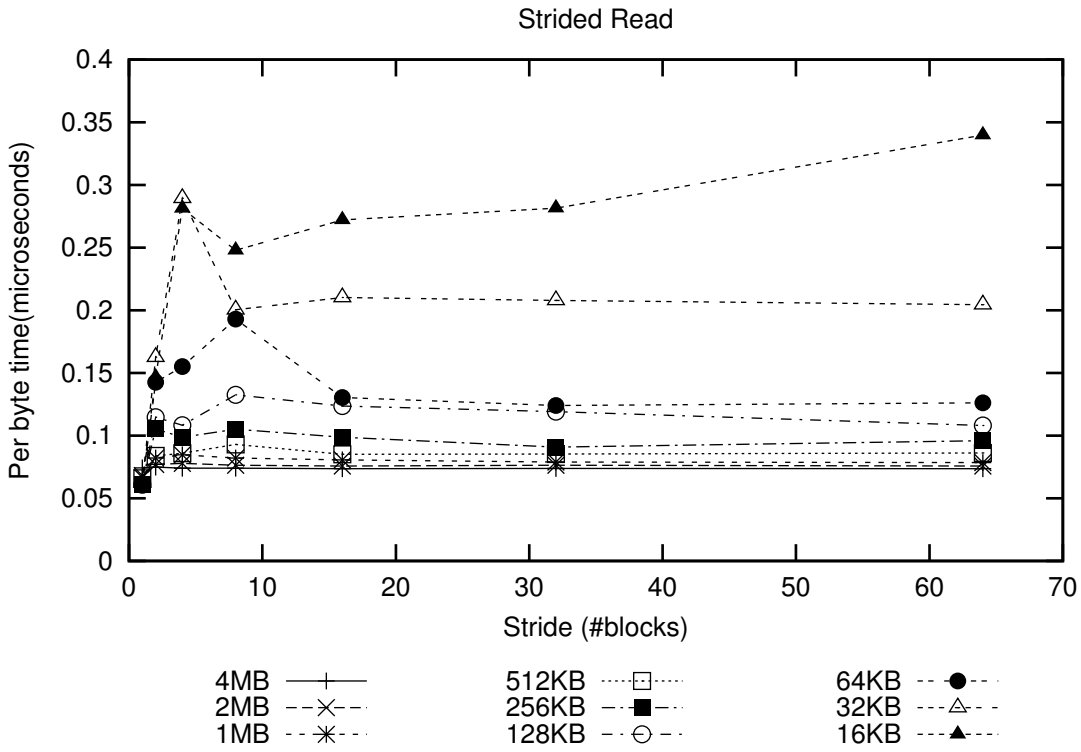

Figure 1: Strided read times for the Pentium II PC.

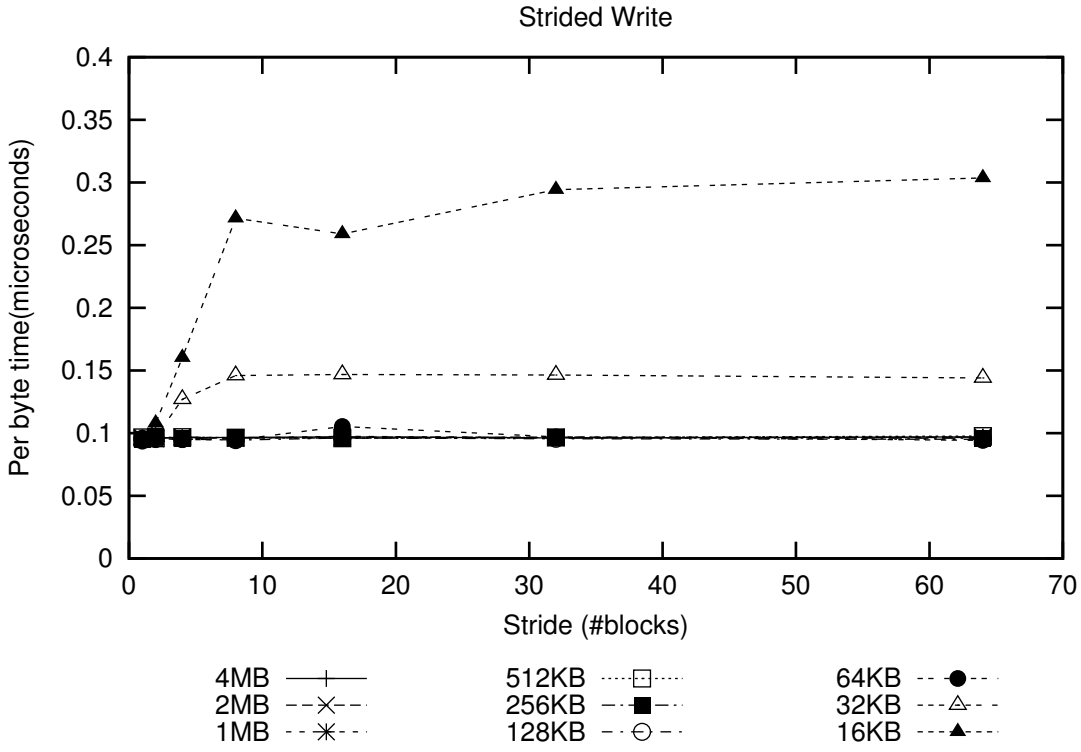

Figure 2: Strided write times for the Pentium II PC.

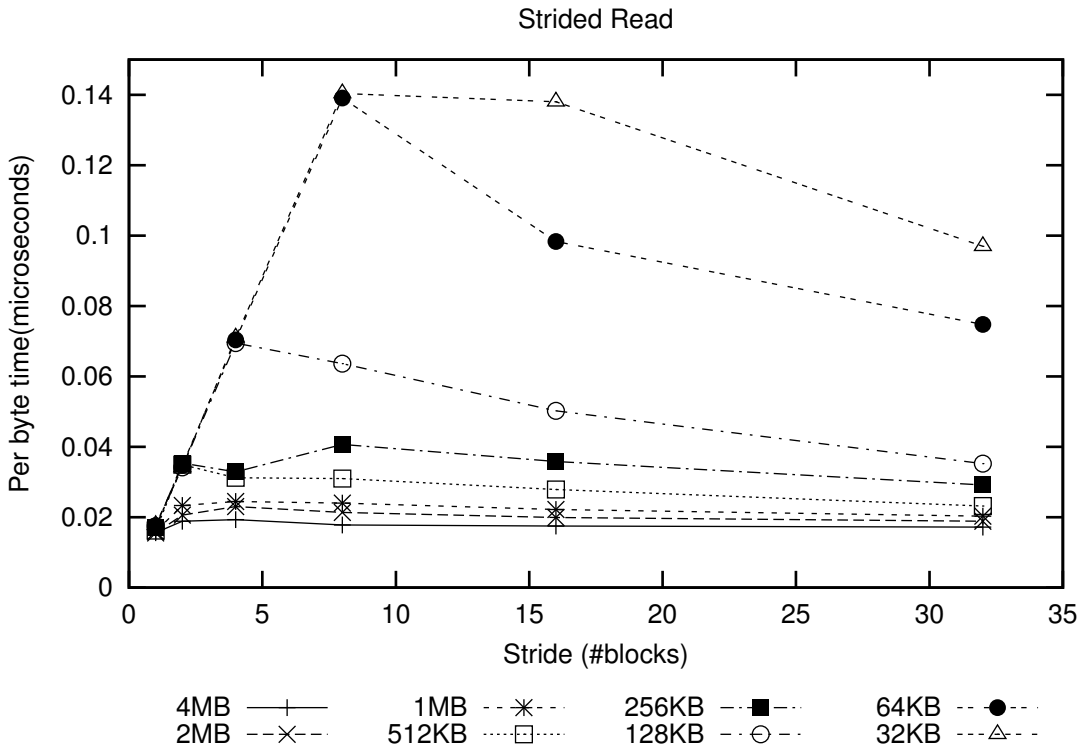

Figure 3: Strided read times for the HP zx6000 system.

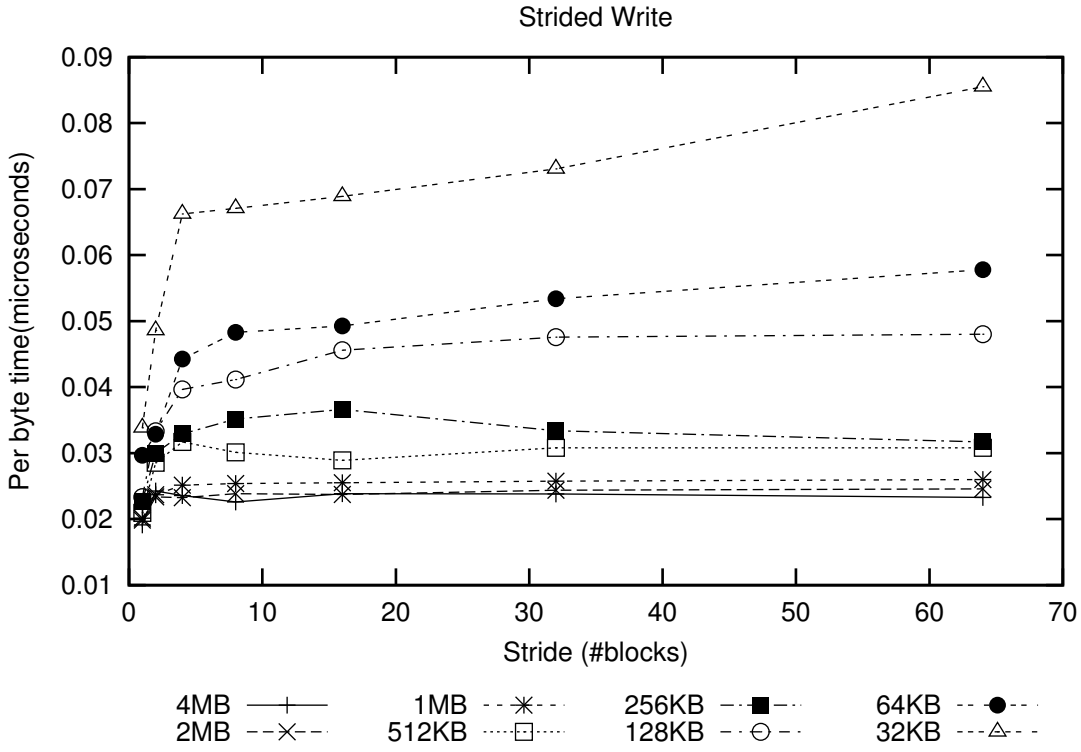

Figure 4: Strided write times for the HP zx6000 system.

4 Matrix Vector Product Formulation of Transposition Algorithms

In this section, matrix transposition algorithms are formulated using the matrix vector product notation. The formulation is based on the matrix-vector notation used in [9]. This section provides a generic formulation for transposition algorithms. The formulation for individual algorithms is given in Section 5.

Transposition of a matrix can be viewed as an interchange of the indices of the matrix. This is a particular case of a general class of index transformation algorithms.

Each element of the array has a linear address vector obtained by concatenating the column index bits to the row index bits. Transposition corresponds to a transformation of this linear address vector and can be represented by a transformation matrix.

The identity of the transformation is I_{2n}. Matrix transposition is defined as the transformation of the address vector i

$$i \rightarrow Ti$$

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

We use the following notation in the discussion. Given two matrices A and B

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

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

$L(I_n, I_n)$ is the desired transformation. Since the entire array does not fit in memory, $L(I_n, I_n)$ is factorized into a number of transformation matrices such that the transformation effected by each of the matrices can be done with the memory available. The following discussion provides the matrix vector formulation of various out-of-core matrix transposition algorithms in the literature.

Any out-of-core matrix transposition algorithm consists of three phases–read, permute and write. Each phase is modeled by a transformation matrix. These phases are repeated on disjoint sets of data 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

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

where W_i is the transformation matrix corresponding to write, R_i is the transformation matrix corresponding to read and P_i corresponds to in-memory permutation for the ith pass. t is the number of passes. 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. Each algorithm is defined by the parameters t, W_i, P_i and R_i. Different algorithms define these parameters in terms of other parameters which are specific to that algorithm.

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 transformation matrix must correspond to a transformation of the given out-of-core 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 restriction of R_i shows that the unit of read must be at least $R(= 2^r)$ elements. The transformation in read as modeled by A determines the pattern of reads. Similarly for writes. The restriction for P_i shows that in-memory permutation can transform only address elements corresponding to the data elements in memory. Given these parameters for an algorithm it can be implemented as

Algorithm 5: Generic Transposition Algorithm
(1) **for** $i = 0$ **to** $t - 1$

(2) **for** $j = 0$ **to** $2^{2*n-m} - 1$

(3) Read M elements at address $R_i^{-1}(j)$ /*Might involve multiple reads*/

(4) Permute data in memory according to P_i.

(5) Write M elements at address $W_i(j)$ /*Might involve multiple writes*/

5 Performance Analysis From Formulation

In this section, we analyze the performance of various algorithms in the literature based on their formulation. The parameters R_i, P_i and W_i of each algorithm are determined and are used to analyze the performance of the algorithm.

The running time of an algorithm depends on the read, write and in-memory permutation time. The I/O time depends on the size and stride of I/O. r and w determine the read and write block size, respectively, and hence are important parameters. In addition, the stride of access plays an important role, as demonstrated by the I/O characteristics. The strides of reads and writes are determined by the A and C sub-matrices, respectively. For some algorithms it might be possible that A (C) could be written as $D \oplus I_k$, for some matrix D. In such cases the read (write) sizes can be larger than 2^r(2^w) elements.

5.1 Block Transposition

The $2n$ bits are partitioned into four components

$$I_{2n} = I_{RH} \oplus I_{RL} \oplus I_{CH} \oplus I_{CL}$$

such that $RL + CL = m$. The values of the parameters are

$$t = 1$$

$$R_i = (I_{RH} \oplus L(I_{RL}, I_{CH}) \oplus I_{CL})$$

$$P_i = (I_{RH+CH} \oplus L(I_{RL}, I_{CL}))$$

$$W_i = (L(I_{RH}, I_{CH} \oplus I_{CL}) \oplus I_{RL})$$

It is a single pass algorithm. The algorithm reads 2^{RL} elements and writes 2^{CL} elements in one I/O operation. Note that the above values for W_i, P_i and R_i satisfy the requirements. The algorithm reads (writes) 2^{RL} (2^{CL}) elements. Generally the components are chosen such that $RL = CL$ producing square blocks. The I/O size is typically $O(\sqrt{M})$ elements. Even for large

memory sizes, this algorithm would fall short of reaching the threshold. This leads to high I/O cost, making this a very inefficient algorithm. Note that this algorithm is not inefficient due to the large number of I/O operations involved ($O(N^{3/2})$), but because of the small I/O size. For systems with memory large enough to make I/O sizes larger than the threshold this algorithm can be very efficient.

But a more effective way of choosing RL and CL would be to minimize the total I/O cost. Thus the problem becomes

$$RL + CL = m$$

Minimize cost(read)+cost(write)

A cost model for read and write can be derived from the I/O characteristics of the system. These cost equations can be used to arrive at the best components for the algorithm.

5.2 Eklundh's Algorithm

Eklundh's algorithm [10] has the following formulation

$$b = m - n$$

$$I_{2n} = I_{n-b} \oplus I_b \oplus I_n$$

$$t = n/b$$

$$R_i = (I_{n-(i+1)*b} \oplus L(I_b, I_{i*b}) \oplus I_n)$$

$$P_i = (I_{n-b} \oplus L(I_b, L(I_{n-(i+1)*b}, I_b)) \oplus I_{i*b})$$

$$W_i = (I_{n-(i+1)*b} \oplus L(I_{i*b}, I_b) \oplus I_n)$$

R_i^{-1} (=R_i^T as R_i is a permutation matrix) and W_i are identical, indicating that the algorithm can be executed in-place. Each phase (read, write and permute) of the algorithm depends on the pass in which it occurs. This algorithm reads and writes N elements in each I/O operation. This is independent of the I/O characteristic of the underlying system. Hence the algorithm might perform well on some machines and poorly on others. Also, unless the matrix size (N^2) is of the order of terabytes, N is lower than the read threshold for the systems analyzed in Section 3.

5.3 Kaushik's Algorithm

Kaushik's algorithm discussed in Section 2 can be formulated in the following manner, given $s_i = b, 0 \leq i < t$.

$$b = m - n$$

$$I_{2n} = I_{n-b} \oplus I_b \oplus I_n$$

$$t = n/b$$

$$R_i = (I_{2n})$$

$$P_i = (I_{n-b} \oplus L(I_n, I_b))$$

$$W_i = (L(I_{n-b}, I_b) \oplus I_n)$$

This is an out-of-place algorithm involving t identical passes. The algorithm reads M elements in one I/O operation thus comfortably achieving the read threshold. Each write involves N elements. This algorithm improves on Eklundh's by reducing the read costs by performing sequential

reads of large size. This algorithm does not take advantage of the I/O characteristics of the system, by writing smaller block sizes than a row, if little additional cost is incurred. The in-memory permutation phase in every pass involves element-wise permutation, unlike Eklundh's algorithm which moves larger blocks with each pass. This could increase the in-memory permutation cost as compared to Eklundh's algorithm.

5.4 Suh and Prasanna's Algorithm

This algorithm does not fit into the formulation discussed as it might involve redundant reads. This algorithm improves upon Kaushik's by reducing the number of I/O operations and increasing the I/O size of writes. The chunking of writes distributes the rows that will have to be operated on in one step. In the next pass, the data to be operated in one step is collected. When the rows to be collected are too far apart to be brought into memory in one I/O operation redundant I/O is involved. In trying to reduce the number of I/O operations, the algorithm does not take this into account. The algorithm could have benefited uniformly from chunking if it avoided redundant I/O, by taking the memory available into account. The algorithm does not choose the chunking factor based on the I/O characteristics. Hence the performance of the algorithm can vary dramatically depending on the parameters of the problem. The algorithm usually benefits from an increase in memory, since an increase in memory reduces the redundant data movement involved.

6 Our Algorithm

Our algorithm tries to minimize the I/O time involved by choosing the parameters for the algorithm appropriately. 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 formulation of all algorithms discussed so far requires m and n to provide a concrete list of operations to perform to transpose the input out-of-core matrix. Our algorithm requires two additional parameters, namely, the read block size (2^r) and the write block size(2^w), which are chosen close to the threshold. The exact value depends on the number of passes that would be involved given some block sizes. Smaller block sizes incur more I/O time but might potentially reduce the number of passes, thus significantly reducing the total 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.

The number of rows to be transformed in each pass is determined as the maximum possible. The chunking factor, the factor which determines the extent of chunking similar to that in Suh's algorithm, is chosen so that no redundant reads are incurred. This provides the benefits of the chunking factor such as increasing the I/O size without increasing the total I/O time.

In previous algorithms, the basic unit of I/O was a row. The I/O transformation matrices are of the form $A \oplus I_n$, while the required transformation $L(I_n, I_n)$ involves exchanging the upper and lower n address elements in the address vector. The nature of the I/O transformation matrices prevents any effective tranformation 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 $(b \ldots n-1) \leftrightarrow$

$(n + b \ldots 2 * n - 1)$ can be done in the read and/or write phases. This reduces the number of address vector elements to be transformed in the in-memory permutation phase and might result in a reduction in the number of passes.

Our algorithm is formulated as shown below. The unit of each read and write is at least 2^r and 2^w elements respectively. Except in the first pass, the algorithm reads M elements in each read operation. In the first pass, the read and write phases transform the address vector elements $(w : n - 1)$ to their appropriate positions. The remaining address vector elements are transformed in the in-memory permutation phase of all the passes and the I/O phases of the remaining passes.

Conditions to be satisfied

$n \geq w$

$m \geq r \geq w$

$m > w$

Parameters

$$s_0 = \begin{cases} \min(m - r, w) & \text{if } r < n \\ \min(m - n, w) & \text{if } r \geq n \end{cases}$$

$$t = \begin{cases} 1 & \text{if } s_0 = w \\ 1 + \lceil \frac{w - s_0}{m - w} \rceil & \text{otherwise} \end{cases}$$

$$s_i = \begin{cases} (w - s_0) \bmod (m - w) & \text{if } i = t - 1 \text{ and } (w - s_0) \bmod (m - w) > 0 \\ m - w & \text{otherwise} \end{cases}$$

$$z_i = \begin{cases} 0 & \text{if } i = t - 1 \\ m - (w + s_{i+1}) & \text{otherwise} \end{cases}$$

First pass $(i = 0)$

Case 1: $r \geq n$

$R_0 = (I_{2n})$

$P_0 = (I_{n-s_0} \oplus L(I_{s_0}, I_{n-w} \oplus L(I_{w-s_0}, I_{s_0})))$

$W_0 = (L(I_{n-s_0}, I_{n-w+s_0-z_0}) \oplus I_{w+z_0})$

Case 2: $r < n$

$R_0 = (I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r)$

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

$W_0 = (L(I_{n-s_0}, I_{n-w+s_0-z_0}) \oplus I_{w+z_0})$

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

$sp_i = \sum_{j=0}^{i-1} s_j$

$R_i = (I_{2n})$

$P_i = (I_{2n-(w+s_i+z_{i-1})} \oplus L(I_{s_i}, I_{z_{i-1}} \oplus L(I_{w-sp_i-s_i}, I_{s_i})) \oplus I_{sp_i})$

$W_i = (I_{n-w} \oplus L(I_{s_{i-1}-z_{i-1}} \oplus L(I_{n-s_{i-1}-s_i}, I_{z_{i-1}}), I_{s_i-z_i}) \oplus I_{w+z_i})$

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 does an in-place in-memory transposition. Suh uses collect buffers to collect data to be written in each

Parameters	Data layout
$n = 2, r = w = 2, m = 3$ $s = \{1, 1\}\ z = \{0, 0\}$ $t = 2$	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
pass=0 (Case 1) $R_0 = I_4$ $P_0 = I_1 \oplus L(I_1, L(I_1, I_1))$ $W_0 = L(I_1, I_1) \oplus I_2$	0 1 2 3 $\overset{R_0}{\Rightarrow}$ 4 5 6 7 8 9 10 11 12 13 14 15 0 4 2 6 $\overset{P_0}{\Rightarrow}$ 1 5 3 7 8 12 10 14 9 13 11 15 0 4 2 6 $\overset{W_0}{\Rightarrow}$ 8 12 10 14 1 5 3 7 9 13 11 15
pass=1 $sp_1 = 1$ $R_1 = I_4$ $P_1 = I_1 \oplus L(I_1, I_1) \oplus I_1$ $W_1 = L(I_1, I_1) \oplus I_2$	0 4 2 6 $\overset{R_1}{\Rightarrow}$ 8 12 10 14 1 5 3 7 9 13 11 15 0 4 8 12 $\overset{P_1}{\Rightarrow}$ 2 6 10 14 1 5 9 13 3 7 11 15 0 4 8 12 $\overset{W_1}{\Rightarrow}$ 1 5 9 13 2 6 10 14 3 7 11 15

Table 2: Illuatration of our algorithm.

write operation. The locality of the permutation operation can be improved by optimizations such a blocking. We use collect operations to perform the permutation, as this was empirically found to take less time than in-memory permutation. Unlike Kaushik et al.'s and Suh and Prasanna's algorithm, in-memory permutation moves larger blocks of data in each succeeding pass. This further reduces the in-memory permutation cost.

The transposition of a 4×4 array by our algorithm is illustrated in Table 2. The parameters are on the left hand side. The actual data layout after each transformation is shown on the right hand side.

7 Experimental Results

In this section, we discuss the results obtained from implementing Kaushik's, Suh's and our algorithm. The algorithms were compared on the systems whose configurations are shown in Table

1. The tests involved transposition of a matrix of integers, each of size 4 bytes. The amount of main memory allocated to the data was varied and the total execution time was measured. On the PC platform, the algorithms were run for array sizes of 256MB and 1GB. On the HP zx6000 machine, the algorithms were run for a data size of 16GB. This corresponds to array dimension N being 8192 and 16384 on the PC platform and 65536 on the HP zx6000 machine. Tests were not performed for larger array sizes due to hard drive capacity limitations. For larger arrays, the comparative performance of the algorithms would be same as when the memory available is small.

For the array sizes considered the read threshold is much higher than N. So only the write threshold influenced the variation in the running time of our algorithm. The results are shown in Table 4, Table 5 and Table 3.

Our algorithm consistently outperforms the other two algorithms. An important characteristic is that our algorithm achieves a low execution time for a given data size for a wide range of memory sizes. The I/O sizes are varied so that the resultant cost of reduced memory size on the overall execution time is incremental at best. Other algorithms need larger memory for performance to become comparable to our algorithm. Thus for larger array sizes, which would correspond to relatively smaller memory sizes, our algorithm would perform much better than other algorithms. Optimizations such as chunking and handling of larger blocks for in-memory permutation result in better performance of our algorithm even for large memory sizes. This property of our algorithm can be used to implement further optimizations, such as out-of-place in-memory permutations for all passes, where we read data into half the available memory and permute the data into the other half.

The performance of Kaushik et al.'s algorithm does not improve steadily with an increase in

memory. This is due to in-place in-memory permutation and the fact that the algorithm does not maximize read and write block sizes with change in memory size.

Suh's algorithm has a longer execution time compared to the the our algorithm, but it improves with an increase in memory. This is due to two reasons, both resulting from the inability of the algorithm to take advantage of the I/O characteristics of the system. Firstly, the algorithm does not modify the read and write block sizes to reduce the number of passes required. Secondly, chunking of writes, introduced to balance the number of read and write operations, results in redundant I/O operations and hence huge penalties for larger arrays. For larger memory sizes, chunking does not result in redundant I/O operations and the algorithm's performance approaches that of our algorithm, but does not perform better than our algorithm.

Our algorithm makes efficient use of memory available by adjusting the I/O block sizes and the chunk sizes. By conscientious use of chunking and management of I/O block sizes, our algorithm performs much better than both Kaushik's and Suh's. For example, for memory size of 32MB in Table 3, the number of passes increases to three from two for Kaushik's and Suh's algorithms. In our algorithm the write block size is reduced to retain the number of passes, thus resulting in a comparatively smaller increase in total execution time.

The number of I/O operations in Kaushik's and Suh's is proportional to N^2/M. If number of I/O operations was an indicator of performance, doubling M should half the execution time of the algorithm. But the results do not exhibit such behavior, showing that number of I/O operations does not truly model the total execution time of the out-of-core matrix transposition algorithms in literature.

Algorithm	Memory Size					
	32MB	64MB	128MB	256MB	512MB	1GB
Our	2897	2378	2529	2298	2237	2007
Suh's	14437	10811	9756	11464	2623	2516
Kaushik's	7750	4423	3944	4138	4051	4057

Table 3: Execution time (in seconds) on the HP zx6000 for the three algorithms for data size of 16GB

Algorithm	Memory Size				
	4MB	8MB	16MB	32MB	64MB
Our	124	117	119	116	122
Suh's	318	171	128	134	132
Kaushik's	208	226	223	207	214

Table 4: Execution time (in seconds) on the Pentium II PC for the three algorithms for data size of 256MB

Algorithm	Memory Size				
	4MB	8MB	16MB	32MB	64MB
Our	621	510	479	465	480
Suh's	1526	2004	927	738	533
Kaushik's	1079	685	829	859	910

Table 5: Execution time (in seconds) on the Pentium II PC for the three algorithms for data size of 1GB

8 Conclusions

In this paper, we addressed the efficient transposition of matrices that are too large to fit in main memory. We addressed the drawbacks of previously proposed algorithms and used empirically derived I/O characteristics of the system in guiding the algorithm. We formulated the out-of-core matrix transposition problem as an index permutation on the addresses of matrix elements and inferred the effect of various components of the formulation on the I/O time and in-memory permutation time. We devised an algorithm by choosing the design parameters that minimize time involved in the I/O and in-memory permutation phases of the algorithm. Thus we improved the overall transposition time, rather than reducing the number of I/O operations, as previous algorithms have done. Experimental measurements were provided, demonstrating the superiority of the proposed approach to existing methods.

Acknowledgments

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

References

[1] W. O. Alltop. A computer algorithm for transposing nonsquare arrays. *IEEE Transactions on Computers*, 24(10):1038–1040, 1975.

[2] G. L. Anderson. A stepwise approach to computing the multidimensional fast Fourier transform of large arrays. *IEEE Transactions on Acoustics and Speech Signal Processing*, 28(3):280–284, 1980.

[3] D. H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, 1990.

[4] G. Baumgartner, D.E. 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] Ohio Supercomputing Center. http://www.osc.edu.

[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, 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.

[8] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. 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.

[9] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1302–1309, 1994.

[10] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 20(7):801–803, 1972.

[11] 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*, pages 656–665. ACM Press, 1993.

[12] NWChem. http://www.emsl.pnl.gov:2080/docs/nwchem/nwchem.html.

[13] Synthesis of High-Performance Algorithms for Electronic Structure Calculations. http://www.cis.ohio-state.edu/˜ saday/TCE/index.html.

[14] H. K. Ramapriyan. A generalization of Eklundh's algorithm for transposing large matrices. *IEEE Transactions on Computers*, 24(12):1221–1226, 1975.

[15] Jinwoo Suh and V. K. Prasanna. An efficient algorithm for out-of-core matrix transposition. *IEEE Transactions on Computers*, 51(4):420–438, April 2002.

[16] 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*, 25(12):950–952, 1976.