Efficient Layout Transformation for Disk-based Multidimensional Arrays

Sriram Krishnamoorthy[1], Gerald Baumgartner[2], Chi-Chung Lam[1], Jarek Nieplocha[3], and P. Sadayappan[1]

[1] Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA
{krishnsr,clam,saday}@cse.ohio-state.edu
[2] Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803, USA
gb@csc.lsu.edu
[3] Computational Sciences and Mathematics
Pacific Northwest National Laboratory, Richland, WA 99352, USA
jarek.nieplocha@pnl.gov

Abstract. I/O libraries such as PANDA and DRA use blocked layouts for efficient access to disk-resident multi-dimensional arrays, with the shape of the blocks being chosen to match the expected access pattern of the array. Sometimes, different applications, or different phases of the same application, have very different access patterns for an array. In such situations, an array's blocked layout representation must be transformed for efficient access. In this paper, we describe a new approach to solve the layout transformation problem and demonstrate its effectiveness in the context of the Disk Resident Arrays (DRA) library. The approach handles re-blocking and permutation of dimensions. Results are provided that demonstrate the performance benefit as compared to currently available mechanisms.

1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of the machine. Due to the extremely large seek time relative to the per-word transfer time for disk access, it is imperative that I/O be done using contiguous blocks of disk resident data. To optimize performance in collective I/O operations between arrays located on disk and in distributed main memory of parallel computers [1], I/O libraries like PANDA [2, 3] and DRA [4] use a blocked layout representation for the disk-based multidimensional arrays instead of the dimension-ordered representation used typically for the representation of multidimensional arrays in main memory. Thus, the disk-based multidimensional array is partitioned into a number of multidimensional blocks or "bricks", and the elements within a brick are linearized using some dimension order. Such a bricked representation of disk-based multidimensional arrays permits efficient access as long as the accessed regions mostly contain full bricks.

However, the access patterns to some disk-based multidimensional arrays in two successive phases (or the access pattern of the producer and the consumer) are so different that no choice of brick shape will allow for efficient access. An example is the out-of-core 2D Fast Fourier Transform (FFT), where the array is accessed by columns in one phase and by rows in the other. The multi-dimensional FFT [5, 6] can be implemented as a series of one-dimensional FFTs, one along each dimension. Another example illustrating very different access patterns is with image data in three and four (including time) dimensions. The production of data from scanning occurs plane by plane. However, examination of the time evolution of a 3D block of data requires a very different access pattern than that by which the data was generated. In isosurface construction in three and four dimensions, the data is typically produced in a row-major format by scanning or simulation. The amount of memory available determines the amount of data generated between writes to disk, and hence limits the blocking possible. To efficiently perform computations on the stored data in a parallel system, the data might have to transformed into a different blocked form [7]. Thus there are situations where performance can be greatly improved by transforming the layout of a multidimensional array on disk to match the application's access pattern.

Our primary motivation for addressing the layout transformation 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)[8], to generate efficient parallel programs from high level expressions, for a class of computations expressible as tensor contractions [9–14]. Often the tensors (essentially multi-dimensional arrays) are too large to fit in memory and must be disk-based. The input tensors are often generated by other quantum chemistry packages such as NWChem [15], with a layout quite different from that needed for efficient processing by the TCE-generated code.

This paper describes an approach to efficient transformation of data between disk-based multidimensional arrays. Experimental results indicate that this approach delivers comparable or better performance than other techniques currently used in practice, that are based on reading data from one disk-based array to distributed main memory, in-memory data transformation, and then writing data to the destination disk array. For example, improvements exceeding 80 percent were observed on a Linux cluster.

The paper is organized as follows. Section 2 describes the DRA framework, within which we implement our solution to the layout transformation problem. The array re-blocking problem is explained is detail in Section 3. Section 4 presents the proposed approach for efficient layout transformation. In Section 5, experimental results are presented. Section 6 concludes the paper.

2 Disk Resident Arrays

The Global Arrays (GA) library [16] [17] provides a shared-memory programming model in which data locality is explicitly managed by the programmer. Explicit function calls are used to transfer data between global address space and local storage. It is similar to distributed shared-memory models in providing an explicit acquire-release protocol, but differs with respect to the level of explicit control in moving blocks of data in multidimensional arrays between remote global storage and local storage. The functionality provided by GA has proved useful in the development of large scale parallel quantum chemistry suites such as NWChem [15] (which contains over a million lines of code).

The Disk Resident Arrays (DRA) model [18] extends the GA NUMA programming model to secondary storage. It provides a disk-based representation for multi-dimensional arrays and functions to transfer blocks of data between global arrays and disk resident arrays. DRA, along with GA, provides a unified programming model for handling different levels of the memory hierarchy in which the user controls the location of data in the memory hierarchy. This has been shown to provide high performance while providing a programming model that is simpler than message passing.

Henceforth, we shall use GA and DRA to refer both to the library and the arrays handled by them. The reference will be clear from the context.

3 The Layout Transformation Problem

Internally, the data in a DRA is stored in a blocked fashion. When a DRA is created, a typical request shape/size can be specified. This is used to determine the shape of the basic layout block or "brick". The shape of the brick is chosen to match the specified access shape. The size of the brick is chosen as a compromise between two competing objectives: 1) optimize disk I/O bandwidth - this requires that the brick size be large enough to amortize the disk seek time and 2) minimize wastage of disk I/O - since I/O is done in units of the basic block (brick), small bricks imply less wastage at the boundaries of the DRA regions being read/written.

An application might have an access pattern that is very different from the organization of the DRA on disk. This can happen when an application uses the output of another program, or because different phases of the same program use different access patterns. This can be handled by creating another copy of the disk resident array to match the new request size and transformed dimensions.

We have implemented the copy routine, referred to as *NDRA_Copy*, together with dimension permutation. The routine takes as input the source and target DRA handles and the dimension permutation to be performed. Henceforth, the data in the DRA corresponding to the dimensions of blocking in the source and target arrays are referred to as the source and target blocks respectively.

The disk array layout transformation problem we consider here is a generalization of the out-of-core matrix transposition problem. Out-of-core matrix transposition has been widely studied in the literature. The algorithms perform out-of-core transposition by making passes through the entire array a number of times. During each pass through the array, each element of the source array is read once and each element of the target array is written once. Each pass consists of a series of steps in which a portion of data from the source array is brought into memory, permuted and written to the target out-of-core array. Different steps in a pass operate on disjoint sets of data. The block transposition algorithm is a single-pass algorithm in which a 2-D tile of data is brought into memory, transposed and written to disk. Since the different row segments of a 2D tile are not contiguous on disk, this could be extremely inefficient unless the tile size is very large. Eklundh [19] proposed a multi-pass algorithm, in which the minimum unit of I/O is a row. The number of passes in the algorithm is proportional to the array dimensions. Kaushik et al. [20] reduced the number of read operations and increased the read block size compared to Eklundh's algorithm. Sun and Prasanna [21] proposed an algorithm that minimized the total number of I/O operations, while potentially increasing the total volume of I/O. Krishnamoorthy et al. [22] formulated these algorithms in a tensor product notation and derived a generic algorithm that attempts to minimizes the total execution time by taking into consideration the I/O characteristics of the system, and subsequently extended it to a multi-processor system, in which each processor has a local disk [23] .

Most of the above approaches assume the array dimensions and the memory size to be powers-of-2. This assumption, coupled with the fact that the required transformation is a transposition, allows different steps in the re-blocking process to operate on disjoint sets of data. In each step, the set of data read into memory form an integral number of write blocks, which are written out. So no data is retained across steps during the transposition. When arbitrary blocking, array dimensions and memory sizes are to be handled, it may not be possible to process and write out all the data read into memory in a given step. Some data either needs to be discarded and re-read, increasing the I/O cost, or needs to be retained, increasing the memory requirement. The memory cost for retaining the data unused from a step depends on the order of traversal of dimensions, and hence is not straight forward. The out-of-core transposition algorithms involve I/O of blocks of data at specific strides, which is fixed for a pass. This regularity allows accurate prediction of the I/O cost. The in-memory permutation of data can be modeled as a bit-permutation on the linear address space of the data stored in disk. This provides a regular structure to the in-memory computation. In the general case, in-memory permutation corresponds to a series of collect operations for combining portions of different read blocks to create a write block. The simplicity in the cost models for the power-of-2 transposition problem makes it amenable to mathematical treatment as done in [22].

In the next section, we detail our approach to solving the generalized re-blocking problem.

4 Algorithm Design

The disk array layout transformation problem is modeled as an I/O optimization problem. The total I/O cost is to be minimized, subject to the amount of physical memory available. The cost model and the algorithm to obtain the multi-pass solution are explained in this section. In the ensuing discussion, we shall consider an n-dimensional matrix of dimensions $< d_1, \ldots, d_n >$. The matrix is blocked in brick shape $< s_1, \ldots, s_n >$. The target matrix has the same ordering of dimensions as the source but is blocked using bricks of shape $< t_1, \ldots, t_n >$. The source and target bricks are assumed to be of size that is large enough for efficient access from/to disk. DRA typically uses a brick size of around 1 Mbyte. Reads from the source disk array are assumed to be in units of the source brick, and writes to the target disk array are done in units of the target brick.

4.1 Solution Approach

If feasible, a single-pass solution (in which each element is read and written exactly once) would provide the minimum I/O cost. But the memory requirement for a single-pass solution might exceed the physical memory available. In this case, we either need to choose a multi-pass solution or perform redundant I/O in one pass. In this sub-section, we present the intuition behind the design of our algorithm. We begin with a basic single-pass algorithm and determine its I/O and memory cost. We then incrementally improve the single-pass algorithm to lower the memory requirement and/or the I/O cost. The multi-pass solution is discussed in a subsequent sub-section.

Consider the region $< 0 - LCM(s_1, t_1), \ldots, 0 - LCM(s_n, t_n) >$. This region contains an integral number of source and target blocks along all the dimensions. Thus the data in the source matrix from this region maps onto complete blocks in the target matrix. This region can be processed independent of other such blocks, without any redundant I/O. We shall refer to such regions as *LCM blocks*. If the amount of physical memory were large enough to hold an LCM block, then a single-pass solution is clearly possible - read in source blocks contained in an LCM block into memory, construct the target blocks corresponding to the data in memory, and write them into the target array. The I/O cost is defined as the I/O required per element of the source array. This algorithm has the minimum I/O cost of one read and one write per element of the source array. Assuming the read and write operations are equivalent the I/O cost is two units per element.

The memory cost is the size of the LCM block. Since arbitrary re-blocking needs to be supported, the source and target block sizes could have arbitrary dimensions (provided their total size corresponds to a reasonable block size for I/O on the target file system). Hence the LCM block can be arbitrarily large and might not fit in physical memory. We can improve the single-pass algorithm to handle this scenario without increasing the I/O cost. Instead of reading entire LCM blocks into memory, the algorithm reads in a set of blocks of data from the source matrix and writes out those target blocks that can be completely constructed from the data available in memory. Any data in memory that cannot be used to construct a complete target block is retained in memory. Any source block in an LCM block contributes to target blocks within the same LCM block. Hence no data needs to be retained across LCM blocks. The algorithm processes all the data in one LCM block before processing any other LCM block. The algorithm requires enough memory to retain unused data and read in additional data for processing. The additional data read into memory for processing must be enough to write at least one target block to disk. This is referred to as the Max block and corresponds to $< M_1, \ldots, M_n >$ where $Max_i = \lceil (\max(s_i, t_i)/s_i) \rceil * s_i$. The algorithm traverses each LCM block along each of the dimensions and processes data in units of the Max block. The buffer to store the unused data is partitioned into one buffer per dimension. Unused data from a Max block along a dimension needs to be retained until the adjacent Max block along that dimension is processed. Thus the amount of unused data to be retained depends on the order of traversal of dimensions. Along the dimension traversed first, only data unused from the last processed Max block needs to be stored. Other dimensions require more data to be retained. A static memory cost model is used, in which the sizes of buffers used to store data is determined before the transformation begins. The maximum memory required to perform the transformation is the sum of the size of the Max block and the sizes of the buffers.

$$\text{MemCost} = \textstyle\sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} Max_i$$

where bsize_i represents the size of buffer to store unused data along the i-th dimension.

Let $< T_1, \ldots, T_n >$ be the order of traversal of dimensions. The unused data along a dimension (say T_i) is an n-dimensional region. For a given dimension i, the size of this region along dimension j can be as much as $\text{LCM}(s_{T_j}, t_{T_j})$ for $j < i$, but is bounded above by Max_{T_j} for $j > i$. Hence, the size of the buffer to store the unused data along a dimension T_i is bounded by

$$\text{bsize}_{T_i} = \textstyle\prod_{j=1}^{n} S_j$$

$$S_j = \begin{cases} \text{LCM}(s_{T_j}, t_{T_j}) & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ Max_{T_j} & \text{if } j > i \end{cases}$$

where U_i be the maximum unused data that needs to be stored along dimension i. Since U_i must be smaller than both s_i and t_i, and for every s_i elements along dimension i brought into memory, at least $\text{gcd}(s_i, t_i)$ elements must be written out, we have

$$U_i = \min(s_i, t_i) - \text{gcd}(s_i, t_i)$$

As can be seen from the above formulae, the sizes of the unused buffers is proportional to the LCM block dimensions. This could lead to situations in which the memory requirement still exceeds the available memory. In this case, there are two options to be considered. A multi-pass solution could be determined, which is discussed later, or a single-pass solution that performs redundant read of data can be designed.

We propose a single-pass algorithm that differs from the discussion above in one respect. Instead of traversing an entire LCM block, a smaller template is chosen. No unused data is stored across templates. A template is an integral number of write blocks along all dimensions. There is no redundant read within a template. But unlike LCM blocks, templates might have source blocks on their boundaries that straddle across two templates. This results in redundant reads across templates, increasing the I/O cost. The memory cost is reduced and is given by:

$$\text{MemCost} = \sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} Max_i$$
$$\text{bsize}_{T_i} = \prod_{j=1}^{n} S_j$$
$$S_j = \begin{cases} \text{templ}_{T_j} & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ Max_{T_j} & \text{if } j > i \end{cases}$$

where templ_i represents the size of the template along the i-th dimension.

For a two-dimensional array, the memory cost due to the unused buffers is $U_1 * Max_2 + \text{LCM}(s_1, t_1) * U_2$ if dimension 1 is traversed fi rst; otherwise, it is $U_2 * Max_1 + \text{LCM}(s_2, t_2) * U_1$. In an n-dimensional array, the traversal order is determined by sorting the dimensions by comparing these expressions.

The minimum template size corresponds to a target block. In this case, the memory requirement is reduced to a Max block. Thus the necessary condition for the existence of a single-pass solution is that the Max block fi t in memory.

The I/O cost is multiplicative along the dimensions. Within an LCM block, the number of source blocks that need to be reread is the number of templates minus one, which is $(\text{LCM}(s_i, t_i) - \text{templ}_i) - 1$. Therefore, the I/O cost of re-blocking is given by templ_i is

$$\text{IOCost} = \prod_{i=1}^{n} \text{IOCost}_i$$
$$\text{IOCost}_i = \frac{\left(s_i * \left(\frac{\text{LCM}(s_i,t_i) - \text{templ}_i}{\text{templ}_i}\right) + \text{LCM}(s_i, t_i)\right)}{\text{LCM}(s_i, t_i)}$$

In reality, the LCM along a dimension might be larger than the length of the array along the dimension, in which case we replace the LCM by the array dimension. Note that the array dimensions are not considered while determining U_i. Hence, U_i does not provide an exact estimate, but only an upper bound on the memory requirement. Note that though the I/O cost for the single-pass solution is increased, the total I/O cost could be reduced due to a decrease in the number of passes.

4.2 Template Determination for Single-pass Solution

Both the I/O cost and the memory cost are affected by the choice of the template. In this section, we discuss the algorithm used to determine the template sizes. The template is a set of write blocks along all the dimensions. It can range in size from one write block, to an LCM block. For re-blocking an n-dimensional array, the template needs to be determined from an n-dimensional solution space. A template is a feasible solution if its processing does not require more memory than available. The algorithm exploits the characteristics of the solution space and the optimization function.

Consider a template A. An enclosing template is defi ned as a template that is at least as large as the given template in all the dimensions. Let B be an enclosing template of A. From the memory cost equations, it can be seen that the memory required to process A cannot exceed that required to process B. Conversely, processing B requires at least as much memory as processing A. This implies that once a template has been determined to require more memory than available (an infeasible solution), no enclosing templates needs to be considered. This relation separates the solution space into a feasible and an infeasible solution space (where the surface of separation approximates to a hyperbola when $n = 2$).

The I/O cost has a similar characterization. The I/O cost equation shows that decreasing the template size along any dimension increases the I/O cost. Thus the I/O cost of template *A* is at least as much as that of template *B*. This implies that when searching through the solution space, no template that is enclosed by a feasible template needs to be considered. Thus the optimal solution resides on the surface separating the feasible and infeasible solution spaces.

Our algorithm to determine the template for a single-pass solution involves three phases. The algorithm begins with the LCM block as the template and tests for feasibility. If an LCM block is the feasible solution, it is chosen as the template. Otherwise, a solution is chosen that is just feasible, i.e. , increasing the template size along any dimension violates the memory constraint. This is a solution on the boundary between the feasible and infeasible solution spaces and hence is a candidate solution. From this solution, we perform a steepest descent to arrive at a local minimum in the search space. Note that other optimization algorithms that can optimize on a surface can be used. The algorithm used is shown in Fig.2.

4.3 Multi-pass Solution Determination

When a single-pass solution does not exist or is too expensive, a multi-pass solution is chosen by determining intermediate block sizes. An intermediate disk-based array is used to store the intermediate results. Hence, additional disk space equal to the size of the arrays is required. The multi-pass solution proceeds as repeated execution of the single-pass algorithm, for the source and target block sizes determined for that pass. The source block size of the fi rst pass is the block size of the source array. The target block size of the last pass if the block size of the target array. The skew between the source and target block sizes decreases as the multi-pass solution proceeds from one pass to the next. The intermediate block size are chosen to effect the maximum re-blocking possible with the available memory.

A simple heuristic is used to determine the intermediate tile sizes for the multi-pass solution. Two candidate intermediate block sizes are considered. The fi rst candidate intermediate block size is the geometric mean of the source and target block sizes. This block size is "equidistant" from the source and target block sizes. This can be an effective intermediate block size of for solutions with an even number of passes. The second intermediate block size is, in fact, a pair of block sizes. Let s_i and t_i be the source and target block sizes along dimension i. The intermediate block sizes chosen are $s_i^{2/3} * t_i^{1/3}$ and $s_i^{1/3} * t_i^{2/3}$. This pair of intermediate block sizes can be effective for solutions with an odd number of passes. These two options allow a more refi ned search for intermediate block sizes. Without the second choice, any solution that requires an odd number of passes, each transforming to an intermediate block "equidistant" from the previous one, might be harder to achieve. Higher order intermediates were not considered as solutions with a larger number of passes seldom occur in practice and can be handled by a combination of these choices.

Once the intermediate block(s) are determined, the multi-pass solution is determined recursively for transforming from source to intermediate, and intermediate to target block sizes. In the case of two intermediate blocks, the transformation between the intermediate blocks is determined as well. The algorithm for determining the multi-pass solution is shown in Fig. 3.

Consider an instance of the matrix re-blocking problem in which the source and target arrays are blocked as $< 32, 9 >$ and $< 5, 16 >$, respectively. The array dimensions are much larger than the blocking and hence are not considered. The Max block is $< 32, 16 >$ and the unused data along each dimension is bounded by $< 4, 8 >$. The solution

```
Input:  (1) Source and target block sizes [s] and [t],
        (2) Template size [templ]
Output: (1) Total memory cost (2) Dimension traversal order [T]
1)  foreach dimension i
2)    L[i] = LCM(s[i], t[i])
3)    U[i] = min(s[i], t[i]) - gcd(s[i], t[i])
4)    M[i] = ceil(max(s[i], t[i])/s[i])*s[i]
5)  Sort dimensions into array T such that
      forall i<j => U[T[i]]*M[T[j]] + L[T[i]]*U[T[j]] <
      U[T[j]]*M[T[i]] + L[T[j]]*U[T[i]]
6)  memCost=0
7)  foreach dimension i
8)   pdt=U[T[i]]
9)   foreach j<i
10)     pdt *= L[T[j]]
11)   foreach j>i
12)     pdt *= M[T[j]]
13)   memCost += pdt
```

Fig. 1. Pseudo-code to determine the memory cost for a given template size

to the re-blocking problem depends on the memory available. An LCM block contains $LCM(s_1,t_1)*LCM(s_2,t_2)$=23040 elements. When enough memory is available to hold an LCM block, the re-blocking can be performed by reading in an entire LCM block and writing out the target blocks. But if the memory can hold $U_2*Max_1+\mathrm{LCM}(s_2,t_2)*U_1+Max_1*Max_2$=1344 elements, it is suffi cient to hold all unused data when an LCM block is processed. The second dimension is traversed fi rst in the re-blocking procedure. If the memory available is lesser, say enough to hold just 900 elements, a single-pass solution with a template size of $<120,6>$ elements is used for the re-blocking. When the memory size is 800, a two-pass solution with an intermediate tile size of $<12,12>$ is determined. The template for the fi rst pass is $<96,12>$, and that for the second pass is $<60,48>$.

5 Experimental Results

In order to evaluate the effectiveness of the proposed approach, we compared the time for layout transformation using our implementation with the time for transformation using currently available mechanisms. The present interface to DRA is through Global Arrays. When a DRA is to be copied to another DRA with different blocking, the source array is read into a GA one section at a time, and written into the same section of the target array. This is a single-pass solution. The basic unit of access, i.e. the shape and size of the GA needs to be determined. The size is determined independent of the blocking of the source and target arrays to equal the amount of available physical memory. We evaluated three options for the shape of the GA used. One option was to use the largest square tile that fi ts within the available memory. If the blocking of the DRAs is known, the GA can be chosen to be a multiple of the source block size or the target block size. These three options are labeled Basic(square), Source Directed and Target Directed, respectively. The implementation of the new approach is labeled NDRA_Copy.

We evaluated the mechanisms on the OSCBW machine at the Ohio Supercomputer Center [24]. Each node in the cluster has Dual AMD Athlon MP processors (1.533 GHz) and 2GB of memory. The PGI pgcc 4.0-2 compiler was used to generate the executables.

```
Input:  Source and target block sizes [s] and [t],
Output: Template size for single-pass solution, if it exists.
Support Routines: MemCost(templ) - Memory cost for processing
                                   the given template
                  DiskCost(templ) - I/O cost for processing
                                    the given template
                  MemoryExceeded(templ) - returns true if the
                                          template is infeasible
1) Initialize template to LCM block
2) Reduce template size along along all dimensions equally
     (in units of write block size) until the template is a
     feasible solution.
3) If no feasible solution is found return "No solution exists"
4) Adjust the template size so that increasing the template size
     along any dimension makes it infeasible.
5) Repeat the following steps
6)   Among adjacent template sizes choose the one that has the
       maximum rate of decrease in I/O cost to increase
       in memory cost.
7)   Determine a feasible solution that leads to the least
      increase in disk I/O cost from the chosen template.
8)   If the feasible solution found has lesser I/O cost than the
       current template, choose that as the current template.
       Otherwise return the current template as the solution.
```

Fig. 2. Algorithm to determine template size for a single-pass solution.

```
Input:  Source and target block sizes [s] and [t],
Output: Sequence of intermediate block sizes [seqB], order of
        traversal of dimensions for each pass I/O cost
1)  Determine the cost (a) of single-pass solution
2)  foreach dimension i
3)    B1[i] = floor(sqrt(s[i]*t[i]))
4)    B2[i] = (s[i]^(2/3)*t[i]^(1/3))
5)    B3[i] = (s[i]^(1/3)*t[i]^(2/3))
6)  Determine the cumulative cost (b) of multi-pass solutions
      for re-blocking from s to B1 and B1 to t by recursively
      calling this routine.
7)  Determine the cumulative cost (c) multi-pass solutions
      for re-blocking from s to B1, B1 to B2 and B2 to t
      by recursively calling this routine.
8)  If no multi-pass solution exists
      return "no solution exists"
9)  Choose solution with least I/O cost from (a), (b) and (c)
10) If the single-pass solution has the minimum cost
11)   Return the solution with the order of traversal
        determined by invoking memCost
12) else
13)   Concatenate the sequence of solutions returned by the
        two parts of the solution with the minimum I/O cost.
```

Fig. 3. Pseudo-code to determine a multi-pass solution

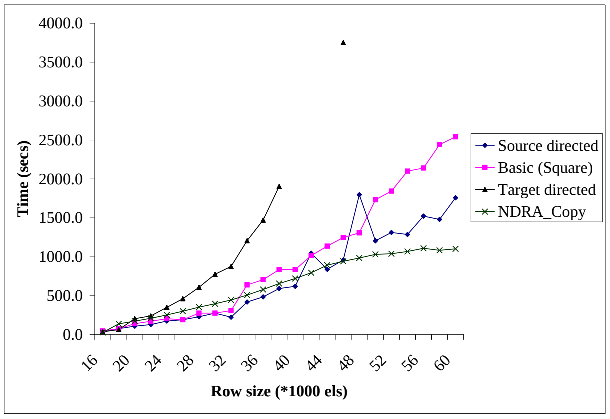

Fig. 4. Execution time to transform set-of-rows blocking to set-of-columns blocking

Two sets of experiments were conducted. In one, a set of rows form the blocks in the source array. The target array is blocked as a set of columns. The corresponding results are shown in Fig. 4. The second experiment involved the reverse - transforming from a set-of-columns blocking into a set-of-rows blocking, and its results are shown in Fig. 5. The number of rows (or columns) in a block was chosen such that the block size was greater than 1MByte, the typical brick size chosen by DRA for this system. For example, for a $< 4096, 4096 >$ array, where each element is of size four bytes, set-of-rows blocking corresponds to a block size of 1 Mb, with each brick holding a $< 64, 4096 >$ block of data; and a set-of-columns layout corresponds to a 1 Mb brick holding a $< 4096, 64 >$ block of DRA data.

In both the experiments, the array size was increased from 16000 to 60000 in steps of 2000 and all four mechanisms were evaluated. For our approach, the template size is determined automatically using the algorithms described in Section 4. The x-axis in the graphs shows the array dimension in number of elements. The y-axis shows the transformation time in seconds. We were unable to run larger experiments due to the limited amount of disk space available on the local disks (around 60GB).

In transforming the set-of-rows bricks into a set-of-columns bricks, the target directed method performs signifi cantly worse than other approaches. This is because the data to be read in is not contiguous on disk. The DRA reads in entire blocks of data to ‘collect’ the data into the global array. This leads to exponential increase in cost. Due to this obvious trend, this approach was evaluated with only certain sample array dimensions. The source directed approach performs better, as DRA implementation allows writes of partial blocks, if it is contiguous on disk. Though the unit of write is small, it still performs better than the target directed approach. With larger array dimensions, both the source directed and basic (square) approach increase in cost.

Our implementation performs better than the alternatives. The relative performance benefi t of our new approach increases with the size of the array. It starts with a single-pass solution and then uses a two-pass solution for arrays with dimensions larger than $32,000$. But the execution time increases gradually and is not drastically affected by the exact problem instance at hand. Unlike the other three approaches, our implementation performs comparably for both the transformations evaluated.

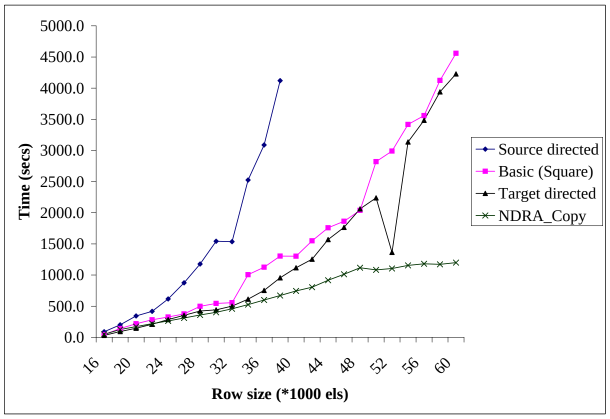

Fig. 5. Execution time to transform set-of-columns blocking to set-of-rows blocking

6 Conclusions

In this paper we proposed a new approach to efficient transformation of the blocked layout of multidimensional disk-based arrays. The proposed approach was implemented as a new copy primitive within the DRA I/O library. Experimental results demonstrated the benefit of the new approach over existing mechanisms. The extension of this approach to the parallel context is being pursued.

Acknowledgments

We acknowledge the support from the National Science Foundation through the Information Technology Research program (CHE-0121676) and Pacific Northwest National Laboratory. In addition, We would like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

References

1. Chen, Y., Foster, I., Nieplocha, J., Winslett, W.: Optimizing collective I/O performance on parallel computers: A multisystem study. In: 11th ACM Intl. Conf. on Supercomputing. (1997)
2. Seamons, K.E., Winslett, M.: Multidimensional array I/O in Panda 1.0. The Journal of Supercomputing **10** (1996) 191–211
3. The Panda Project – Data Management for High-Performance Scientific Computation. (`http://drl.cs.uiuc.edu/panda/`)
4. Foster, I., Nieplocha, J.: Disk Resident Arrays: An array-oriented I/O library for out-of-core computations. In Buyya, R., Jin, H., Cortes, T., eds.: Disk Arrays and Parallel I/O: Theory and Practice. IEEE Computer Society Press (2001)
5. Anderson, G.L.: A stepwise approach to computing the multidimensional fast Fourier transform of large arrays. IEEE Transactions on Acoustics and Speech Signal Processing **28** (1980) 280–284

6. Bailey, D.H.: FFTs in external or hierarchical memory. Journal of Supercomputing **4** (1990) 23–35
7. Kazhiyur-Mannar, R., Wenger, R., Crawfis, R., Dey, T.K.: Adaptive resolution isosurface construction in three and four dimensions. Technical Report OSU-CISRC-7/03–TR38, School of Computer and Information Science, The Ohio State University (2003)
8. Tensor Contraction Engine – Synthesis of High-Performance Algorithms for Electronic Structure Calculations. (`http://www.cse.ohio-state.edu/~saday/TCE/`)
9. Baumgartner, G., Bernholdt, D., Cociorva, D., Harrison, R., Hirata, S., Lam, C., Nooijen, M., Pitzer, R., Ramanujam, J., Sadayappan, P.: A high-level approach to synthesis of high-performance codes for quantum chemistry. In: Proceedings of Supercomputing 2002. (2003)
10. Cociorva, D., Gao, X., Krishnan, S., Baumgartner, G., Lam, C., Sadayappan, P., Ramanujam, J.: Global communication optimization for tensor contraction expressions under memory constraints. In: 17th International Parallel & Distributed Processing Symposium (IPDPS). (2003)
11. Cociorva, D., Baumgartner, G., Lam, C., Sadayappan, P., Ramanujam, J., Nooijen, M., Bernholdt, D., , Harrison, R.: Space-time trade-off optimization for a class of electronic structure calculations. In: Proc. of ACM SIGPLAN PLDI 2002. (2002)
12. Cociorva, D., Wilkins, J., Baumgartner, G., Sadayappan, P., Ramanujam, J., Nooijen, M., Bernholdt, D., Harrison, R.: 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)
13. Krishnan, S., Krishnamoorthy, S., Baumgartner, G., Cociorva, D., Lam, C., Sadayappan, P., Ramanujam, J., Bernholdt, D., Choppella, V.: Data locality optimization for synthesis of efficient out-of-core algoritms. In: Proc. of the Intl. Conf. on High Performance Computing. (2003)
14. Krishnan, S., Krishnamoorthy, S., Baumgartner, G., Lam, C., Ramanujam, J., Choppella, V., Sadayappan, P.: Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. In: Proc. of 18th Intl. Parallel & Distributed Processing Symposium (IPDPS). (2004)
15. High Performance Computational Chemistry Group: NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6. Pacific Northwest National Laboratory, Richland, Washington 99352–0999, USA. (2004)
16. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: a portable programming model for distributed memory computers. In: Supercomputing. (1994) 340–349
17. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: A nonuniform memory access programming model for high-performance computers. The Journal of Supercomputing **10** (1996) 169–189
18. Nieplocha, J., Foster, I.: Disk resident arrays: An array-oriented I/O library for out-of-core computations. In: Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press (1996) 196–204
19. Eklundh, J.O.: A fast computer method for matrix transposing. IEEE Trans. on Computers **20** (1972) 801–803
20. Kaushik, S.D., Huang, C.H., Johnson, R.W., Sadayappan, P., Johnson, J.R.: Efficient transposition algorithms for large matrices. In: Proceedings of the 1993 ACM/IEEE conference on Supercomputing, ACM Press (1993) 656–665
21. Suh, J., Prasanna, V.K.: An efficient algorithm for out-of-core matrix transposition. IEEE Trans. on Computers **51** (2002) 420–438
22. Krishnamoorthy, S., Baumgartner, G., Cociorva, D., Lam, C., Sadayappan, P.: On efficient out-of-core matrix transposition. Technical Report OSU-CIRSC-9/03-T52, School of Computer and Information Science, The Ohio State University (2003)
23. Krishnamoorthy, S., Baumgartner, G., Cociorva, D., Lam, C.C., Sadayappan, P.: Efficient parallel out-of-core matrix transposition. In: Proceedings of the International Conference on Cluster Computing, IEEE Computer Society Press (2003) to appear.
24. The Ohio Supercomputer Center. (`http://www.osc.edu`)