Layout Transformation Support for the Disk Resident Arrays Framework[*]

Sriram Krishnamoorthy[1] Gerald Baumgartner[2]
Chi-Chung Lam[1] Jarek Nieplocha[3]
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 70810, USA
gb@csc.lsu.edu

[3]Computational Sciences and Mathematics
Pacific Northwest National Laboratory, Richland, WA 99352, USA
jarek.nieplocha@pnl.gov

Abstract

The Global Arrays (GA) toolkit provides a shared-memory programming model in which data locality is explicitly managed by the programmer. It inter-operates with MPI and supports a variety of language bindings. The Disk Resident Arrays (DRA) model extends the GA programming model to secondary storage. GA and DRA together provide a convenient programming model that encourages locality-aware programming by the user, while presenting a high-level abstraction. High performance depends on the appropriate distribution of the data in the disk-resident arrays. In this paper, we discuss the addition of layout transformation support to DRA. The implementation of an efficient parallel layout transformation algorithm is done on top of existing GA/DRA functions; thus GA/DRA is itself used in implementing the enhanced DRA functionality. Experimental performance data is provided that demonstrates the effectiveness of the new layout transformation functionality.

[*]This work was supported in part through funding from the U.S. Department of Energy and the National Science Foundation (award 0121676)

1 Introduction

The Global Arrays (GA) library [19] [20] 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 the global address space and local memory. It is similar to distributed shared-memory models in providing an explicit acquire-release protocol. However, it also acknowledges that local data access is faster than remote data access. The GA model exposes to the programmer the hierarchical nature of memory in modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. Its focus is on array data types and blocked access patterns. The GA programming model inter-operates with the message passing model; in particular, the programmer can use full MPI functionality on both GA and non-GA data. The library can be used in C, C++, Fortran 77, Fortran 90 and Python programs. This functionality has proved useful in numerous significant computational chemistry applications, such as NWChem, that comprise over a million lines of code. [21].

The Disk Resident Arrays (DRA) model [18] extends the GA programming model to secondary storage (Fig. 1).

It provides a disk-based representation of arrays, and functions to transfer blocks of data between global arrays and disk resident arrays. The simple interface allows programmers to access data located on disk in terms of arrays rather than files. The benefits of GA (in particular, the absence of complex index calculations and the use of optimized array communication) are extended by DRA to programs that operate on arrays that are too large to fit in memory. 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 significantly improve performance while providing a programming model that is simpler than message passing.

Implementations of out-of-core computations can use disk resident arrays to implement user-controlled virtual memory, locating on disk the arrays that are too big to fit in aggregate physical memory, and transferring sections of these disk resident arrays into main memory for use in the computation. DRA functions can be used to read a section of a disk resident array into a global array and individual processors then use GA functions to transfer global array components to local memory. DRA has been designed to support collective transfers of large data blocks. No attempts are made to optimize performance for small (roughly less than 1MB) requests.

Performance studies [18, 4] have demonstrated that DRA is capable of achieving high performance in a variety of situations. However, high performance may depend on appropriate choices being made for two classes of key performance parameters:

- *Library configuration parameters.* These parameters include the number of I/O nodes used, the I/O buffer size on each I/O node, the communication mechanisms used, and the underlying file system used, together with the specific file system facilities used, such as the use of a special file mode, a special file layout, or a particular file system policy.

- *Disk array distribution parameters.* These parameters describe various aspects of the distribution of the array data on disks.

  Application developers can provide hints that guide, but do not direct, disk array distribution. DRA uses these hints (which can indicate, for example, the shape and size of a typical I/O request) as well as information about the internal I/O buffer size and file system characteristics to choose a disk array distribution.

The library configuration parameters are specified during the compilation of the library or during the creation of the disk resident array. The array layout is more dynamic and the appropriate parameters can vary at runtime. In this paper, we discuss the addition of layout transformation support to the GA/DRA framework.

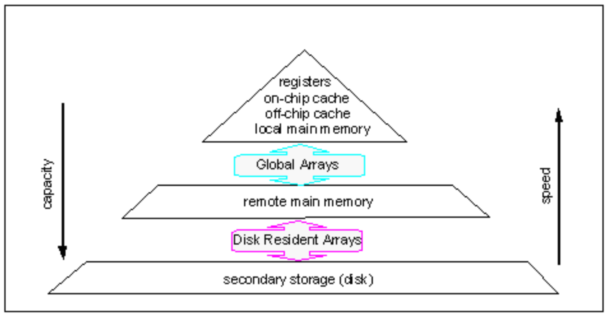

Figure 1: Role of GA and DRA in managing the memory hierarchy

We first describe an approach to efficient sequential data layout transformation for disk resident arrays. When we investigated the generation of a parallel implementation of the disk layout transformation algorithm, the needed inter-processor communication pattern was found to be quite complex. However, it was possible to use existing GA/DRA primitives to greatly simplify the programming of the parallel layout transformation algorithm. Henceforth, GA and DRA will be used to refer both to the libraries and the arrays used by them - the usage will be clear from the context.

The paper is organized as follows. Section 2 motivates the need for layout transformation. The layout transformation problem is defined in Section 3. The algorithm design is explained in Section 4. The considerations leading to the parallel implementation, including load balancing of the computation, are discussed in Section 5. Experimental results are presented in Section 6. Section 7 concludes the paper.

2 Motivation

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 [4], I/O libraries like PANDA [22, 9] and DRA [10] use a blocked representation for disk-resident multidimensional arrays instead of the dimension-ordered representation used typically for the representation of multidimensional arrays in main memory. Thus, the disk-resident 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-resident multidimensional arrays permits efficient access as long as the accessed regions mostly contain full bricks.

However, for some disk-resident multidimensional arrays, the access patterns of successive phases (for example, access patterns of the producer and consumer) are so different that no choice of brick shape allows for efficient access. An example is out-of-core 2-D FFT, where the array is accessed by columns in one phase and by rows in the other. The multi-dimensional Fast Fourier transform (FFT) [1, 2] can be implemented as a series of one-dimensional FFTs, one along each dimension. Another example illustrating very different access patterns is related to simulation data in three and four (including time) dimensions. The data produced by simulation can be too large to be handled in an in-memory fashion. For example, the 3-D simulation of the Richtmyer-Meshkov instability [17] generates $2048 \times 2048 \times 1920$ data elements per time step for 274 time steps. The production of data occurs one time-step after another. However, examination of the time evolution of a plane or 3-D block of data requires a very different access pattern than that by which the data was generated. When the data so produced is processed on a parallel system, the data might have to transformed into a different blocked form [12].

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)[24], to generate efficient parallel programs from high level expressions, for a class of computations expressible as tensor contractions [3, 6, 5, 7, 15, 16]. Often the tensors (essentially multi-dimensional arrays) are too large to fit in memory and must be disk-resident. The input tensors are typically generated by another chemistry package such as NWChem [21], with a layout quite different from that needed for efficient processing by the TCE-generated code.

3 The Layout Transformation Problem

Internally, the data in a DRA is stored in a blocked fashion. When a DRA is created, the shape/size of a typical request, referred to as the request 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: optimizing disk I/O bandwidth requires that the brick size be large enough to amortize the disk seek time; minimizing the wastage of disk I/O, due to the reading of additional data at the boundaries of DRA regions being read/written, requires a small brick size.

An application might have an access pattern that is very different from the organization of the disk-resident array. This can happen when an application uses the output of another program, or when different phases of the same program exhibit 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 a 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 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 2-D tile are not contiguous on disk, this could be extremely inefficient unless the tile size is very large. Eklundh [8] 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. [11] reduced the number of read operations and increased the read block size compared to Eklundh's algorithm. Sun and Prasanna [23] proposed an algorithm that minimized the total number of I/O operations, while potentially increasing the total volume of I/O. Krishnamoorthy et al. [14] 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 [13] .

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 [14].

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 $< 0 - \text{lcm}(s_1, t_1), \ldots, 0 - \text{lcm}(s_n, t_n) >$, where $\text{lcm}(s_i, t_i)$ is the least-common-multiple of s_i and t_i. 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 system). Hence the lcm-block can be arbitrarily large and might not fit in the 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 sufficient 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

$$M_i = \lceil (\max(s_i, t_i)/s_i) \rceil * s_i.$$

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.

$$\text{memCost} = \sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} M_i$$

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 M_{T_j} for $j > i$. Hence, the size of the buffer to store the unused data along a dimension T_i is bounded by

$$\begin{array}{ll} \text{bsize}_{T_i} & = \prod_{j=1}^{n} S_j \\ S_j & = \begin{cases} \text{lcm}(s_{T_j}, t_{T_j}) & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ M_{T_j} & \text{if } j > i \end{cases} \end{array}$$

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 formulas, 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:

$$\begin{array}{ll} \text{memCost} & = \sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} M_i \\ \text{bsize}_{T_i} & = \prod_{j=1}^{n} S_j \\ S_j & = \begin{cases} \text{templ}_{T_j} & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ M_{T_j} & \text{if } j > i \end{cases} \end{array}$$

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 $(U_1 * M_2 + \text{lcm}(s_1, t_1) * U_2)$ if dimension 1 is traversed first; otherwise, it is $(U_2 * M_1 + \text{lcm}(s_2, t_2) * U_1)$. 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 fit in memory.

The I/O cost is multiplicative along the dimensions. When a template does not align with the end of an lcm-block along a dimension, it is guaranteed to share read blocks with the next template along that dimension. Along a give dimension, the factor of re-read is thus proportional to the number of templates within an lcm-block. Each block is read and written at least once, with the factor of re-read increasing the I/O cost over the minimum possible. The I/O cost of re-blocking using a given template is thus given by

$$\begin{aligned}
\text{ioCost} &= 2 + \prod_{i=1}^{n} \text{overhead}_i \\
\text{lcm}_i &= \text{lcm}(s_i, t_i) \\
\text{overhead}_i &= \frac{s_i * \left(\frac{\text{lcm}_i - \text{templ}_i}{\text{templ}_i} \right)}{\text{lcm}_i}
\end{aligned}$$

When lcm_i is larger than the length of the array along dimension i, we replace the lcm_i by the array dimension. The size of the buffers to hold the unused data, determined by U_i, is an upper bound on the memory requirement and does not take into account the actual array dimensions. Even though the choice of a smaller template increases the I/O cost for the single-pass solution, 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 defined 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 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 first 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 block sizes for the multi-pass solution. Two candidate intermediate block sizes are considered. The first 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 refined 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. 4.

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. Let the array dimensions be much larger than the block sizes. The max-block is $< 32, 16 >$ and the unused data along each dimension is bounded by $< 4, 8 >$. The solution 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 $U_2 * M_1 + \text{lcm}(s_2, t_2) * U_1 + M_1 * M_2$=1344 elements, it is sufficient to hold all unused data when an lcm-block is processed. The second dimension is traversed first 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 first pass is $< 96, 12 >$, and that for the second pass is $< 60, 48 >$.

```
Input:  Source and target block sizes [s] and [t]
Output: Template size for single-pass solution
        if it exists.
Routines:
  memCost(templ) - Memory cost for processing the
                   given template
  DiskCost(templ) - I/O cost for processing the
                   given template
  MemExceeded(templ) - returns true if templ is
                      infeasible

 1. Initialize template to lcm-block
 2. Reduce template size along along all dimensions
      equally in units of write block size, until
      it is feasible.
 3. If no feasible template 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 steps 6 through 8
 6.   Among adjacent template sizes choose the
        one that has the maximum ratio of decrease
        in I/O cost to increase in memory cost.
 7.   From the chosen (infeasible) template,
        determine a feasible template that leads
        to the least increase in disk I/O cost.
 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.
```

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

5 Implementation

A pseudo-code for the sequential implementation, using file I/O, is shown in Fig. 5. The number of passes and the intermediate block sizes for each pass are first determined using the multi-pass solution algorithm in Fig. 4. The target DRA and an additional temporary file are used to store the intermediate data. The input file in the first pass is the one corresponding to the source DRA. The input and output files for each pass are chosen in such a way that the output file in the last pass is the file corresponding to the target DRA. The computation proceeds in a sequence of passes. The buffers to hold the unused data and the max-block are initialized. In each pass, the templates are processed one after another. The data corresponding to each template is traversed in units of max-block, in the predetermined order. In each step, a max-block is read into memory, complete write blocks are constructed and written to the output file. Reading a max-block from disk involves a sequence of I/O operations one for each brick in the max-block. If the max-block contains any unused data corresponding to the current template, it is stored in the unused buffers. If the max-block is only partially present in the current template (i.e. some of it corresponds to write blocks in another template), the data not relevant to the current template is discarded. Construction of the complete write blocks involves determining the regions of the read blocks to be combined, locating the regions from the buffers, and patching the data onto a temporary buffer. The data in the temporary buffer is then written to disk.

5.1 Implementation Choices

We needed a parallel implementation that can handle the different forms of disk arrays, in particular arrays on local disks and on a shared file system. Various alternatives in obtaining a parallel implementation of the algorithm were considered. The alternatives differed in the the level of abstraction utilized and the granularity of parallelism exploited.

At the coarsest level of parallelism, each template can be processed independently and hence can be assigned to a different process. Each process handles the next available template, which is determined at runtime. This

```
Input: Source and target block sizes [s] and [t],
       Template size [templ]
Output: Total memory cost [memCost]
        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
```

Figure 3: Algorithm to determine the memory cost for a given template size

```
Input:  Source and target block sizes [s] and [t],
Output: Sequence of intermediate block sizes,
        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 cost (b) of multi-pass solutions for
       re-blocking from s to B1 and B1 to t recursively.
 7.  Determine cost (c) of multi-pass solutions for
       re-blocking from s to B1, B1 to B2 and B2 to t
       recursively.
 8.  If no multi-pass solution exists
       return "no solution exists"
 9.  Choose the solution with minimum I/O cost from
       (a), (b) and (c) and determine the output
       appropriately.
```

Figure 4: Algorithm to determine a multi-pass solution

provides automatic load-balancing. Since the processes operate on disjoint sets of data, a low-level abstraction is required. GA/DRA requires a collective operation to perform I/O on the disk array, which is not suitable for template-level parallelism. The absence of one-sided access to the data on remote disks necessitates co-ordination of the computation amongst the different processes. This requires a load-balancing mechanism more sophisticated than the process-next-template scheme.

Another significant drawback of utilizing template-level parallelism is the orchestration of the computation amongst all the processes can utilize the global memory for processing. This can potentially reduce the number of passes by allowing a greater component of the transformation to be done in each pass. It is thus advantageous to have all the processes co-operate in transforming each template. Parallelism in the form of distributed ownership of the bricks by the I/O processes, those that perform I/O, is exploited. We redefine the max-block in each step to be large enough to construct complete write blocks so that all the available I/O processes are utilized.

The co-ordination amongst the processes can be achieved either using MPI and file I/O or using GA/DRA. Using MPI and file I/O provides greater flexibility and predictability to the computation. This could allow tuning the implementation to the specific environment. On the other hand, GA/DRA abstracts away the complexity in dealing with file offsets, packing and unpacking of data and message passing. That GA allows the use of message passing, in particular MPI method calls, on both GA and non-GA data in a GA/DRA program allows for incremental tuning of the implementation. A GA/DRA implementation can be further tuned using MPI and file I/O if such tuning can improve performance. When the tuning does not improve performance, a more maintainable code is available. The lessons learned from the tuning process can help in making further improvements to the GA/DRA model.

To illustrate the incremental tuning in the GA/DRA model, let us consider two possible optimizations. The GA/DRA implementation reads data from a disk array into a global array in memory. The data is processed in the global array and written back to the disk array. The data could be read into local memory, copied into a global array, and then written from the global array to the disk array. This could reduce the communication overhead by scheduling the communication in an intelligent manner. Alternatively, data can be read from the disk array into a global array and each process can copy the data into its local memory and write to the block it handles.

One attendant disadvantage of using GA/DRA for the operation is the increased disk space requirement. At any point in the computation, space is required for the source and target arrays of the current pass and the ultimate source and target arrays of the transformation. The input array is assumed to be read-only. The output array is unused until the last pass, and can potentially be utilized. But accessing the space allocated to the target array via the DRA induces a blocking that is usually incompatible with the blocking of the intermediate data. Operating at the file I/O level, one can bypass the blocked view and directly access the space. A simple extension to the GA/DRA framework, in which multiple disk arrays can use the same file (analogous to the union type in C) could be provided to allow different blocking views to the same data space on disk.

Another optimization is possible when operating at the file I/O level. Instead of operating on the entire array on a pass before proceeding to the next pass, each template can be processed through all the passes and written into

```
Input:  Source and target DRAs [d_s] and [d_t],
Output: d_t contains the data in d_s.

 1. Determine the multi-pass solution.
 2. Create a file as an intermediate. Use
      space in d_t as the other intermediate.
 3. foreach pass
 4.   Determine source and target files for this pass
        (so that the target in the last pass is d_t)
 5.   Allocate memory for unused buffers along
        each dimension, the buffer to contain
        the max-block, and a write block.
 6.   foreach template t
 7.     while max-blocks remain to be processed
 8.       Read next max-block into memory from source.
 9.       Construct complete write blocks from
            max-block and unused buffers.
10.       Write constructed complete write blocks to
            target.
11.       If max-block contains unused data
            corresponding to current template,
            store it into unused buffers.
12. Delete the temporary file
```

Figure 5: Pseudo-code for sequential implementation of the layout transformation algorithm

the final array before processing the next template. Thus additional space for only two templates is required (the space on the output array is not useful in this case as it is used during the transformation and might not have enough contiguous free space to store the intermediates). With the GA/DRA model, using this optimization would involve creating and using disk arrays the size of a template.

Thus the GA/DRA model supports incremental development and tuning by not precluding the use of lower-level programming models.

5.2 Parallel Implementation

The parallel implementation is similar to the sequential implementation, whose pseudo-code is shown in Fig. 5. The max-block and the unused buffers are global arrays. Each max-block is read in directly using the DRA interface. As illustrated in Fig. 6, the global array corresponding to the max-block is allocated additional space, i.e., dimension i of the global array is of size (M[i]+U[i]). The patching of the data from the unused buffers is done in the additional space allocated in the global array, such that the complete write blocks form an n-dimensional rectangular region. Thus, the construction of the complete write blocks is done in-place, eliminating the movement cost of the data in the intersection between the max-block and the complete write blocks, which do not need to go through the unused buffers. The complete write blocks are then written to disk. The GA/DRA abstraction greatly simplifies the implementation by allowing direct translation of most of the algorithm statements into invocation of DRA routines.

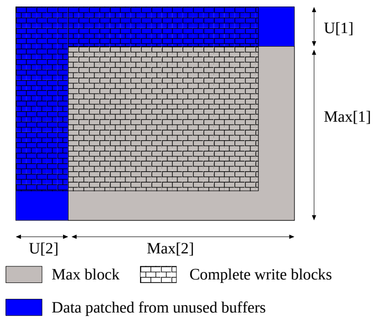

Figure 6: In-place construction of complete write blocks in the parallel implementation. The max-block, data from unused buffers, and the constructed write blocks are shown. Note that the regions overlap.

5.3 Load Balancing

In the parallel implementation, more than one process co-operates in performing the transformation. The basic unit of I/O, the max-block, is increased in size to allow all the processes to actively participate in the transformation. In the algorithm, the max block is defined as the set of read blocks that guarantees that at least one complete write block can be written out in each step. With P I/O processes, the max-block is defined to be the set of read blocks that guarantee that P complete write blocks can be written out in each step.

This set of read blocks can be chosen in a number of ways. To balance the load among the I/O processes, the P write blocks written out in each step should each be handled by a different I/O process. This allows for a balanced distribution of the I/O load, with all I/O processes actively performing I/O in each step.

A max-block that results in a load-balanced schedule can be determined by determining an n-dimensional rectangular region of P write blocks, each handled by a different I/O process. The read blocks that cover this region form the max-block.

A max-block that covers P consecutive write blocks along the fastest varying dimension form a simple load balanced schedule. However, such a scheme does not take advantage of the flexibility available in choosing the max-block so as to contribute to a global optimal solution. For example, the above scheme would not perform well if the target blocking had a very different orientation. A simple heuristic would be to choose a max-block that aligns with the target block.

In the algorithm design, the max-block was defined first, and other parameters such as memory cost were defined in terms of the max block. A choice in the max-block determination affects other costs and hence the optimal solution.

We discuss an algorithm to enumerate all possible load balanced max-blocks. Currently, the implementation chooses any load balanced max-block. Choosing the most appropriate load balanced max-block is not dealt with here.

The algorithm is shown in Fig. 7. It is based on the observation that the round-robin distribution enables partitioning of the entire array into load-balanced max-blocks of the same size and shape. If a partition results in the max-block at the origin of the array being load-balanced, all the max-blocks in the partition are guaranteed to be load balanced.

The algorithm can be viewed as a factorization of P to be assigned to different dimensions. The factor assigned to a dimension is the number of write blocks along that dimension to be covered by the max block.

The algorithm represents the array size indirectly using an offset vector. An offset vector is an n-dimensional vector, in which the i-th element represents the number of distance between two write blocks along that dimension in a linearization of the array into write blocks. For example, for a 10×10 array, blocked using 3×3 tiles, the offset vector is $(1, 4)$. The offset is always one along the fastest varying dimension. Along the next dimension, it is the number of blocks in all lower dimensions, which is four here. In the algorithm the offset is represented modulo the number of I/O processes. Thus, with two I/O processes the offset vector in the above example would be $(1, 0)$. In this form, the offset vector also represents the I/O processes that handle the blocks adjacent to the block at origin, along each dimension.

The offset along a dimension can be used to determine the number of different I/O processes that handle blocks, if one traverses the array along that dimension. In the above example, if the blocks are identified using a row-column pair, all blocks along the column $(0, *)$ are handled by I/O process zero. In fact, an offset of zero along a dimension implies that all blocks along that dimension are handled by the same I/O process.

A factor of more than one is assigned to a dimension only if the corresponding blocks chosen are handled by different I/O processes. All dimensions with non-zero offsets are chosen as candidates and are added to the set D. The routine *GenRecursively* is then invoked that recursively determines all feasible load balanced max-blocks. The routine recursively factorizes P and assigns factors to dimensions along the way.

```
Input:  Blocking and array dimensions.
        The number of I/O processes P.
Output: All feasible parallel max-blocks

GenMaxBlocks:
 1.  D = {}
 2.  Compute offsets into ''offset'' array
 3.  foreach dimension i
 4.    factor[i] = 1
 5.    if offset[i] > 0
 6.      D = D + {i}
 7.  GenRecursively(P, factor[], offset[], D)

GenRecursively(P, factor[], offset[], D):
 1. if P = 1
 2.   /**factor[] has a valid parallel
        max-block**/
 3.   print factor[]
 4.   return
 5. for each dimension i in D
 6.   if gcd(P, offset[i]) < P
 7.     f = P / gcd(P, offset[i])
 8.     if |D| > 1 or f = P
 9.       factor[i] = f
10.       GenRecursively(P/f,factor[],offset[],D-{i})
11.       factor[i] = 1
```

Figure 7: Algorithm to enumerate all parallel max-blocks

If P has been completely factorized and an invocation of the routine *GenRecursively* finds P to be one, the factorization in factor[] is a load balanced max-block. If not, the routine expands the search along each dimension, by attempting to assign a factor to each dimension and then backtracking to determine more possible solutions.

The possible assignment of a factor to a dimension is determined by $gcd(P, \text{offset}[i])$. The gcd determines the number of different I/O processes that handle blocks along that dimension. If the gcd is 1, it means all the I/O processes own blocks along that dimension. The number of I/O processes is correspondingly lower for higher gcd values. The number of I/O processes along a dimension i is given by $f = P/gcd(P, \text{offset}[i])$. Also, along a dimension, all I/O processes own a block before any I/O process owns a second block. Hence f can be assigned as a factor to that dimension. The algorithm uses these observations to enumerate the load balanced max-blocks.

6 Experimental Results

The implementation was evaluated on an Itanium 2 cluster at the Ohio Supercomputer Center (OSC) and the Mpp2 cluster at the Molecular Sciences Computing Facility in the Pacific Northwest National Laboratory (PNNL). The configuration of the systems is shown in Table 1. Initially the data is stored in row-major order on disk. We varied the data access pattern and measured three costs. The *skewed access cost* was first measured for each access pattern. The skewed access cost is the cost of accessing all the elements in the array using the specified access pattern, with the data stored in row-major layout. The skew refers to the *misalignment* between the access pattern and the layout of data on disk.

We then measured the cost of transforming the data layout to match the access pattern. This is referred to as the *conversion cost*. Finally, the cost of accessing the elements in the transformed array is measured. The access pattern is now fully aligned with the data layout and this cost if referred to as the the *aligned access cost*.

Table 2 shows the costs for a 32768×32768 array of doubles, on the Itanium 2 cluster. The costs were measured on one and two nodes, where one processor was used per node. The costs for a 65536×65536 array on four nodes is shown in Table 3. The results for 1, 2 and 4 processors (one per node) on the Mpp2 cluster for a 65536×65536 array is shown in Table 4. Each row in these tables represents a different access pattern being evaluated. The array is accessed in row-major order in units whose size/shape is specified. With P processors, each access corresponds to a read of P such blocks. The size of a block for all the access patterns was 1MB, the size internally chosen by DRA for a brick.

It can be observed that when the access pattern is closely aligned with the data layout on disk, the skewed access cost is higher than the aligned access cost, but not high enough to warrant layout transformation. If the transformed array needs to be accessed multiple times, then the layout transformation cost might be amortized by the lower aligned access cost. As the skew increases, the skewed access cost gets so high as to warrant a layout transformation even if the array is to be accessed just once after the transformation. As expected, the aligned access cost is similar for all block sizes. The layout transformation cost does not vary significantly with the transformation performed. This is because I/O is performed in units of an efficient block size determined by DRA. Thus the I/O cost does not vary between transformations unless the number of passes varies. We observed that all the transformations were performed in one pass.

7 Conclusions

In this paper we presented a new approach to efficiently transform the blocked layout of multidimensional disk-resident arrays. The number of passes in the layout transformation is determined based on the specific transformation, such that the overall I/O cost is minimized. The proposed approach was implemented as a new copy primitive within the DRA I/O library. Experimental results demonstrated the benefits of the layout transformation primitive.

	Itanium 2 cluster (OSC)	Mpp2 cluster (PNNL)
Processor	Dual Itanium 2 (900 MHz)	Dual Itanium 2 (1.5 GHz)
Memory	4GB	8GB
Local disk	80GB	430 GB
Interconnect	Myrinet 2000	Quadrics
Messaging Layer	GM	Elan-4

Table 1: Configuration of systems on which the implementation was evaluated.

The layout transformation implementation illustrated the convenience and incremental development enabled by the GA/DRA framework. The programming model allows for quick development using high-level constructs followed by tuning depending on the expected gains.

Acknowledgments

We would like to thank the Ohio Supercomputer Center (OSC) and the Molecular Sciences Computing Facility (MSCF) at the Pacific Northwest National Laboratory (PNNL) 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*, 28(3):280–284, 1980.

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

[3] 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.

[4] Y. Chen, I. Foster, J. Nieplocha, and W. Winslett. Optimizing collective I/O performance on parallel computers: A multisystem study. In *11th ACM Intl. Conf. on Supercomputing*, 1997.

[5] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and

Access Pattern		Access and transformation cost (seconds)					
		#procs = 1			#procs = 2		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access	Skewed access	Conv. cost	Aligned access
4	32768	176	359	172	97	241	90
8	16384	179	343	178	88	191	88
16	8192	182	345	175	91	173	91
32	4096	196	357	180	105	188	92
64	2048	249	368	181	129	190	93
128	1024	340	372	179	172	202	94
256	512	517	371	183	266	173	93
512	256	861	372	181	434	165	92
1024	128	1580	377	183	749	163	94
2048	64	2994	384	184	1393	167	93
4096	32	5760	373	180	2697	170	95

Table 2: Access and transformation cost (in seconds) for a 32768×32768 array stored in row-major order, on the Itanium 2 cluster.

Access Pattern		Access and transformation cost (seconds) (#procs=4)		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access
8	16384	207	733	212
16	8192	238	644	229
32	4096	300	743	230
64	2048	419	723	230
128	1024	650	623	230
256	512	1110	538	230
512	256	2030	466	230

Table 3: Access and transformation cost (in seconds) for a 65536×65536 array stored in row-major order, on the Itanium 2 cluster.

Access Pattern		Access and transformation cost (seconds)								
		#procs = 1			#procs = 2			#procs = 4		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access	Skewed access	Conv. cost	Aligned access	Skewed access	Conv. cost	Aligned access
8	16384	155	370	221	137	249	71	54	186	49
16	8192	209	420	229	177	224	72	83	138	63
32	4096	298	428	292	321	241	69	95	133	54
64	2048	436	423	298	521	265	71	129	116	58
128	1024	734	469	304	973	287	68	194	128	62
256	512	1315	453	307	1938	252	71	322	144	54
512	256	2473	446	316	3648	276	64	579	149	56

Table 4: Access and transformation cost (in seconds) for a 65536×65536 array stored in row-major order, on the Mpp2 cluster.

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.

[6] 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.

[7] 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.

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

[9] The Panda Project: Data Management for High-Performance Scientific Computation. http://drl.cs.uiuc.edu/panda/.

[10] I. Foster and J. Nieplocha. Disk Resident Arrays: An array-oriented I/O library for out-of-core computations. In Rajkumar Buyya, Hai Jin, and Toni Cortes, editors, *Disk Arrays and Parallel I/O: Theory and Practice*. IEEE Computer Society Press, 2001.

[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] R. Kazhiyur-Mannar, R. Wenger, R. Crawfis, and T. K. Dey. Adaptive resolution isosurface construction in three and four dimensions. Technical Report OSU-CISRC-7/03–TR38, Dept. of Computer and Information Science, The Ohio State University, July 2003.

[13] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan. Efficient parallel out-of-core matrix transposition. In *Proceedings of the International Conference on Cluster Computing*. IEEE Computer Society Press, 2003.

[14] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan. On efficient out-of-core matrix transposition. Technical Report OSU-CISRC-9/03-T52, Dept. of Computer and Information Science, The Ohio State University, Sept 2003.

[15] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Data locality optimization for synthesis of efficient out-of-core algoritms. In *Proc. of the Intl. Conf. on High Performance Computing*, 2003.

[16] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, V. Choppella, and P. Sadayappan. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. In *Proc. of 18th International Parallel & Distributed Processing Symposium (IPDPS)*, 2004.

[17] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70. ACM Press, 1999.

[18] J. Nieplocha and I. Foster. 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*, pages 196–204. IEEE Computer Society Press, 1996.

[19] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable programming model for distributed memory computers. In *Supercomputing*, pages 340–349, 1994.

[20] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

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

[22] Kent E. Seamons and Marianne Winslett. Multidimensional array I/O in Panda 1.0. *The Journal of Supercomputing*, 10(2):191–211, 1996.

[23] 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.

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