MPI AND OPENMP PARADIGMS ON CLUSTER OF SMP ARCHITECTURES: THE VACANCY TRACKING ALGORITHM FOR MULTI-DIMENSIONAL ARRAY TRANSPOSITION

YUN HE* AND CHRIS H. Q. DING*

Abstract. We evaluate remapping multi-dimensional arrays on cluster of SMP architectures under OpenMP, MPI, and hybrid paradigms. Traditional method of multi-dimensional array transpose needs an auxiliary array of the same size and a copy back stage. We recently developed an in-place method using vacancy tracking cycles. The vacancy tracking algorithm outperforms the traditional 2-array method as demonstrated by extensive comparisons. Performance of multi-threaded parallelism using OpenMP are first tested with different scheduling methods and different number of threads. Both methods are then parallelized using several parallel paradigms. At node level, pure OpenMP outperforms pure MPI by a factor of 2.76 for vacancy tracking method. Across entire cluster of SMP nodes, by carefully choosing thread numbers, the hybrid MPI/OpenMP implementation outperforms pure MPI by a factor of 3.79 for traditional method and 4.44 for vacancy tracking method, demonstrating the validity of the parallel paradigm of mixing MPI with OpenMP.

Key words. multidimensional arrays, array transpose, dynamical data remapping, index reshuffle, vacancy tracking cycles, global exchange, MPI, OpenMP, hybrid MPI/OpenMP, SMP cluster.

AMS subject classifications.

1. Introduction. Large scale highly parallel systems based on cluster of SMP architectures are today's dominant computing platforms. OpenMP has recently emerged as the definitive standard for parallel programming at the SMP node level[1][2]. Using MPI to handle communications between SMP nodes and OpenMP within nodes, the MPI/OpenMP hybrid parallelization paradigm is the emerging trend for parallel programming on cluster of SMP architectures[3] [4] [5] [6] [7], especially for those applications which exhibit naturally a two-level parallelism.

The MPI/OpenMP hybrid parallelization paradigm is elegant in both conceptual and architectural level. At node level, use of OpenMP avoids the extra communications that MPI would require. It also allows the fine granularity of the application, making good usage of shared memory with increased dynamic load balancing. MPI is used to handle coarse-grained parallelism among SMP nodes, with larger message sizes therefore more efficient communications.

Although the hybrid MPI/OpenMP could have better performance than both pure MPI, However, in practice, performance of many large scale applications indicate otherwise (for example, NAS benchmarks[4], conjugate-gradient algorithms[5] and particle simulations[7]). There are some positive experiences[3][6], but it is often the case that existing applications using pure MPI paradigm over all processors and ignoring the shared memory nature among the processors on the single SMP node outperform the same application codes utilizing the hybrid OpenMP with MPI paradigm.

In this paper, we provide an in-depth analysis on remapping problem domains on cluster of SMP architectures under several OpenMP, MPI, and hybrid paradigms.

Dynamically remapping problem domains 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.

*NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. (yhe@lbl.gov, chqding@lbl.gov)

For example, 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[8] and the shallow water equation[9], 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 data input/output[10].

To transpose a multidimensional array A, say between 2nd and 3rd indices, the conventional method uses an auxiliary array B of same size of A.

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

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 as 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 reshuffle results are stored at the same location as the original array A. The issue of memory copy in OpenMP were studied in [11].

Recently, a vacancy tracking algorithm for multidimensional array index reshuffle is developed[12] that can perform the transpose in Eq.2 in-place, i.e., without requiring the auxiliary array B. This reduces the memory requirement by half, therefore lifting a severe limitation on memory-bound problems.

Both the conventional two-array method and the new vacancy tracking algorithm can be implemented on cluster of SMP architectures using OpenMP, MPI, and hybrid MPI/OpenMP paradigms. We have investigated them systematically. Some preliminary results were presented at SC2002[13]. Here we perform more systematic studies on both sequential and distribited platforms for both methods. Our main results are that (i) Vacancy tracking algorithm outperforms conventional two-array method in all situations. (ii) OpenMP parallelism performs slightly better than MPI for the traditional two-array method but is substantially faster than MPI parallelism for the vacancy tracking method on a single SMP node. (iii) On up to 128 CPUs, the hybrid paradigm performs about a factor of 4 faster than pure MPI paradigm for both methods. (iv) Contrary to some existing negative experience of developing hybrid programming applications, our hybrid MPI/OpenMP implementation for the vacancy tracking algorithm outperforms pure MPI by a factor of 4.44.

2. Vacancy Tracking Algorithm . Array transpose can be viewed as a mapping from original memory locations to new target memory locations. The key idea of this algorithm is to move elements from old locations to new locations in a specific memory-saving order, by carefully keeping track of the source and destination memory locations of each array element. When an element is moved from its source to new location, the source location is freed, i.e., a vacancy. This means another appropriate element can be moved to this location without any intermediate buffer. Then the source location of that particular element becomes a vacancy, and yet another element is moved directly from its source to this destination. This is repeated several times, and a closed loop of vacancy tracking cycles is formed.

Consider transposition of a 2D array A(3,2). Six elements of A(3,2) are labeled as A_0,A_1, A_2, A_3, A_4, A_5, and are stored in six consecutive memory locations L_0, L_1, L_2, L_3, L_4, L_5 (shown in the leftmost layout in Figure 1). The transposition is accomplished by moving elements following the vacancy tracking cycle

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

Move content in L_1 to a temporary buffer, now L_1 becomes the vacancy; Move content in L_3 to L_1, now L_3 becomes the vacancy; Move content in L_4 to L_3; Move content in L_2 to L_4; Move content in buffer to L_2. Note that contents in L_0 and L_5 are not touched, because they are already in the correct locations. Assume the buffer is a register in CPU, the total memory access is 4 memory writes and 4 memory reads. In contrast, the conventional two-array transpose algorithm will move all 6 elements to array B, and copy them back to A, with a total of 12 reads and 12 writes. The vacancy tracking algorithm achieves the optimal (minimum) number of memory access.

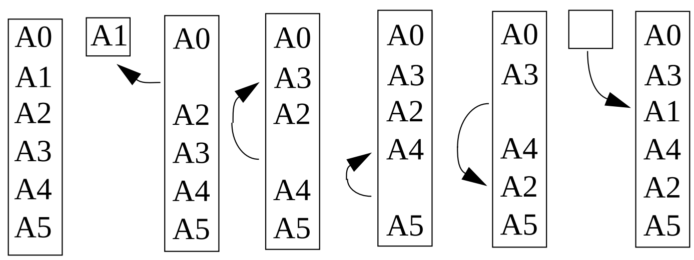

FIG. 1. *Transposition for the array A(3,2).* L_1, L_3, L_4, L_2 *are successive vacancies.*

Vacancy tracking algorithm applies to many other index operations more complex than two-index reshuffle. For example, the simultaneous transpose of three indexes, the left-circular-shift,

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

can be easily accomplished. For a 3D array $A(4, 3, 2)$ with 24 elements, the three-index left-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
```

A simple algorithm to automatically generate the cycles for 2 indices transpose is

```
! 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.1)
   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 `index_to_offset` and `offset_to_index` are two simple routines that converts two-dimensional *index* from/to one-dimensional *offset*. A slight modification can handle 3 indices operations. The cycle information will be stored in a table first in the actual implementation and the outer do loop (C.1) is performed to move the data from actual memory locations.

We assess the effectiveness of the algorithm by comparing the timing results between the traditional 2-array method and the in-place vacancy tracking method for the index reshuffling of a three-dimensional array $A(N_1, N_2, N_3)$ by moving around the array elements in the block size of the first dimension as shown in Eq.(2). The algorithm is implemented in F90, and tests are carried out on a POWER3 IBM SP. We use compiler option -O5 for highest level of optimization for both methods.

Figure 2 shows the ratio of timing between the two-array method (include array copy back) and in-place algorithm for three array sizes. In-place algorithm performs better for almost all the array sizes except for N $\leq$ 4. For large array size, vacancy tracking algorithm achieves more than a factor of 3 speedup for N_3 = 8, 16, and 64.

The number of memory access and the access pattern could explain the speedup. First, the in-place algorithm eliminates the copy back phase so it reduces memory access by half. Second, for a 2D array $A(N_1, N_2)$, the number of memory access required for the $B[k_2, k_1] \Leftarrow A[k_1, k_2]$ reshuffle phase of a two-array method is $N_1 N_2$. But for the in-place method, the total lengths of all cycles would be $N_1 N_2$. The length-N cycle involves N-1 memory-to-memory copies, one memory-to-`tmp` copy and one `tmp`-to-memory copy. Normally, access to the `tmp` storage is a register or cache, and is much faster than the DRAM access. We could safely count the number of a memory access for the length-N cycle as `N`. Meanwhile, all the length-1 cycles means the memory locations are untouched, thus saves the number of memory access. Third, on cache-based processor architectures, the memory access pattern is as important as the number of memory access. Though memory access pattern for the in-place method seems more random than traditional method, the number of bytes moved is often large for the problems the method is targeted for. The gap is reduced and the memory access in vacancy tracking algorithm is not irregular at scales relevant to cache performance when the size of the move is larger than cache-line size, which is 128 bytes (16 real*8 data elements). Also, the $B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]$ reshuffle phase of the two-array method has the same disadvantage of the in-place method: not efficient memory access due to the large stride.

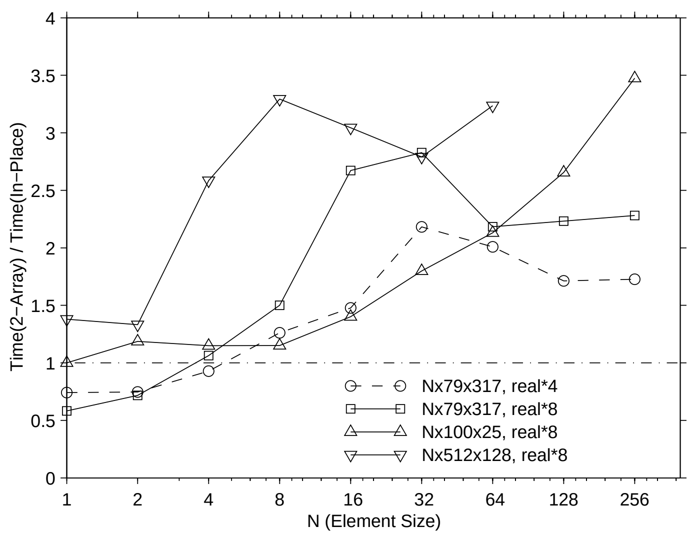

FIG. 2. *Timing for a local 3D array index reshuffle on IBM SP. Plotted are the ratio of timing between the two-array method (include array copy back) and in-place algorithm.*

3. Parallel Paradigms on Cluster SMP Architectures.

3.1. Multi-Threaded Parallelism. The traditional two-array method adopts a nested do loops to perform index reshuffles. The OpenMP parallelism could be added straightforwardly. The pseudo code is as follows:

```
!$OMP PARALLEL DO DEFAULT (PRIVATE)
!$OMP&         SHARED (N3, N2, A, B)
!$OMP&         SCHEDULE (AFFINITY)
      do i3 = 1, N3
      do i2 = 1, N2
         B(:,i2,i3) = A(:,i3,i2)          (C.2)
      end do
      end do
!$OMP END PARALLEL DO
```

The vacancy tracking algorithm can also be easily parallelized using a multi-threaded approach in a shared-memory multi-processor environment to speed up data reshuffles, e.g., to re-organize a database on an SMP server. As mentioned in Ding[12], 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.1) 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.

The pseudo code for the OpenMP implementation for the main loop for each vacancy tracking cycle is as follows:

```
!$OMP PARALLEL DO DEFAULT (PRIVATE)
!$OMP&         SHARED (N_cycles, info_table, Array)
!$OMP&         SCHEDULE (AFFINITY)
      do k = 1, N_cycles
         an inner loop of memory exchange for each cycle using info_table   (C.3)
      enddo
!$OMP END PARALLEL DO
```

Proper scheduling the independent cycles to threads are important. The workload is based on the cycle lengths. So, it could be done either statically or dynamically. 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. Or, if the cycles are short, we could group cycles into chunks so that each thread will pick up a chunk instead of an individual cycle to minimize thread overhead.

3.2. Pure MPI Parallelism. Transposition of a global multi-dimensional array distributed on a distributed-memory system is a remapping of processor 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 the data access along this dimension corresponds to the fastest running storage index, just as in the usual array transpose.

Using MPI to communicate data between different processors is the best paradigm for distributed memory architectures such as Cray T3E, where each node has only a single processor. On cluster of SMP architectures, such as IBM SP, each node has, say, 16 processors and they share a global memory space on the node.

Running MPI between different processors on the same node is equivalent to communicating messages using inter-process communication (IPC) between different Unix processes under the control of a single operating system running on the SMP node. Therefore, a simulation code compiled for 64 processors can successfully run on 4 SMP nodes with 16 processors on each node, since the OS automatically replaces MPI calls by IPC when the relevant inter-processor communications are detected to be within a single SMP node. Communications between processors residing on different SMP nodes will go on to the inter-node communication networks. This "pure" MPI paradigm therefore has the desired portability and flexibility.

Here we outline the algorithm for remapping a 3D array $A(N_1, N_2, N_3)$ regarding the 2nd and 3rd indices using pure MPI (see more details in [12]). For a global multi-dimensional array, it involves local array transpose and global data exchange. Use $A(N_1, N_2, N_3)$ as an example, to transpose it to be $A(N_1, N_3, N_2)$ on P MPI processors, the steps are:

(G1) Do a local transpose on the local array $A(N_1, N_2, N_3/P) \Rightarrow A(N_1, N_3/P, N_2)$

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

(G3) Do a local transpose on the local array viewed as $A(N_1 N_3/P, N_2/P, P)$
$\Rightarrow A(N_1 N_3/P, P, N_2/P)$ viewed as $A(N_1, N_3, N_2/P)$.

Local in-place algorithm is used for steps (G1) and (G3). For step (G2) global exchange, the following all-to-all communication pattern[9][14][15] is used:

```
! All processors simultaneously do the following:
do q = 1, P - 1
   send a message to destination processor destID       (C.4)
   receive a message from source processor srcID
end do
```

Here we adopt `destID = srcID = (myID XOR q)`, where `myID` is the processor id, and `XOR` is the bit-wise exclusive OR operation. This is a pairwise symmetric exchange communication. As `q` loops through all processors, the `destID` traverses over all other processors.

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:

$$T_P = 2MN_1N_2N_3/P + 2L(P-1) + [2N_1N_3N_2/BP](P-1)/P \qquad (4)$$

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.

3.3. Hybrid MPI/OpenMP Parallelism. The emerging programming trend on cluster of SMP is to use MPI between SMP nodes and use multi-threaded OpenMP on the processors within an SMP node. This matches most logically with the underlying system architecture.

A variant of this hybrid parallelism is to create several MPI tasks (Unix processes) on an SMP node and use multi-threaded OpenMP within each such MPI task. For example, on 4 SMP nodes with 16 processors per node, one may create a total of 8 MPI tasks; within each MPI task, one may create 8 threads to match 8 CPUs per MPI task. Therefore, the pure MPI parallelism of Section 3.2 can be viewed as a special case of this hybrid paradigm, where one simply creates 4×16=64 MPI tasks for each of the 64 CPUs on the 4 SMP nodes.

For the remapping problem on cluster SMP architectures, the array is decomposed into subarrays owned by each MPI task. Local transpose will be done by each MPI task, with the choice of either the traditional two-array method or vacancy tracking method. It is parallelized with the multiple threads created by each MPI task. The total number of vacancy tracking cycles shared by the threads are determined by the local array size. After that, global data exchange is done among all the MPI tasks, and another local transpose as in (G.3) is performed. The timing analysis can be approximated by

$$T = 2MN_1N_2N_3/N_{CPU} + 2L(N_{MPI} - 1) + [2N_1N_3N_2/BN_{MPI}](N_{MPI} - 1)/N_{MPI} \quad (5)$$

where N_{CPU} is the total number of CPUs in the system and N_{MPI} is number of MPI tasks created.

As the number of MPI tasks increases, the local array size is reduced and so does local reshuffle time (first term in Eqs. 4 and 5). As importantly, as the number of MPI tasks reduces from N_{CPU} to the number of SMP nodes N_{SMP}, the total communication volume decreases, and so does the communication time. (Here we simply assume the communication rates between MPI tasks are the same. In practice, communication between MPI tasks on the same SMP nodes are faster than those between different SMP nodes.) Thus theoretically, we expect the choice $N_{MPI} = N_{SMP}$ is optimal.

A specific issue regarding the speedup on more threads is that vacancy tracking cycles are split among different threads; thus reduce the local reshuffle time as well.

Given the same amount of resources (the total number of CPU), the more MPI tasks we choose, the less available CPUs that OpenMP threads could use, and *vice versa*. No OpenMP parallelism is added in the global exchange stage, the local array size and the number of MPI tasks related to the network communication determine the time spent in this stage. To gain the best performance from the above analysis, we need to utilize an optimal combination of MPI tasks and OpenMP threads on cluster of SMP architectures.

4. Performance.

4.1. Scheduling for OpenMP Parallelism. We tested different scheduling methods combined with different number of the threads used on different array sizes within one IBM SP node. The algorithm is implemented in F90 with OpenMP directives, and tests are carried out on a 16-way SMP node IBM SP. Notice, with OpenMP directives (use compiler option -qsmp), optimization level specified as -O5 in compiler option will change the loop structures so that the results are incorrect. We use level -O4 instead. The tested schedules are:

- `Static`: Loops are divided into *N_threads* partitions, each containing *ceiling (N_iterations / N_threads)* iterations. Each thread is responsible for one partition.
- `Affinity`: It is an IBM extension, and not part of the OpenMP standard. Loops are divided into *N_threads* partitions, each containing *ceiling (N_iterations / N_threads)* iterations. Then each partition is subdivided into chunks containing *n* (if specified) or *ceiling (N_remain_iterations_in_partion / 2)* (if not specified) iterations. Each thread takes a chunk from its partition first, if none left, then takes a chunk from another thread.
- `Guided`: Loops are divided into progressively smaller chunks until the minimum size of chunk (default 1) is reached. The first chunk contains *ceiling (N_iterations / N_threads)* iterations. Subsequent chunk contains *ceiling (N_remain_iterations / N_threads)* iterations. Threads taking chunks on a first-come-first-serve basis.
- `Dynamic`: Loops are divided into chunks containing *n* (if specified) or *ceiling (N_iterations / N_threads)* (if not specified) iterations. We choose different chunk sizes. Threads taking chunks on a first-come-first-serve basis.

For the traditional two-array method, the tested array size is $64 \times 512 \times 128$. Table 1 lists the timing results obtained from ensemble average of 100 test runs. A star is marked for the fastest timing among all different number of threads. Among all the schedules, `static` and `affinity` have very close timings and are overall the fastest. reasonable speedup is gained for up to 16 threads with these schedules. Although we could set up number of threads as many as we like, there is no gain in performance beyond 16 threads on the 16-way IBM SP.

The overhead of threads creation often deteriorates the performance as shown in columns for $N_{threads} = 32$.

TABLE 1
Timing for Array Size 64×512×128 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Traditional Two-Array Method (Time in seconds)

Schedule	64×512×128				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	44.1*	28.7*	46.1*	90.4*	164.8
Affinity	48.1	27.8*	45.8*	90.6*	164.0
Guided	51.9	45.9	52.6	91.3*	160.8*
Dynamic,1	48.1	50.9	60.5	104.6	191.8
Dynamic,2	50.0	49.6	59.7	99.4	173.7
Dynamic,4	47.6	50.4	55.8	100.0	169.3
Dynamic,8	46.1	48.0	53.7	99.4	173.8
Dynamic,16	56.9	60.3	57.1	97.7	169.1
Dynamic,32	86.9	93.8	87.6	98.7	167.9
Dynamic,64	150.9	163.1	152.5	97.4	162.1
Dynamic,128	296.1	327.0	298.4	328.3	331.9
Dynamic,256	296.3	326.2	301.3	333.0	324.5

For the vacancy tracking method, the tested array sizes are: (i) 64×512×128, N_cycles = 4114, *cycle_lengths* = 16; (ii) 16×1024×256, N_cycles = 29140, *cycle_lengths* = 9, 3; (iii) 8×1000×500, N_cycles = 132, *cycle_lengths* = 8890, 1778, 70, 14, 5; and (iv) 32×100×25, N_cycles = 42, *cycle_lengths* = 168, 24, 21, 8, 3.

TABLE 2
Timing for Array Sizes 64×512×128 and 16×1024×512 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Vacancy Tracking Method (Time in seconds)

Schedule	64×512×128					16×1024×256				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	34.1	15.3	15.0	25.3	47.4	34.2*	17.8	24.9	42.4	83.6
Affinity	28.8*	10.8*	13.9*	24.7*	47.2*	34.2*	15.5*	23.0*	42.0*	83.1*
Guided	35.3	14.2	20.8	30.4	47.5	38.0	17.6	27.4	46.8	83.3
Dynamic,1	32.3	16.5	22.9	36.1	58.6	358.8	348.7	55.6	68.0	151.6
Dynamic,2	32.6	16.1	22.3	34.7	55.7	180.6	165.8	37.5	61.6	103.3
Dynamic,4	33.8	16.7	22.7	35.6	54.7	39.9	23.3	35.5	58.2	98.8
Dynamic,8	32.4	16.1	21.4	33.5	53.9	39.4	21.4	33.3	54.3	94.8
Dynamic,16	30.3	16.0	22.8	33.6	53.3	36.9	20.6	31.4	52.5	93.0
Dynamic,32	28.9	16.2	21.4	32.4	52.4	38.9	21.2	31.4	62.3	91.5
Dynamic,64	34.9	16.0	20.5	32.8	59.9	38.6	20.2	29.9	50.9	89.9
Dynamic,128	28.9	16.1	20.0	35.8	51.0	33.5	19.2	29.6	64.9	88.9
Dynamic,256	29.5	16.0	20.0	31.8	52.7	34.4	18.9	29.8	50.0	87.6
Dynamic,512	-	-	-	-	-	34.5	19.9	29.9	63.2	87.9
Dynamic,1024	-	-	-	-	-	35.3	19.6	30.7	49.0	87.2

Tables 2 and 3 list the timing results obtained from ensemble average of 100 test runs. It is shown that with large number of cycles and small cycle lengths, schedule `affinity` is among the best; and with small number of cycles and large or uneven cycle lengths, `dynamic` schedule with small chunk size is preferred. This holds true for almost all number of threads tested. Reasonable speedup is reached with schedules `guided` and `affinity` for relative large arrays up to 16 threads used. There are limited speedup with some of the schedules, even in some cases speed-down, especially for smaller arrays. It is due to the large overhead associated with the creation of the threads, for example, array size 16×1024×256 with `Dynamics,1` and `Dynamics,2` schedules.

TABLE 3

Timing for Array Sizes 8×1000×500 and 32×100×25 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Vacancy Tracking Method (Time in seconds)

Schedule	8×1000×500					32×100×25				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	57.9	58.8	93.3	158.3	261.5	18.7	2.84	1.49	1.34	1.10
Affinity	48.5	38.0	59.3	86.3	158.0	16.6	1.88	1.72	0.95	1.31
Guided	58.0	58.4	92.7	159.9	261.4	15.3*	3.99	1.71	1.93	1.08*
Dynamic,1	47.1*	32.7*	49.7*	84.0	147.2	19.3	0.81*	1.05*	0.94*	1.35
Dynamic,2	50.8	32.7*	51.9	82.5*	145.8	17.0	2.68	1.12	0.97	1.38
Dynamic,4	56.9	37.8	53.9	83.7	144.3	17.0	3.45	2.03	0.98	1.24
Dynamic,8	63.3	52.7	52.3	82.5*	144.1*	16.5	3.29	2.68	1.57	1.28
Dynamic,16	107.9	92.1	92.2	90.2	148.6	18.7	4.28	3.20	2.09	1.33
Dynamic,32	165.1	159.4	158.8	187.8	155.8	-	-	-	-	-

4.2. Pure MPI and Pure OpenMP Parallelisms within One Node. Figure 3 shows the performance of the parallel implementation with different number of threads (pure OpenMP) or processors (pure MPI) within one node on 16-way SMP of IBM SP with each method, respectively. As tested in Ding[12], vacancy tracking method performs better with real*8 than real*4 as compared to two-array method. We use real*8 in this test for all MPI data types. The array sizes are chosen to be fit in local memory of one processor.

Clearly, both parallelisms with both methods have quite good speedup, except that the MPI performance with vacancy tracking method at 2 processors has a drop, which is due to the increased communication overhead compared to the entire local remapping. The vacancy tracking method is about twice faster than the two-array method. Both methods indicate a better OpenMP scaling than MPI scaling. With two-array method, OpenMP is slightly faster than MPI. With vacancy tracking method, OpenMP outperforms MPI substantially at all times. With 16 threads, it is faster than MPI by a factor of 2.76 for array size 64×512×128 and by a factor of 1.99 for array size 16×1024×256. Thus it makes sense to develop a hybrid MPI/OpenMP parallelism, however, experiences of other researchers[3][7] indicate that a better performance is not guaranteed[3].

4.3. Pure MPI and Hybrid MPI/OpenMP Parallelisms across Nodes. Figure 4 shows the performance of the parallel implementation across several nodes on IBM SP with each method, respectively. (timing with total CPUs = 16 is plotted for comparison) with array size 64×512×128 (results for array size 16×1024×256 are very similar). We use schedule `affinity` for OpenMP parallelism in both methods. The timing for both methods are very close while the speedup values are about twice in the two-array method than those in the vacancy tracking method. This is due to the ratio of sequential running time for both methods used as base to calculate the speedups is about 2 to 1. The closeness of total remapping timing in these methods is due to the fact thay the majority (over 90%) of total timing is spent at the global exchange stage, which is only parallelized by pure MPI.

The maximum number of threads we could efficiently use for one MPI task per node is 16. Pure MPI [NETWORK.MPI=SHARED is already utilized] does not scale above 16 processors. Using one MPI task per node with full 16 threads at each node results even worse performance at 32 total CPUs, although close or better performance than pure MPI are achieved at 64 and 128 total CPUs. The dip at 32 CPUs for both pure MPI and hybrid MPI/OpenMP parallelisms could be explained by the large across-node communication overhead for the global exchange stage (last term in Eq.5) which accounts for more than 90% of total remapping time.

Given total number of CPUs (N_{CPU}), we could adjust the number of processors to be

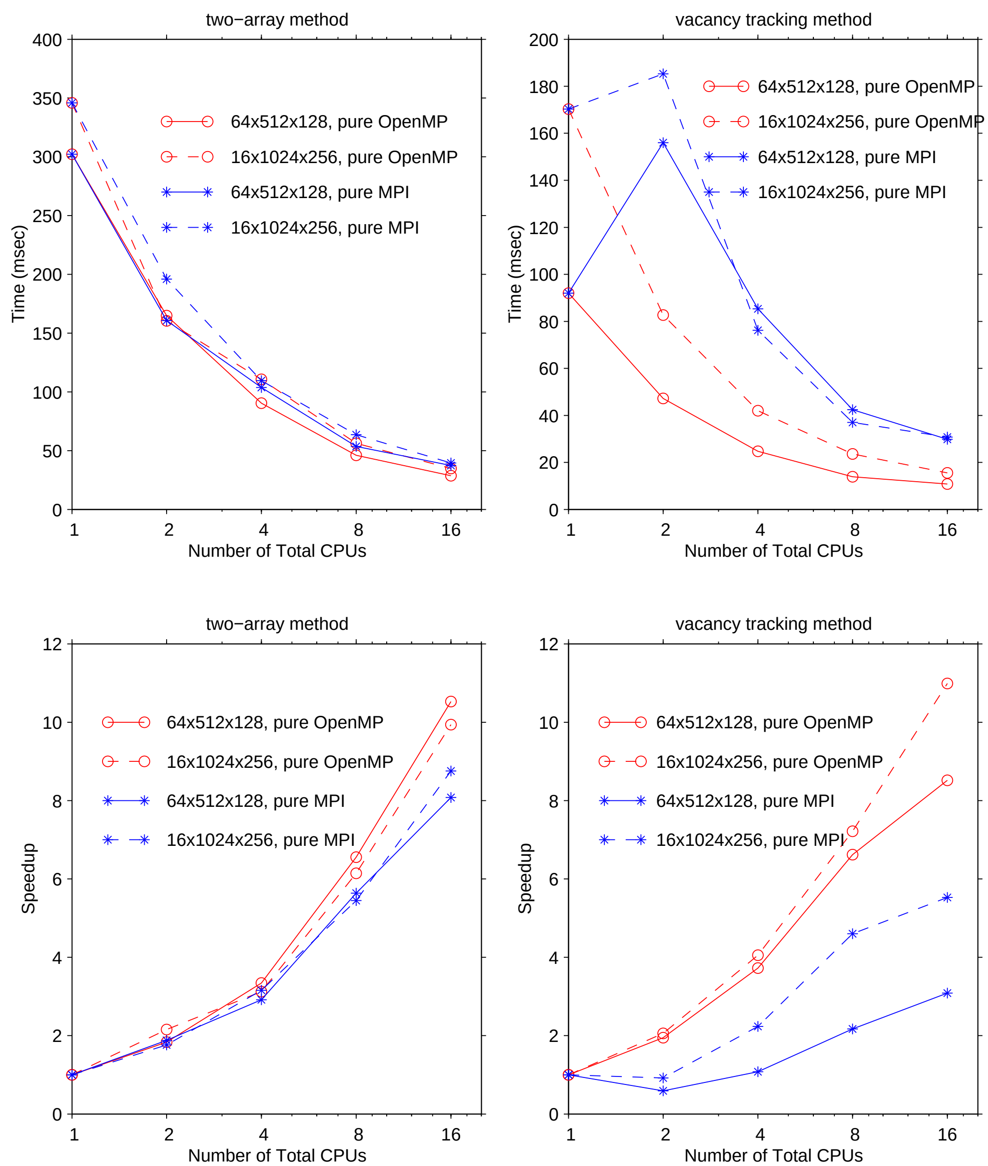

FIG. 3. *Time and speedup for global array remapping on different number of processors (pure MPI) or threads (pure OpenMP).*

used as MPI tasks (N_{MPI}) and number of threads per MPI task ($N_{threads}$). An example of choices for N_{MPI} and $N_{threads}$ with N_{CPU} = 64 would be:

```
N_CPU = N_MPI * N_threads
  64        4        16
  64        8         8
  64       16         4
```

The communication overhead is reduced when MPI tasks within same node utilize the in-node MPI network. The different subarray size each MPI task owns also contributes to the timing difference. Although we use the same number of total CPUs, the subarray sizes

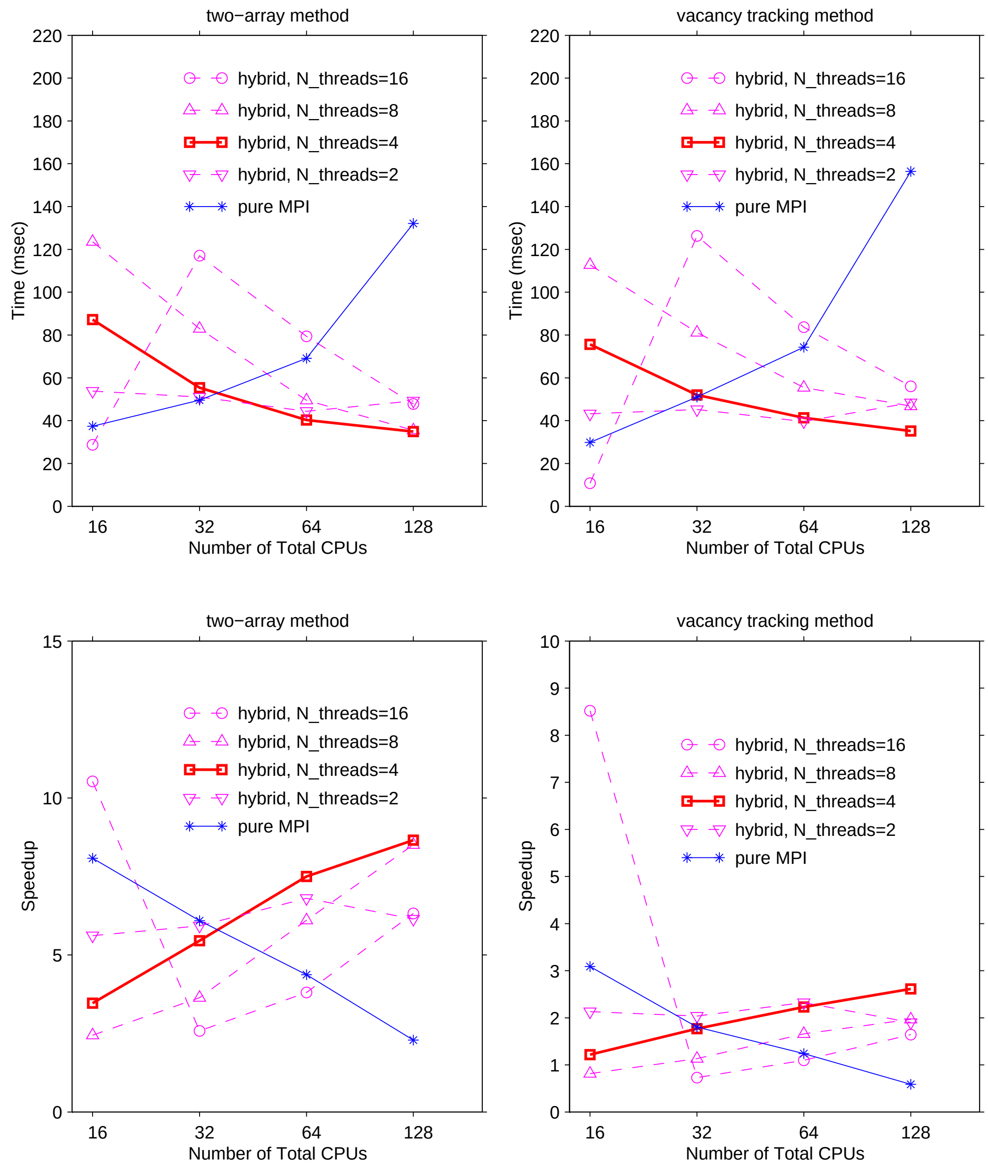

FIG. 4. *Time and speedup for global array remapping with different combinations of MPI tasks and OpenMP threads for array size 64×512×128.*

are determined by the number of MPI tasks; thus the numbers of vacancy tracking cycles in the loop for different N_{MPI} are different. Since we only have OpenMP directives for the local reshuffles, we need to find an optimal combination of MPI tasks and OpenMP threads to achieve the overall best performance.

From our experiments, $N_{threads} = 4$ gives an overall best performance among all other number of threads. At 128 total CPUs, the hybrid MPI/OpenMP parallelism with $N_{threads} = 4$ performs faster than with $N_{threads} = 16$ by a factor of 1.37 (two-array method) and 1.59 (vacancy tracking method), respectively; and it performs faster than pure MPI parallelism by a factor of 3.79 (two-array method) and 4.44 (vacancy tracking method), respectively.

Although it still does not have better performance than with $N\_threads = 16$, the hybrid MPI/OpenMP parallelism scales up from 32 CPUs to 128 CPUs, compared to scales down with pure MPI.

5. Summaries and Discussions. In this paper, we first assess the effectiveness of an in-place vacancy tracking algorithm for multi-dimensional data remapping by comparing the timing results with traditional 2-array methods. We tested with different array sizes on IBM SP. The in-place method outperforms the traditional method for all the array sizes when the data block to move is not too small. The speedup we gain for a big array is 3.24. This could be explained by two factors: 1) the elimination of the auxiliary array thus the copy back; 2) the memory access volume and pattern.

The vacancy tracking algorithm is efficient and easy to parallelize with OpenMP in a shared programming model. The independency of the vacancy tracking cycles allows us to parallelize the in-place method using a multi-threaded approach in a shared-memory processor environment to speed up data reshuffles. The vacancy tracking cycles are non-overlapped so multiple threads can process each tracking cycles simultaneously. We extensively tested the different thread scheduling methods on 16-way IBM SP and found that schedule *affinity* optimizes the performance for arrays with large number of cycles and small cycle lengths, while *dynamic* schedule with small chunk size is preferred for arrays with small number of cycles and large or uneven cycle lengths. Meanwhile, the OpenMP parallelism could be used directly upon the nested loops of the memory copy for the traditional two-array mothod. The timing results show that schedules `static` and `affinity` are among the best.

On distributed memory architectures, both methods could be parallelized using pure MPI with the combination of local array transpose method and an existing global exchange method. Based on the fact that pure OpenMP performs faster and has better scaling than pure MPI on a single node of IBM SP, we are encouraged to develop a hybrid MPI/OpenMP approach on cluster SMP architectures for an efficient global data remapping algorithm.

We discussed the algorithms of the hybrid approach, advantages and disadvantages of choosing the number of MPI tasks and OpenMP threads if the total number of nodes is given, and carried out systematic tests on IBM SP. For both array sizes we tested, pure MPI does not scale beyond total 16 processors, in fact, scales down from 32 processors to 128 processors. But by using hybrid MPI/OpenMP approach, and by carefully choosing the number of threads per MPI task, we gained about a factor of 4 speedup for both the two-array method and the vacancy tracking method from pure MPI.

In a distributed memory model, the local transpose is only applied within each MPI task and a standard all-to-all communication algorithm is used across the nodes. Thus the OpenMP parallelization is more efficient than the MPI parallelization within one SMP node. As expected, the hybrid OpenMP/MPI parallel performance is in between.

Contrary to some existing negative experience in hybrid programming applications, this paper gives a positive aspect of developing hybrid MPI and OpenMP parallel paradigms for real applications. The vacancy tracking algorithm itself also eliminates an important memory limitation for multi-dimensional data remapping on sequential, distributed memory and cluster SMP computer architectures while improving performance significantly at same time.

Acknowledgments. This work is supported by Office of Biological and Environmental Research, Climate Change Prediction Program, and Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

REFERENCES

[1] OpenMP: Simple, Portable, Scalable SMP Programming. http://www.openmp.org

[2] COMPunity - The Community for OpenMP Users. http://www.comunity.org

[3] L. SMITH AND M. BULL, *Development of hybrid mode MPI/OpenMP applications*, Scientific Programming, Vol. 9, No 2-3, 83-98, 2001.

[4] F. CAPPELLO AND D. ETIEMBLE, *MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks*, Proceedings of Supercomputing 2000, Dallas, Texas, Nov 4-10, 2000.

[5] P. LANUCARA AND S. ROVIDA, *Conjugate-Gradient Algorithms: An MPI Open-MP Implementation on Distributed Shared Memory Systems*, EWOMP 1999, Lund University, Sweden, Sept.30-Oct.1, 1999.

[6] A. KNEER, *Industrial Hybrid OpenMP/MPI CFD application for Practical Use in Free-surface Flow Calculations*, WOMPAT2000: Workshop on OpenMP Applications and Tools, San Diego, July 6-7, 2000.

[7] D. S. HENTY, *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling*, Proceedings of Supercomputing 2000, Dallas, Texas, Nov 4-10, 2000.

[8] 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.

[9] 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.

[10] 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 1999, Nov 1999.

[11] A. MAROWKA, Z. LIU, AND B. CHAPMAN, *OpenMP-Oriented Applications for Distributed Shared Memory Architectures*, to appear in Concurrency and Computation: Practice and Experience, Feb. 2003.

[12] C. H. Q. DING, *An Optimal Index Reshuffle Algorithm for Multidimensional Arrays and Its Applications for Parallel Architectures*, IEEE Transactions on Parallel and Distributed Systems, V.12, No.3, pp.306-315, 2001.

[13] Y. HE AND C. H.Q. DING, *MPI and OpenMP Paradiagms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transpose*, Proceedings of Supercomputing 2002, Baltimore, Maryland, Nov 15-19, 2002.

[14] 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.

[15] S. H. BOKHARI, *Complete Exchange on the Intel iPSC-860 hypercube*, Technical Report 91-4, ICASE, 1991.