An Optimal Index Reshuffle Algorithm for Multidimensional Arrays and its Applications for Parallel Architectures

Chris H.Q. Ding
NERSC Division, Lawrence Berkeley National Laboratory
University of California, Berkeley, CA 94720. chqding@lbl.gov

April, 1999

Abstract

Reshuffling elements of a multidimensional array according to an index operation traditionally requires an auxiliary buffer of the same size as the original array. Here we describe a new in-place algorithm using vacancy tracking cycles with minimum memory access, which eliminates the buffer array and the related copy-back, therefore speeding up the reshuffle significantly for large arrays. The algorithm can be parallelized using a multi-thread approach on shared-memory multi-processor computers. On distributed-memory multi-processor computers, index reshuffle of distributed multidimensional arrays amounts to a remapping of processor domains and is carried out using the in-place local algorithm combined with a global exchange algorithm. Implementation and test results on CRAY T3E and IBM SP indicate the effectiveness of the algorithm.

Keywords: multidimensional arrays, index reshuffle, vacancy tracking cycles, global exchange, dynamical remapping.

1 Introduction

Dynamically remapping problem domains on distributed-memory multi-processor architectures are encountered frequently in many scientific and engineering applications. Instead of fixing the problem decomposition during entire computation, dynamically remapping the problem domains to suit the specific needs at different stages of the computation can often simplify computational tasks significantly, saving coding efforts and reducing total problem solution time.

An example is shown in Figure 1. The 3D fields of an atmosphere (or ocean) model are mapped onto 8 processors, with horizontal dimensions split among the processors. In spectral transform based models, such as the CCM atmospheric model[1, 2] and the shallow water equation[3], one often needs to dynamically remap between the *height-local* domain decomposition and the *longitude-local* decomposition for tasks of distinct nature. In grid-based atmosphere and ocean models, similar remappings are needed for polar filtering[4] and for data input/output[5].

An important aspect of the multidimensional array remapping problem is the memory usage. This becomes a pressing issue because increasingly larger problems are being solved on today's highly parallel systems with ever increasing computing power. In climate simulations, doubling the model resolution typically requires a factor of 8 increase of array sizes. Traditional implementation of 3D array remapping requires an auxiliary buffer array of the same size as the original data array, to hold the temporary data during the remapping. [A simple example is exchange the two indexes of a 2D array $A(N_1, N_2)$.] This puts a severe limitation on such memory-bound problems.

In this paper, we first introduce a vacancy tracking method for multidimensional array *index reshuffle* in local memory (i.e., reshuffling array elements in a way that corresponds to certain index operation, see section 2). The method reshuffles elements within the original array according to vacancy tracking cycles, and eliminates the need for the auxiliary array. It therefore (i) reduces the memory requirements by half; (ii) eliminates the copy-back process in traditional methods, i.e., copying the reshuffled data from the auxiliary array back to the original array, thus speeds up the index reshuffle substantially. In fact, the vacancy tracking algorithm reduces the total number of memory access to the absolute minimum possible — it is an optimal algorithm. (Section 2).

The vacancy tracking algorithm for multidimensional array can be parallelized on shared-memory or symmetric multi-processor (SMP) architectures with a multi-thread approach, making use of the independence of vacancy tracking cycles. (Section 3).

For distributed memory multi-processor (DMP) architectures, the local vacancy tracking algorithm is combined with a global all-to-all exchange method, leading to a global in-place

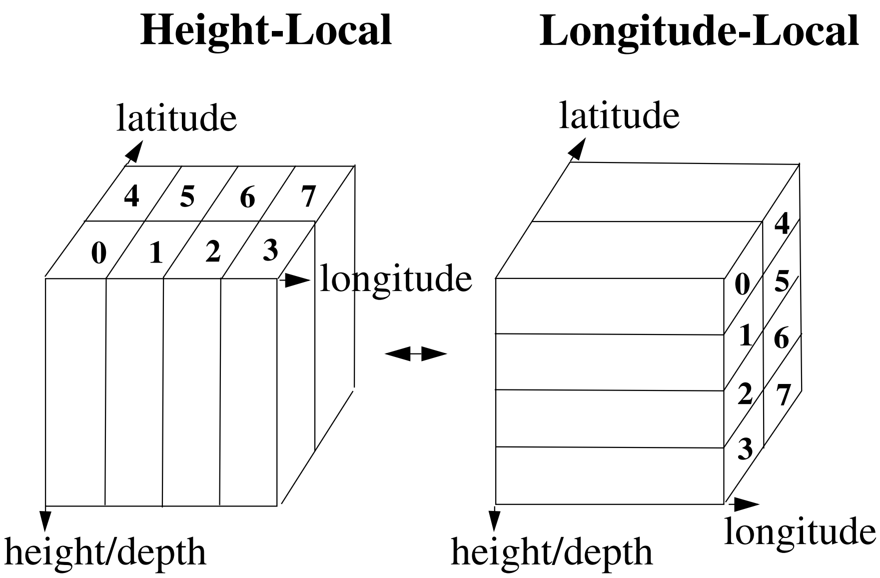

Figure 1: Dynamical remapping of 3D atmosphere/ocean models. In height-local domain decomposition, all data points along vertical dimension are on the same processor. In longitude-local decomposition, all points along a longitude are on the same processor.

multi-dimensional array remapping algorithm. This provides a solution to the memory usage issue for dynamical remapping. Implementation details are discussed (section 4).

The method and specific algorithms represented here are implemented on distributed memory computers Cray/SGI T3E and IBM SP with MPI message passing library. The performance and analysis are presented for sequential cases (single processor) in section 2 and for multi-processor cases in section 4. Issues with cache usage and scaling to large number of processors are also discussed. Some concluding remarks are made in section 5.

To our knowledge, this paper is the first study of an in-place index reshuffle algorithm for multi-dimensional arrays with arbitrary dimension sizes. For one-dimensional arrays when the size is a power of 2, a study of index permutations has been discussed by Fraser [6]. For more index reshuffle related studies, see Ref.[7] and references there.

2 Index reshuffle using a vacancy tracking algorithm

In many computational problems, we are interested in re-arranging the order of a multi-dimensional array, i.e., reshuffling array elements, in such a way that corresponds to array index operations, such as exchanging two indices. Here we focus on index reshuffles on 3-dimensional (3D) arrays, but the algorithm can be easily extended to higher dimensions.

Given a 3D array A with dimensions N_1, N_2, N_3 indexed as $A(k_1, k_2, k_3)$, we consider the reshuffle with index exchange between k_2 and k_3. Using an auxiliary buffer array B of equal size, the index exchange between k_2 and k_3 can be easily carried out as follows:

```
do k3 = 1, N3
do k2 = 1, N2
do k1 = 1, N1
   B(k1,k3,k2) = A(k1,k2,k3)                                    (C.1)
end do
end do
end do
```

We denote this index reshuffle of A and storing the results in B as:

$$B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]. \tag{1}$$

In many situations, B is copied back to memory locations of A (denoted as $A \Leftarrow B$), and memory for B is freed. We will call this traditional method the two-array reshuffle method, because of the need of the auxiliary array B. Combining the $B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]$ reshuffle phase and the copy-back $A \Leftarrow B$ phase, the net effect of the two-array reshuffle method can be written symbolically as

$$A'[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3] \tag{2}$$

Here A' indicates that reshuffled results are stored at the same location as the original array A. Intuitively one may interpret 3D array $A'[k_1, k_3, k_2]$ as having the 1st array index k_1 as the fastest running index in storage, the 3rd array index k_3 as 2nd fastest running index in storage, and the 2nd index k_2 as the slowest running index in storage (here we follow Fortran storage convention).

Clearly, index reshuffles corresponding to exchanging k_1 and k_3 indexes also occur frequently. This can be denoted as,

$$A'[k_3, k_2, k_1] \Leftarrow A[k_1, k_2, k_3] \tag{3}$$

The code segment implementing this reshuffle is very similar to (C.1).

Reshuffles with three-index exchanges also occur. They can be considered as successive reshuffles with two-index exchanges. For example, a reshuffle with left-circular-shift of all three indexes,

$$A'[k_2, k_3, k_1] \Leftarrow A[k_1, k_2, k_3] \tag{4}$$

is the combined effect of two-index reshuffles of Eqs.(2, 3). In actual implementations, three-index reshuffles are carried in one step, just as two-index reshuffles are.

In many situations, the array is very large and there is not enough memory to store the auxiliary array B to perform the above mentioned index exchanges. In these situations, we need an in-place method, i.e., the index reshuffle must be carried out using A's memory space only. The main contribution of this paper is to introduce a new in-place algorithm for this reshuffle. Furthermore, with the elimination of the auxiliary array, the copy-back phase is eliminated too; thus the in-place algorithm speeds up the reshuffle by nearly a factor of 2 (see performance analysis in section 2.4).

As is well-known, in the special case of two-index exchanges and when the dimensions of the two indices are the same, a simple in-place exchange algorithm can be used which is essentially the transposition of a square matrix; we need only a temporary buffer to hold a single array element. However, in most applications dimension sizes are not equal. Furthermore, in parallel remapping, even if the global array dimensions have same sizes, local subdomain arrays have different dimension sizes due to the decomposition. An in-place algorithm for arbitrary sizes and dimensions is necessary.

2.1 Vacancy tracking cycles

The key idea of this in-place algorithm is to view the index reshuffle as a mapping from original memory locations to new target memory locations and to move elements from old locations to new locations in a specific memory-saving order. In doing so, closed loops of vacancy tracking cycles are generated.

We start with a very simple example of a two-index exchange. Consider a 2D array A with dimension sizes 3 and 2, denoted as A(3,2) as in Fortran. The six elements of A(3,2) are labeled as $A_0, A_1, A_2, A_3, A_4, A_5$, as they are stored in the six consecutive memory locations $L_0, L_1, L_2, L_3, L_4, L_5$, in the original array (indicated as the left-most diagram in Figure 2). The task of index reshuffle here is to reshuffle elements to the order indicated as the right-most diagram in Figure 2. Each element has a starting location before the reshuffle and a target location after the reshuffle. Our task is to move them from starting locations to target locations in minimal steps with only one temporary buffer `tmp` space.

Consider A_1. A_1 should be in L_2 after the reshuffle. We move A_1 to the buffer `tmp` and L_1 is a vacancy now. Who should go to L_1 after the reshuffle? A_3 does, we move A_3 to fill the vacancy L_1. L_3 is the vacancy now. Who should go to L_3? A_4 does, we move A_4 to L_3. Now L_4 is the vacancy. After moving A_2 to L_4 to fill the vacancy, L_2 is the vacancy. A_1 should go to L_2, thus we come back to A_1 where we started. A_1 has been moved to `tmp` at the beginning of the cycle, and we move A_1 from `tmp` to L_2. The cycle is closed. We see that this vacancy tracking algorithm naturally leads to the following closed length-4 vacancy tracking cycle:

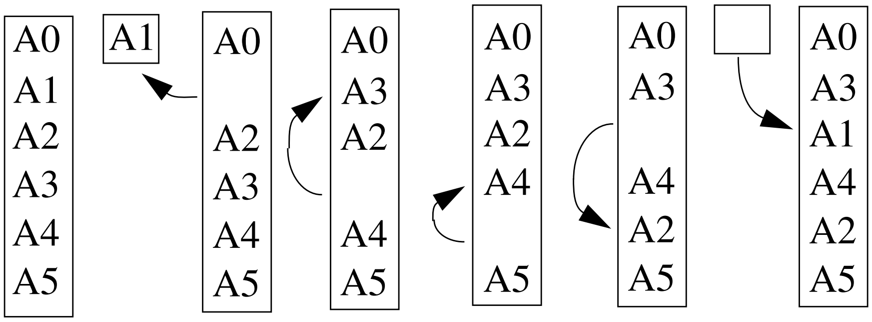

Figure 2: Permutations of elements in 2D array A(3,2). These moves follow the 1 - 3 - 4 - 2 - 1 vacancy tracking cycle.

```
1 - 3 - 4 - 2 - 1
```

These moves are schematically indicated in Figure 2. It is important to note that no additional memory space is needed to carry out this length-4 cycle. In this algorithm, element A_0 is never touched, and we denoted this fact as length-1 cycle 0 - 0. Similarly, A_5 is never touched; we have cycle 5 - 5. Including length-1 cycles, we say that every element belongs to a vacancy tracking cycle. One can also see that the number of memory accesses is the minimum possible: 4 elements are moved into 4 new locations in 4 reads and 4 writes (assuming `tmp` is a register).

Using the same method, one can also complete the index exchange of 2D array A(4,2) by following the two length-4 cycles:

```
1 - 4 - 2 - 1
3 - 5 - 6 - 3
```

as can be verified by visual inspection. There are two length-1 cycles: 0 - 0 and 7 - 7. For 2D arrays with equal dimension sizes, this algorithm will generate cycles with length-1 (for diagonal elements) and length-2 (for non-diagonal elements) only, reducing to the traditional pair-wise exchange algorithm mentioned earlier. (From now on, we will neglect all length-1 cycles to simplify the discussion, since elements involved in these cycles are never touched in our algorithm.)

Higher dimensional arrays can be dealt in the same way. Consider a 3D array A(3,2,2) with dimensions 3, 2, 2. An in-place index-exchange between 1st and 3rd indices, denoted as Eq.3, can be achieved by the following vacancy tracking cycles:

```
1 - 6 - 4 - 1
2 - 3 - 9 - 8 - 2
```

```
5 - 7 - 10 - 5
```

For the same 3D array, the in-place three-index left-circular-shift reshuffle as denoted in Eq.4, can be achieved using the following two cycles:

```
1 - 3 - 9 - 5 - 4 - 1
2 - 6 - 7 - 10 - 8 - 2
```

For a 3D array A(4,3,2) with 24 elements, the three-index left-circular-shift reshuffle can be achieved by the following two cycles:

```
1 - 4 - 16 - 18 - 3 - 12 - 2 - 8 - 9 - 13 - 6 - 1
5 - 20 - 11 - 21 - 15 - 14 - 10 - 17 - 22 - 19 - 7 - 5
```

2.2 Implementation

Clearly, these vacancy tracking cycles must be automatically generated. Here we give an implementation for the in-place index reshuffle of a multi-dimensional array. For presentation purposes, the algorithm is written as if it is dealing with a 2D array $A(N_1, N_2)$. For 3D or higher dimensional arrays, the necessary modifications are also illustrated. The following psuedo-Fortran code segment outlines the procedure.

```
! For 2D array A, viewed as A(N1,N2) at input and as A(N2,N1) at output.
! Starting with (i1,i2), find vacancy tracking cycle
        ioffset_start = index_to_offset(N1,N2,i1,i2)
        ioffset_next = -1
        tmp = A(ioffset_start)
        ioffset = ioffset_start
        do while( ioffset_next .NOT_EQUAL. ioffset_start)             (C.2)
           call offset_to_index(ioffset,N2,N1,j1,j2)  ! N1,N2 exchanged
           ioffset_next = index_to_offset(N1,N2,j2,j1)! j1,j2 exchanged
           if(ioffset .NOT_EQUAL. ioffset_next) then
               A(ioffset) = A(ioffset_next)
               ioffset = ioffset_next
           end if
        end_do_while
        A(ioffset_next) = tmp
```

Here the function `index_to_offset(N1,N2,i1,i2)` returns the offset from the base of the array in memory space, given *input* indices `(i1, i2)` and sizes N_1, N_2. Subroutine `offset_to_index(ioffset,N1,N2,j1,j2)` computes indices `(j1,j2)` given the offset and array dimension sizes. (Here `ioffset`, and indices `i,j` are all zero-based). Note also that array `A` is indexed by the offset, independent of whether is declared as `A(N1,N2)` at input or declared as `A(N2,N1)` at output — `A` is best viewed as a pointer. If explicit dimension declaration is needed, one can declare two arrays, `A(N1,N2)` and `B(N2,N1)` and use equivalence or same common block in Fortran 77, reshape in Fortran 90, or union in C to indicate they occupy the same memory location.

Here is a brief description of the code segment. The vacated location, index by `(i1,i2)` in the original $A(N_1, N_2)$ array, has the `ioffset` as returned by `index_to_offset(N1,N2,i1,i2)`. The same location identified by `ioffset` is then interpreted as a target location in the reshuffled array $A'(N_2, N_1)$ with index `(j1,j2)`, as computed in `offset_to_index(ioffset,N2,N1,j1,j2)` [note the exchange of N1 with N2 here]. Index reshuffle requires that the element with indices `(j1,j2)` in the original array be moved to this location. That element has an offset, `ioffset_next`, and is calculated by `index_to_offset(N1,N2,j1,j2)`. Once the content at location `ioffset_next` is moved to location `ioffset`, the vacated location is now indicated by `ioffset = ioffset_next`. The process is repeated until the vacated location come back to where it started (`ioffset_start`), and the content in `tmp` is moved back to the last vacated location.

The index routines can be easily implemented given an array storage scheme. For the conventional Fortran linear storage scheme, for 2D arrays,

```
function index_to_offset(N1,N2,i1,i2)
     return (i1 + i2*N1).
subroutine offset_to_index(ioffset,N1,N2,j1,j2)
     return {j2 = ioffset/N1, j1 = MOD(ioffset,N1)}
```

For 3D arrays, say $A(N_1, N_2, N_3)$, the above code segment remains unchanged, except `(i1,i2)` are replaced by `(i1,i2,i3)`, and (N_1, N_2) are replaced by (N_1, N_2, N_3). The indexing conversion routines are implemented as

```
function index_to_offset(N1,N2,N3,i1,i2,i3)
    return (i1 + i2*N1 + i3*N2*N1).
subroutine offset_to_index(ioffset,N1,N2,N3,j1,j2,j3)
    return {j3 = ioffset/(N1*N2),
            j2 = (ioffset-j3*N1*N2)/N1,
            j1 = MOD(ioffset,N1)}
```

For a reshuffle corresponding to the 1st and 3rd index exchange, $A'[k_3, k_2, k_1] \Leftarrow A[k_1, k_2, k_3]$, the code lines containing `offset_to_index()` and `index_to_offset()` should be replaced by

```
call offset_to_index(ioffset,N3,N2,N1,j1,j2,j3)  ! N1,N3 exchanged
ioffset_next = index_to_offset(N1,N2,N3,j3,j2,j1)! j1,j3 exchanged
```

For three-index left-circular-shift reshuffle, $A'[k_2, k_3, k_1] \Leftarrow A[k_1, k_2, k_3]$, they should be replaced by

```
call offset_to_index(ioffset,N2,N3,N1,j1,j2,j3)  ! N1,N2,N3 left-shift
ioffset_next = index_to_offset(N1,N2,N3,j3,j1,j2)! j1,j2,j3 right-shift
```

Note here that the indices `(j1,j2,j3)` are right-shifted, not left-shifted, to find the element in the original array.

One can easily implement this code and use it to generate vacancy tracking cycles for various cases, including those listed in section 2. When this algorithm is applied to 2D arrays with $N_1 = N_2$, it will produce exactly those length-2 cycles as in the standard square matrix transposition, except that the diagonal elements (length-1 cycles) are not touched as they should not be. When applied to 3D arrays with $N_1 = N_2 = N_3$ for a three-index shift reshuffle, it will generate only length-3 cycles as expected. This algorithm is far more efficient than the intuitive method of two consecutive two-index exchanges as discussed earlier.

The correctness of this algorithm depends on that the `do while` loop always lead to closed vacancy tracking cycles, and that the cycles are non-overlapping, i.e., every array element is moved and each moves only once. This can be easily proved based on the following two facts: (a) the uniqueness of the index-offset relationship (results of `index_to_offset()` and `offset_to_index()` are unique); and (b) the number of array elements is finite; so does the number of distinct offsets. As a cycle proceeds according to the `do while` loop, `ioffset` can only touch offset values it had not touched before, until it reaches the starting value again and completes the cycle. Two different cycles can not partially overlap, i.e., share one or more common offset values, which would contradict the uniqueness of the index-offset relationship; they are either completely non-overlapping or entirely identical.

There are a number of ways to use the basic cycle generation algorithm of C.2. One can generate vacancy tracking cycles and move the data items on the fly as in C.2. In the simplest implementation, a bit array of size of total elements is needed to indicate whether the memory location is *touched* or not.

A better implementation would be to pre-calculate the cycles' information before actually moving data items. Starting offsets and cycle lengths for each cycle are stored in a cycle table with the number of table entries equal to the total number of cycles. An outer do loop over the code segment C.2 then does one single actual cycle after another to move the data items in an orderly way, using the cycle information stored in the cycle table.

This saves the small overhead time over the repeated dynamical remappings during the course of a computation task, and has a number of other advantages: (a) diagnostics — we have a record of exact reshuffle moves; (b) facilitates inverse reshuffle or remapping; (c) helps parallelization on SMP architectures; (d) remove requirement for the bit array when generating the cycle information by using a few integers to keep track of the next *untouched* locations. (e) further speedup data movements by also storing the cycle offsets.

Inverse indexing reshuffle refers to restoring the 3D array from a reshuffled index order back to the original index order. It can be handled in the identical way as a forward reshuffle. Suppose we had shuffled $A(N_1, N_2, N_3)$ to $A'(N_3, N_2, N_1)$ with two-index exchange. To reshuffle back, one can invoke the identical reshuffle codes on A', but with dimensions (N_3, N_2, N_1), instead of dimensions (N_1, N_2, N_3) in the forward reshuffle. Notice that in the inverse reshuffle, the same vacancy tracking cycles will be generated as in the forward reshuffle, except that the cycle traversal direction is reversed. Therefore, if the cycles in the forward shuffle are pre-calculated and stored, the inverse reshuffle can make use of it, instead of generating them anew.

We note that the larger the element size is, the less the number of moves for a fixed number of total array size, and the more effective the method. In the climate modeling application, all elements in a dimension [k1 dimension in code segment (C.1)] are collapsed into one element and are moved together in a single step. Thus the number of elements that appear in vacancy tracking cycles are $N_2 * N_3$, much less than the size of the total data array $N_1 * N_2 * N_3$. Using short integers (4-byte), the size of the table that stores the cycle information therefore is about $4 * N_2 * N_3$ bytes. Of course, if memory is extremely tight, one can generate the cycles on the fly, eliminate this storage completely. The bit array of $N_2 * N_3/8$ bytes is required in our present implementation, which could be eliminated with a more sophisticated method to account for the untouched locations.

The new vacancy tracking method introduced here for multi-dimensional array index reshuffle is quite generic. It can be usefully applied when storage capacity and access are major concerns, such as re-arranging large relational tables in a relational database system. There, the table re-arrangement can be done in-place, with `tmp` being a memory cache.

2.3 Memory access

The in-place array reshuffle algorithm reduces the total number of memory accesses in two important ways. First, the copy-back phase in the traditional reshuffle method is eliminated, therefore reducing memory access by half.

Second, in moving elements from starting locations to target locations, the vacancy tracking cycle algorithm uses the minimum number of memory accesses possible. In the example of reshuffle of A(3,2) (cf Figure 1), a total of 4 elements are moved to new locations with 4 reads and 4 writes (assuming `tmp` is on register or cache). For these two reasons, our in-place algorithm is an optimal one regarding memory access.

Equivalently, we can count memory access in the following way. The length-4 cycle involves 3 memory-to-memory copies, one memory-to-`tmp` copy and one `tmp`-to-memory copy. If we assume the `tmp` storage is a register or cache, the access time will be negligible compared to access to DRAM. For counting memory access purposes, we can combine the memory-to-`tmp` copy and `tmp`-to-memory copy as one memory-to-memory copy. Thus the length-4 cycle requires a total of 4 memory-to-memory copies.

In contrast, the two-array $B \Leftarrow A$ reshuffle phase (C.1) requires 6 memory-to-memory copies, one for each element. In this case, our in-place algorithm saves 2 memory-to-memory copies, because elements A_0 and A_5 are already in the right place and do not need to be copied. This observation holds for all cases. For example, a 5x3 array has two length-6 cycles, and requires 12 memory-to-memory copies. There are 3 length-1 cycles, i.e., 3 elements are already in the right places. The two-array method would require 15 memory-to-memory copies.

On cache-based processor architectures, the memory access pattern is as important as the number of memory access. It may appear that our vacancy tracking algorithm has an access pattern more random than the regular access pattern in the two-array method. However, two points should be noted: (a) for the type of memory-bound problems that this algorithm is targeted for, the number of bytes in each move is often large; as long as this is larger than a cache-line size, which is typically about 64 - 128 bytes long (64-byte long on CRAY T3E and 128-byte long on IBM SP), memory access in vacancy tracking algorithm is not irregular at scales relevant to cache performance; (b) the write access in the two-array $B \Leftarrow A$ reshuffle phase [see code segment (C.1)] has a large stride. Thus its write memory access efficiency is about the same as the vacancy tracking algorithm. None of them can take advantage of the efficient unit-stride writes provided in some processor architectures if the element size is small. [On T3E, the write-buffer is 32-byte long. Element size equal or larger than this will have efficient write access.]

2.4 Performance.

The in-place index reshuffle algorithm is implemented using Fortran 90. We report timing results on two widely used high performance computer systems, the Cray T3E and the

IBM SP. The tests are for reshuffle index k_2, k_3 in 3D array $A(N_1, N_2, N_3)$. The vacancy tracking cycles are pre-calculated and stored to reduce the overhead for both forward and inverse remapping, as explained above. Since the conventional two-array reshuffle method as outlined in code segment (C.1) is widely used in practice, its performance is also measured and serves as the baseline to judge the new in-place algorithm.

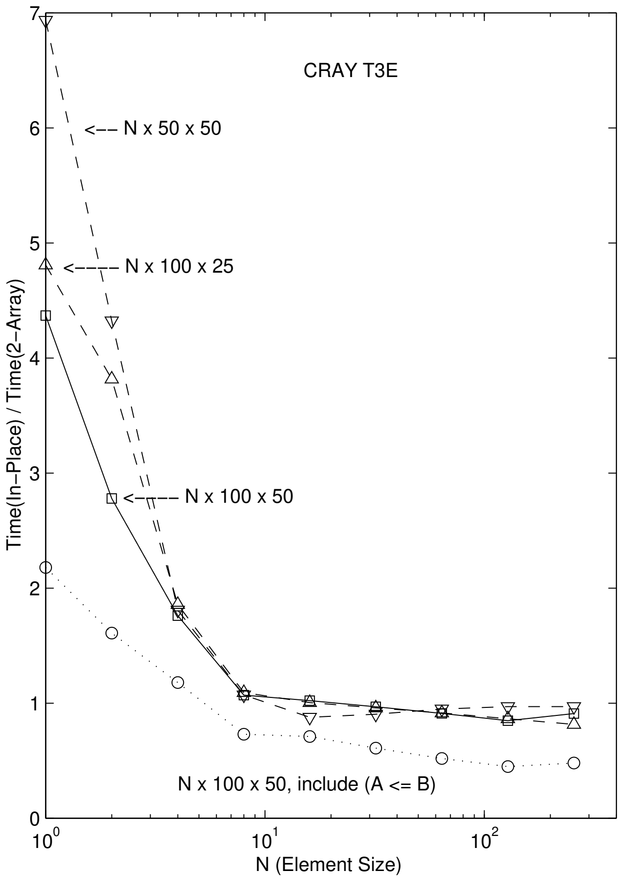

Figure 3: Timing for local 3D array index reshuffle on CRAY T3E. Plotted are the ratio of timings between the in-place algorithm and the two-array method (including $B \Leftarrow A$ reshuffle phase only for 3 upper curves, and including both $B \Leftarrow A$ and $A \Leftarrow B$ phases for the bottom curve).

In Figure 3, we plotted the timing of reshuffling 3D double precision arrays with sizes $N \times 100 \times 50$, $N \times 100 \times 25$, and $N \times 50 \times 50$, as a function of N on a single processor of CRAY T3E. The ratios between the timing of the in-place algorithm and that of the two-array method are shown.

For $N = 1$, the in-place algorithm is about 6 times slower than the two-array method. This is partly due to the overhead for the extra loop in implementing the vacancy tracking cycles, but probably mainly due to the inefficient memory access. This overhead quickly becomes negligible as the element size N increase to about 8. On T3E, memory access is through cache lines of 64-byte long, which is 8 double precision numbers. This indicates that the vacancy tracking algorithm has the same efficient memory access as the two-array method does when the element size reaches the cache-line size. As N further increases, the in-place algorithm typically outperform the two-array method $B \Leftarrow A$ reshuffle phase for about 5-10% because of reduced memory access.

The bottom curve is the timing when both $B \Leftarrow A$ and $A \Leftarrow B$ phases are included in the two-array reshuffle method. It behaves similarly as N increases and reaches about 0.5 for large N.

Array size $N \times 50 \times 50$ seems to perform the least efficiently for in-place algorithm; $N \times 100 \times 25$ is better and $N \times 100 \times 50$ is the best. This is because the number of vacancy tracking cycles varies significantly: they are 1225, 42, and 14 respectively. The less cycles, the less overhead in the algorithm.

In Figure 4, we plotted the similar timing of array $N \times 100 \times 50$ on a single processor of IBM SP. The curves are similar to those on T3E, with a few clear differences. At small N, the overhead of the in-place algorithm is about 50%, much smaller than 400-500% on T3E. On the SP Power 3 processor, cache lines are 128-byte long, so the in-place algorithm reaches the break even points at about $N = 16$ for REAL*8. For REAL*4, the overhead is always bigger than that of the REAL*8 data type. (On T3E we did not test REAL*4 because T3E has REAL*8 only.)

A note on compiler options. On SP, the timing is very sensitive to optimization levels. The two-array codes C.1 performs poorly at the default O2 level, and doubles its performance when compiled at the highest O5 level. The in-place algorithm C.2 performs relatively well at O2, and gains much less performance at O5. All timings here are done at O5, otherwise the timing is much more favorable to the in-place algorithm. On CRAY T3E, both codes perform well at the default O2 and the highest O3 levels; we use O3. Another subtle timing issue is the $A \Leftarrow B$ copyback phase. Implemented in explicit indexing or array syntax (A=B) in F90, the timing is typically the same as in the $B \Leftarrow A$ reshuffle phase. To achieve higher performance, we implemented it using the BLAS DCOPY routine; with this, the copyback is typically 30% faster than the $B \Leftarrow A$ reshuffle phase.

3 A Multi-thread Parallelization for SMP Architectures

The vacancy tracking algorithm can be easily parallelized using a multi-threaded approach in a shared-memory multi-processor environment to speed up data reshuffles, e.g., to re-

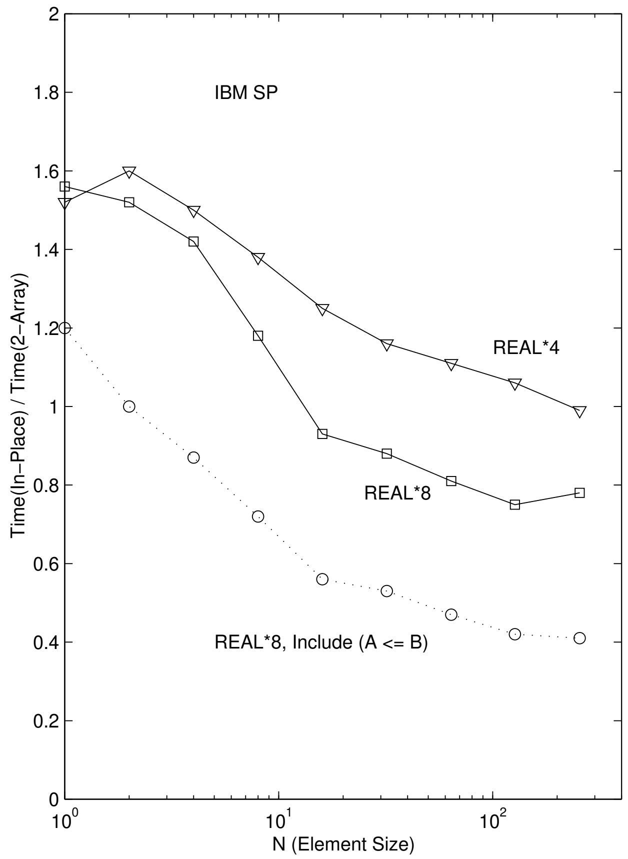

Figure 4: Timing for a local 3D array index reshuffle on IBM SP. Plotted are the ratio of timing between the in-place algorithm and the two-array method (including $B \Leftarrow A$ reshuffle phase only in two upper curves; and including both $B \Leftarrow A$ and $A \Leftarrow B$ phases in the bottom curve). The array size is $N \times 100 \times 50$, with both REAL*4 and REAL*8 data types.

organize a database on a SMP server. As discussed in detail in the correctness proof, the vacancy tracking cycles are non-overlapping. If we assign a thread to each vacancy tracking cycle, they can proceed *independently* and *simultaneously*.

The cycle generation code (C.2) runs first in the initialization phase before the actual data reshuffle, to determine the number of independent vacancy tracking cycles and associated cycle lengths and starting locations. These cycle information can be stored in a table, each cycle entry with a starting location offset and cycle length. The starting offset uniquely determines the cycle, and the cycle length determines the work-load.

In a static multi-thread implementation, with a given fixed number of threads, an optimization is needed to assign nearly same work-load to each thread. After this assignment, the data reshuffle can be carried out as a regular multi-threaded job.

In a dynamic multi-thread implementation, the next available thread picks up the next independent cycle from the cycle information table and completes the cycle. How to choose the next independent cycle among the remaining cycles in order to minimize the total runtime is a scheduling optimization. For example, a simple and effective method is to choose the task with largest load among the remaining tasks on the queue.

Further study of the multi-thread implementation on SMP architectures is important, but goes beyond the scope of this paper. (We also note that traditional two-array method can also parallelized using similar multi-thread approach.) Our original motivation is an in-place global remapping algorithm on distributed memory architecture, which we will discuss in the next section.

4 A Parallel Implementation on Distributed-memory Architectures

Index reshuffle of a global multidimensional array on a multi-processor distributed-memory system is, in essence, a remapping of problem subdomains. It involves local array index reshuffles and global data exchanges. The goal is to remap 3D array on processors such that data points along a particular dimension is entirely locally available on the processor, and that the data access along this dimension corresponds to the fastest running storage index, just as in the usual array reshuffle. As discussed in the introduction, normally the remapping will require an auxiliary array of the same size as the original data, due to the local index reshuffle of 3D arrays of non-equal dimension sizes. With the new in-place algorithm discussed above, the auxiliary array can be eliminated and we have an in-place parallel array remapping algorithm.

As shown in Figure 1, the essential remapping involves the last two dimensions of the 3D array $A(N_1, N_2, N_3)$. All data along the first dimension (latitude in Figure 1) are entirely local to a processor. They are moved around during the dynamical remapping as a single block. Thus this block may be viewed as an array element of a 2D array.

In this context, the remapping algorithm of a 3D array adopts the well-known communication algorithm of transposition of 2D arrays on distributed memory environments (see [8] and many papers referred there). Our main task here is to integrate the local in-place reshuffle algorithm to make the global remapping algorithm in-place; in doing so, we provide a concrete and efficient implementation.

4.1 Algorithm

Here we outline the in-place parallel remapping algorithm for a 3D array. Start with 3D array $A(N_1, N_2, N_3)$. Note that in the situation shown in Figure 1, processors 0,1,2,3 form an independent reshuffle group. The sizes here refer to the volume of data on these four processors. (Similar reshuffle also happens on processors 4,5,6,7 as another independent reshuffle processor group.)

In the beginning, the 3rd array dimension (longitude) are split among all P processors, i.e., on each processor, the local array (subdomain) is $A(N_1, N_2, N_3/P)$. The following simple steps will accomplish the remapping:

(G1) Do in-place two-index array reshuffle on the local array $A(N_1, N_2, N_3/P)$, between 2nd and 3rd indices.

(G2) Do global all-to-all exchange of data blocks, each of size $N_1(N_3/P)(N_2/P)$.

(G3) Do in-place two-index array reshuffle between 2nd and 3rd indexes on the local array which is viewed as $A(N_1N_3/P, N_2/P, P)$. The final local array is $A(N_1N_3/P, P, N_2/P)$, which can be equivalently viewed as $A(N_1, N_3, N_2/P)$.

In both steps (G1) and (G3), the algorithm in the previous section can be adopted without modification. In traditional implementation [8], the relevant small blocks are picked from A and inserted into the auxiliary array B in a more elaborate fashion; after step (G2), they are put back from B to A in an similar fashion. Our index reshuffle method simplifies these procedures. In case there is enough memory space for the auxiliary array B, this method is then a simplified version of the traditional method. In that case, the copy-back phase is not necessary: step (G1) moves data from A to B and step (G3) moves data from B back to A.

In step (G2), the global exchange does essentially a pair-wise block exchange where the local 3D arrays on each processor are viewed as an 1D array of blocks. This exchange involves all-to-all communications. Each processor sends $P-1$ blocks out, each to a different processor. Each processor also receives $P-1$ blocks, each from a different processor. The relevant code segment is well-known [9, 10, 7, 3, 8]:

```
! All processors simultaneously do the following:
do q = 1, P-1
   send a message to destination processor destID              (C.3)
   receive a message from source processor srcID
end do
```

There are two popular methods to determine `destID` and `srcID`. One method is to set `destID = srcID = (myID XOR q)`, here `myID` is the processor id, and XOR is the bit-wise exclusive OR operation. This is a pair-wise symmetric exchange communication. As `q` increases each step of the way, `destID` traverses over all other processors much like the hypercube broadcast tree algorithm [11]. On hypercube network, this algorithm is congestion free, and on the 3D mesh such as Cray T3E, it is also very effective. This algorithm needs to be slightly modified when P is not a power of 2. Another method is to set `destID = MOD(myID+q, P)`, and `srcID = MOD(myID-q, P)`. Here, P is not required to be power of 2. These send/receives can be implemented with `MPI_sendrecv`, requiring a buffer of size $N_1(N_3/P)(N_2/P)$.

The inverse array remapping follows the identical steps (G1), (G2), G3), except the starting array size is $A(N_1, N_3, N_2/P)$. As explained in section 2, the vacancy tracking cycles in the inverse remapping will be exactly the inverse of the forward remapping. Thus in actual implementation, we make use of the existing cycles, and traverse in reversed order in (G1) and (G3), as discussed in previous sections. Step (G2) remains identical in inverse remapping.

The above algorithm assumes that N_2 and N_3 are integer multiples of P for optimal efficiency. In actual implementation, N_2 and N_3 can be arbitrary, giving much more flexibility on size considerations. When one of N_2, N_3 is not multiples of P , the algorithm remains same, except that some holes in the final remapped array should be squeezed out. When both of N_2, N_3 are not multiples of P , some padding is needed before the remapping and the holes need to be squeezed out after the remapping.

In the situation shown in Figure 1, processors 0,1,2,3 form an independent exchange group; Processors 4,5,6,7 form another independent communication group. Using the communicator or processor group constructs provided in MPI, the above algorithm can be applied without any change, except that the dimension sizes and P refer to the array and processors in the processor group. Within each group, processors are still ranked from 0 to $P-1$, and the code segment (C.3) remains valid. MPI communicators will automatically distinguish different processor groups and send/receive messages to/from appropriate processors correctly.

4.2 Performance

This in-place parallel array remapping algorithm is implemented using Message Passing Interface (MPI) on Cray T3E and IBM SP. The implementation deals with the remapping indicated in Figure 1, with all processors participating in the single reshuffle group. The code is extracted from an ocean model parallel I/O module [5]. We fix the 3D array size to be $64 \times 512 \times 128$. (Several array sizes are tested and the timing ratio are similar. This

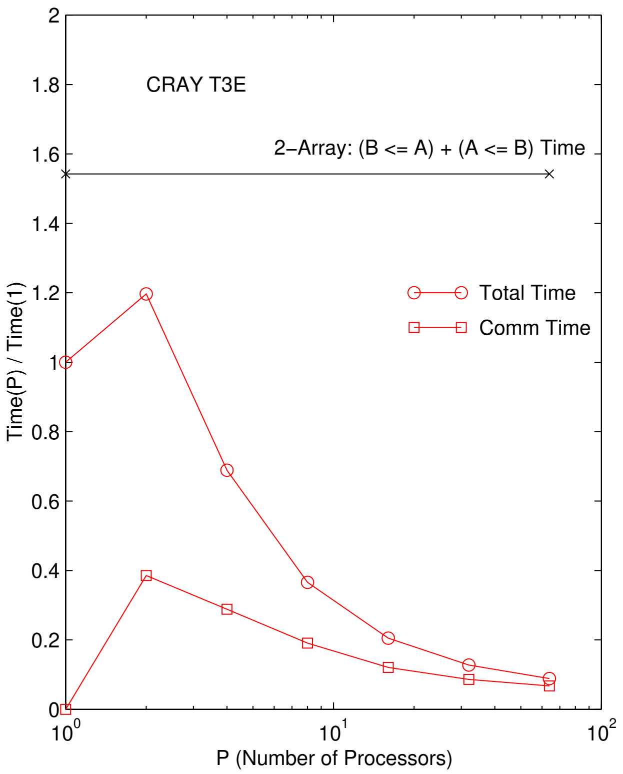

Figure 5: Timing for global array 64x512x128 remapping on 1, 2, 4, 8, 16, 32 and 64 nodes of Cray T3E using the in-place algorithm. Plotted are the ratio of the timing on P processors vs the timing on 1 processor. Both total time and communication time are shown. The two-array method on single processor is shown as the horizontal line.

size is chosen because it fits in the memory of a single processor.) This array is remapped between the two different decompositions as shown in Figure 1.

Timing for remapping on CRAY T3E is shown in Figure 5. Here the timing is the ratio between the remapping time on P processors and the time to reshuffle the array on one processor in which steps G2 and G3 drop out. On two processors, the global remapping is about 20% slower compared to reshuffling the entire 3D array locally. As the number of processors increases, the global remapping becomes faster and faster. On 64 processors, the global remapping is 12.3 times faster than reshuffling the entire 3D array locally.

The main reason for the reduced time on P processors is that the local array size in the local reshuffle is $N_1 N_2 N_3/P$, which is reduced by half as P is doubled, and so does local reshuffle time. This can be seen clearly from the actual timing, the difference between the total time and the communication time curves in Figure 5.

Communication time can be approximately calculated from a simple *latency* + *message-size/bandwidth* model. Assuming there are enough communication channels, and no traffic congestion on the network, every processor will spend the same time interval for the global exchange. Adding the local reshuffle time, we have the total global remapping time T_P on P processors:

$$T_P = 2MN_1N_2N_3/P + 2L(P-1) + [2N_1N_3N_2/BP][(P-1)/P] \tag{5}$$

where M is the average memory access time per element, L is the communication latency including both hardware and software overheads, and B is the point-to-point communication bandwidth.

Typically, the latency term is very small, thus the communication time ($T_{\rm comm}$) decreases steadily as number of processors increases, as seen in Figure 5. However, $T_{\rm comm}$ decreases much slower than Eq.5 would have predicted, due to two factors: number of communication channels and traffic congestion as more processors are involved.

Consider communication channels. In this all-to-all communication, the average case analysis is best described by bisection bandwidth, $B_{\rm bisec}$, the maximum bandwidth across a minimum bisection of the network[12]. In Eq.5 it is essentially assumed to be $B_{\rm bisec} = BP$. For d-dimensional mesh topology, the 3D Torus on T3E, $B_{\rm bisec} = BP^{(d-1)/d}$. For large P , this difference in exponent of P makes a big difference in scaling.

As P increases further, the latency term becomes important. Eventually, a saturation point, $P_{\rm sat}$ will be reached beyond that more processors will not reduce the total remapping time. From Eq.5, we have

$$P_{\rm sat} \simeq \left(\frac{d-1}{d} \frac{N_1N_2N_3}{LB} \right)^{d/(2d-1)} \tag{6}$$

for large P . $P_{\rm sat}$ depends on two fundamental characteristic quantities: LB, communication bandwidth times communication latency, and d, the dimension of the network topology. For T3E, we measured $L = 17\mu$sec and $B = 300$ MBytes/sec. Thus for array A(64,512,128), $P_{\rm sat} = 139$, consistent with Figure 5. In general, traffic congestion on T3E is small.

In Figure 6, timings for IBM SP are shown. The total time and communication time behave similar to those on T3E, but with a few clear differences. First, the communication time, relative to local reshuffle time, is much large on SP than on T3E. SP has faster local memory bandwidth, but slower communication bandwidth. Second, the saturation point is much smaller: $P_{\rm sat} = 32$. On SP, the network is a multi-stage fat-tree topology, which has a large bisection bandwidth: $B_{\rm bisec} = BP$, leading to

$$P_{\rm sat} \simeq (N_1N_2N_3/LB)^{1/2} \tag{7}$$

Using our measured parameters for SP, $L = 26\mu$sec and $B = 133$ MBytes/sec, we get the theoretical $P_{\rm sat} = 98$. The large discrepancy between the theoretical calculated $P_{\rm sat}$ and the

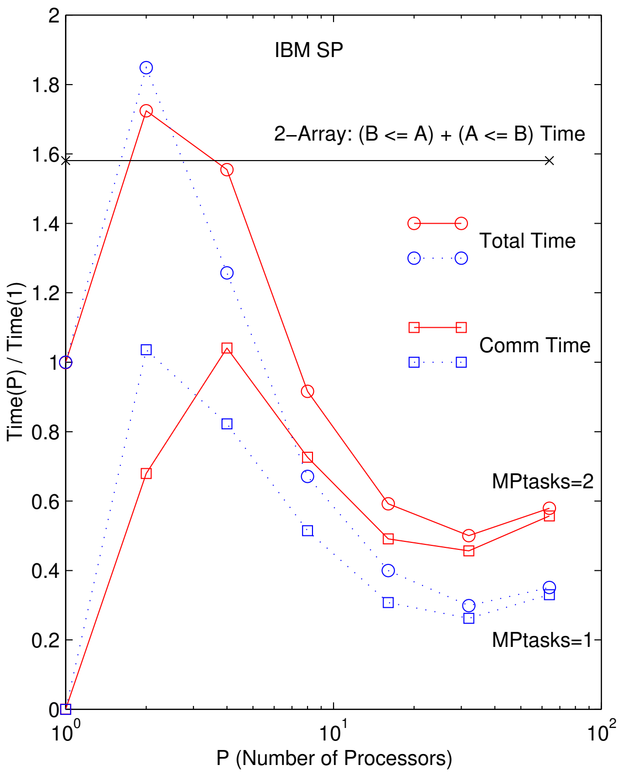

Figure 6: Timing for global array 64x512x128 remapping on 1, 2, 4, 8, 16, 32 and 64 processors of IBM SP using the in-place algorithm. Plotted are the ratio of timings on P processors vs that on 1 processor. Both total time and communication time are shown. The two-array method on a single processor is shown as the horizontal line. Each SP node has 2 POWER3 processors. MPtasks=1 indicates only 1 processor on each node is used; MPtasks=2 indicates both processors on a node are used.

actually measured $P_{\rm sat}$ indicates non-negligible traffic congestion and other communication factors with large number of nodes.

On SP, with current 2-way SMP nodes, traffic between the two processors on the same SMP node negatively affects the communication. For example, the same communication task on 32 nodes with only one processor per node used (indicated as MPtasks=1 in Figure 6) takes about half time as the same communication task between on 16 nodes with both two processors used (MPtasks=2 in Figure 6). (The improvement due to MP_MEMORY_SHARED=yes is included in the timing). This indicates that serially accessing the adaptor on the switch chip by the two processors on the same node are serialized, and is a serious bottleneck in communication. We note that all these results are consistent with accumulated general experiences on IBM SP.

5 Conclusions

We have described a new in-place multi-dimensional array index reshuffle algorithm following vacancy tracking cycles. This eliminates the need for auxiliary buffer arrays, and related copy-backs. Compared to the conventional method, this algorithm saves half of the total memory required for the reshuffle. Detailed implementation are given and its correctness is proved.

Although the vacancy tracking cycles seem to have a random memory access pattern, as the array element size reaches the cache line size, its disadvantages in memory access becomes insignificant compared to the conventional two-array reshuffle method. The minimum memory access property of the vacancy tracking algorithm becomes important for large element size and the algorithm runs slightly faster than the conventional method. If the time required for copying reshuffled data back to the original array is included in timing, the new in-place algorithm outperforms the conventional one by almost a factor of 2 for large element sizes. The algorithm have been implemented and tested on CRAY T3E and IBM SP and its effectiveness were shown.

Using the mutual independence of the vacancy tracking cycles, the algorithm can be parallelized on SMP architectures using a multi-threaded approach. On DMP architectures, the local vacancy tracking algorithm can be combined with an existing global exchange method leading to an efficient in-place global index reshuffle algorithm, for remapping problem subdomains. We described the global remapping algorithm, discussed some points in implementation, and carried out systematic tests on CRAY T3E and IBM SP. Parallel performances are analyzed and some useful observations are discussed.

This algorithm eliminates an important memory limitation in reshuffle/remapping of multidimensional arrays on sequential, SMP and DMP computer architectures while improving performance at same time. In addition, we believe the algorithm will have other applications in which storage is a serious consideration, such as out-of-core methods and database reorganizations.

Acknowledgment. I thank Drs. Horst Simon and David Bailey for valuable discussions, and for pointing out reference [6] to me while discussing an earlier version of this paper. I also thank anonymous referees for suggestions and comments that helped improving the paper. This work is supported by Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, and Office of Biological and Environmental Research, Climate Change Prediction Program, of the U.S. Department of Energy under contract number DE-AC03-76SF00098. Initial idea and preliminary implementation were developed while I was working on a NASA HPCC project at the Jet Propulsion Laboratory.

References

[1] J.J. Hack, J.M.Rosinski, D.L.Williamson, B.A.Boville and J.E. Truesdale, "Computational Design of NCAR community climate model", Parallel Computing, v.21, pp.1545-1555, 1995.

[2] J.Drake, I.Foster, J.Michalakes, B.Toonen and P. Worley, "Design and performance of a scalable parallel community climate model", Parallel Computing, v.21, pp.1571-1581,1995.

[3] I. T. Foster and P. H. Worley. "Parallel algorithms for the spectral transform method," SIAM J. Sci. Stat. Comput., v.18, pp. 806-837. 1997.

[4] A.A. Mirin, D. Shumaker, M.F. Wehner. "Efficient Filtering Techniques for Finite-Difference Atmospheric General Circulation Models on Parallel Processors." Parallel Computing, v.24, pp.729-740, 1998.

[5] C.H.Q. Ding and Y. He, "Data Organization and I/O in a parallel ocean circulation model", Lawrence Berkeley National Lab Tech Report 43384. Proceedings of Supercomputing '99, Nov 1999.

[6] D. Fraser, "Array Permutation by Index-Digit Permutation," J. ACM, v.23. pp.298-309, 1976.

[7] A. Edelman, S. Heller and S. L. Johnsson. "Index Transformation Algorithms in a Linear Algebra Framework", IEEE Transactions on Parallel and Distributed Systems v.5, pp.1302–1309, 1994.

[8] V. Kumar, A. Grama, A. Gupta, G. Karypis. Introduction to Parallel Computing. Benjamin/Cummings, Redwood City, CA. 1994.

[9] S.L. Johnsson and C.-T. Ho. "Matrix transposition on boolen n-cube configured ensemble architectures", SIAM J. Matrix Anal. Appl. v.9. pp.419-454, 1988.

[10] S.H. Bokhari. "Complete Exchange on the Intel iPSC-860 hypercube", Technical Report 91-4, ICASE, 1991.

[11] G.Fox, M.Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker. Solving Problems on Concurrent Processors. Vol 1. Prentice Hall. Englewood Cliffs, New Jersey. 1988.

[12] J.L. Hennessy, D.A. Patterson. Computer Architecture: A Quantitative Approach, 2nd Ed. Morgan Kaufmann, 1995.