MPI and OpenMP Paradigms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition

Yun (Helen) He and Chris Ding

Lawrence Berkeley National Laboratory

Outline

- Introduction
  - Background
  - 2-array transpose method
  - In-place vacancy tracking method
  - Performance on single CPU
- Parallelization of Vacancy Tracking Method
  - Pure OpenMP
  - Pure MPI
  - Hybrid MPI/OpenMP
- Performance
  - Scheduling for pure OpenMP
  - Pure MPI and pure OpenMP within one node
  - Pure MPI and Hybrid MPI/OpenMP across nodes
- Conclusions

Background

- Mixed MPI/openMP is software trend for SMP architectures
  - Elegant in concept and architecture
  - Negative experiences: NAS, CG, PS, indicate pure MPI outperforms mixed MPI/openMP
- Array transpose on distributed memory architectures equals the remapping of problem subdomains
  - Used in many scientific and engineering applications
  - Climate model: longitude local <=> height local

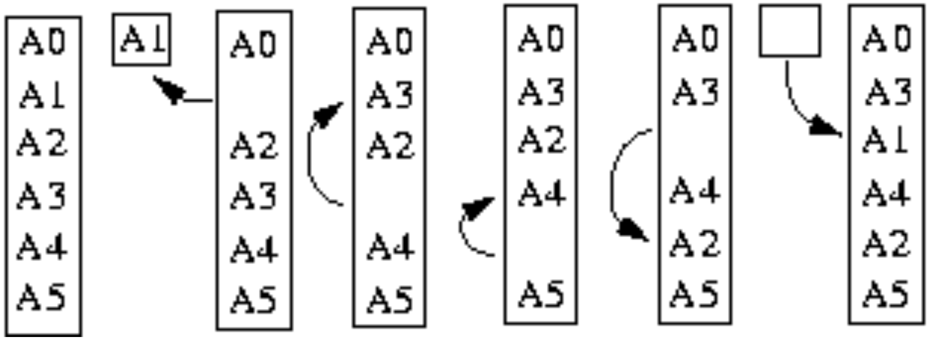
Two-Array Transpose Method

- Reshuffle Phase
  - $B[k_1,k_3,k_2] \Leftarrow A[k_1,k_2,k_3]$
  - Use auxiliary array *B*
- Copy Back Phase
  - $A \Leftarrow B$
- **Combine Effect**
  - $A'[k_1,k_3,k_2] \Leftarrow A[k_1,k_2,k_3]$

Vacancy Tracking Method

A(3,2) ➔ *A(2,3)*

Tracking cycle: 1 – 3 – 4 – 2 - 1



A(2,3,4) ➔ *A(3,4,2)*, tracking cycles:

1 - 4 - 16 - 18 - 3 - 12 - 2 - 8 - 9 - 13 - 6 - 1

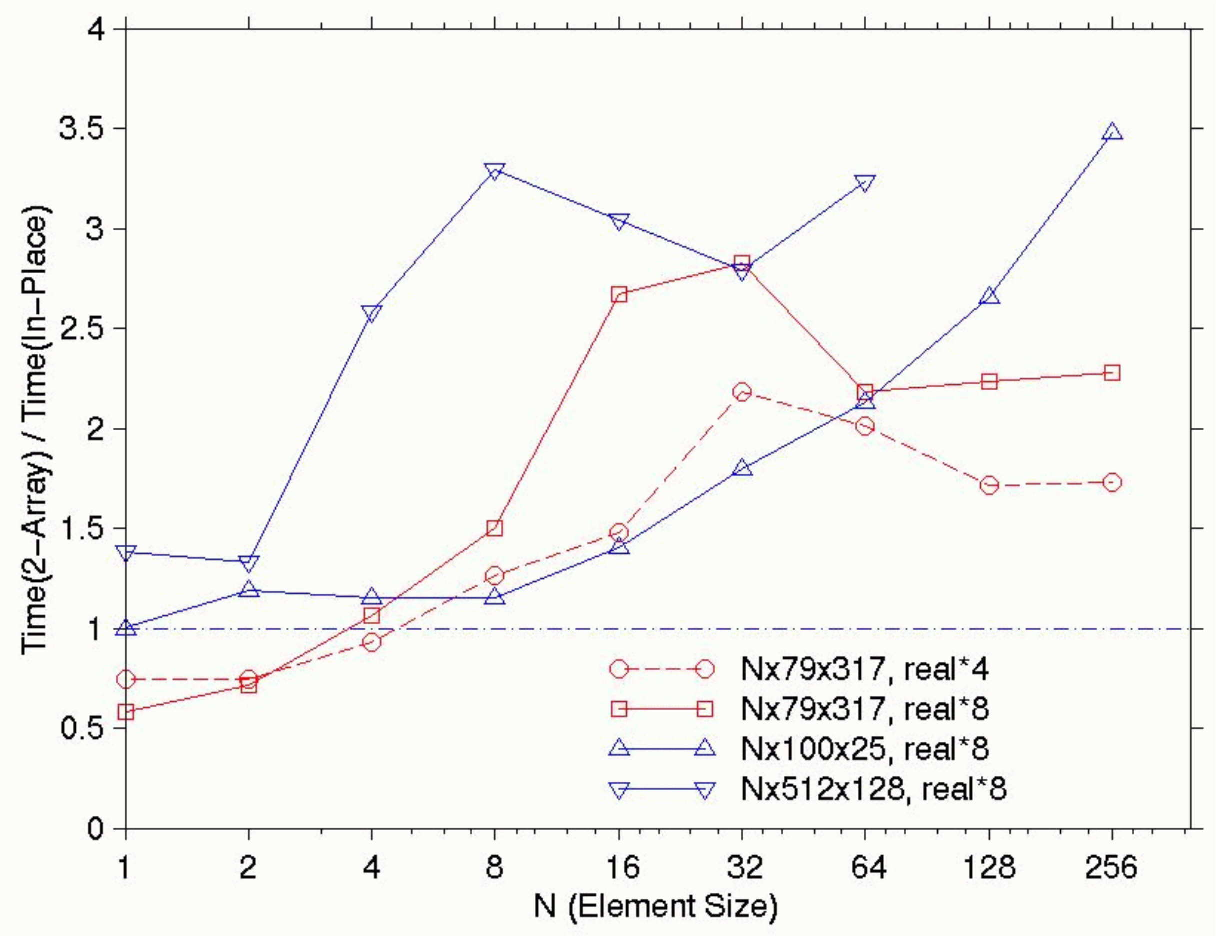
5 - 20 - 11 - 21 - 15 - 14 - 10 - 17 - 22 - 19 - 7 – 5

Cycles are closed, non-overlapping.

Algorithm to Generate Tracking Cycles

```
! 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
```

In-Place *vs.* Two-Array



Memory Access Volume and Pattern

- Eliminates auxiliary array and copy-back phase, reduces memory access in half.
- Has less memory access due to length-1 cycles not touched.
- Has more irregular memory access pattern than traditional method, but gap becomes smaller when size of move is larger than cache-line size.
- Same as 2-array method: inefficient memory access due to large stride.

Outline

- Introduction
  - Background
  - 2-array transpose method
  - In-place vacancy tracking method
  - Performance on single CPU
- Parallelization of Vacancy Tracking Method
  - Pure OpenMP
  - Pure MPI
  - Hybrid MPI/OpenMP
- Performance
  - Scheduling for pure OpenMP
  - Pure MPI and pure OpenMP within one node
  - Pure MPI and Hybrid MPI/OpenMP across nodes
- Conclusions

Multi-Threaded Parallelism

Key: Independence of tracking cycles.

```
!$OMP PARALLEL DO DEFAULT (PRIVATE)
!$OMP&        SHARED (N_cycles, info_table, Array)      (C.2)
!$OMP&        SCHEDULE (AFFINITY)
      do k = 1, N_cycles
         an inner loop of memory exchange for each cycle using info_table
      enddo
!$OMP END PARALLEL DO
```

Pure MPI

$A(N_1,N_2,N_3) \rightarrow A(N_1,N_3,N_2)$ on P processors:

(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

$A(N_1,N_3/P,N_2)$, viewed as $A(N_1N_3/P,N_2/P,P)$

$\rightarrow A(N_1N_3/P,P,N_2/P)$, viewed as $A(N_1,N_3,N_2/P)$.

Global all-to-all Exchange

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

Total Transpose Time (Pure MPI)

Use "latency+ message-size / bandwidth" model

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

where P --- total number of CPUs

M --- average memory access time per element

L --- communication latency

B --- communication bandwidth

Total Transpose Time (Hybrid MPI/OpenMP)

Parallelize local transposes (G1) and (G3) with OpenMP

N_CPU = N_MPI * N_threads

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

where N_{CPU} --- total number of CPUs

N_{MPI} --- number of MPI tasks

Outline

- Introduction
  - Background
  - 2-array transpose method
  - In-place vacancy tracking method
  - Performance on single CPU
- Parallelization of Vacancy Tracking Method
  - Pure OpenMP
  - Pure MPI
  - Hybrid MPI/OpenMP
- Performance
  - Scheduling for pure OpenMP
  - Pure MPI and pure OpenMP within one node
  - Pure MPI and Hybrid MPI/OpenMP across nodes
- Conclusions

Scheduling for OpenMP

- **Static**: Loops are divided into *n_thrds* partitions, each containing *ceiling(n_iters/n_thrds)* iterations.
- **Affinity**: Loops are divided into *n_thrds* partitions, each containing *ceiling(n_iters/n_thrds)* iterations. Then each partition is subdivided into chunks containing *ceiling(n_left_iters_in_partion/2)* iterations.
- **Guided**: Loops are divided into progressively smaller chunks until the chunk size is 1. The first chunk contains *ceiling(n_iter/n_thrds)* iterations. Subsequent chunk contains *ceiling(n_left_iters /n_thrds)* iterations.
- **Dynamic, n**: Loops are divided into chunks containing *n* iterations. We choose different chunk sizes.

Scheduling for OpenMP within one Node

Table 1: 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 (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

64x512x128: N_cycles = 4114, cycle_lengths = 16
16x1024x256: N_cycles = 29140, cycle_lengths= 9, 3

Scheduling for OpenMP within one Node (cont' d)

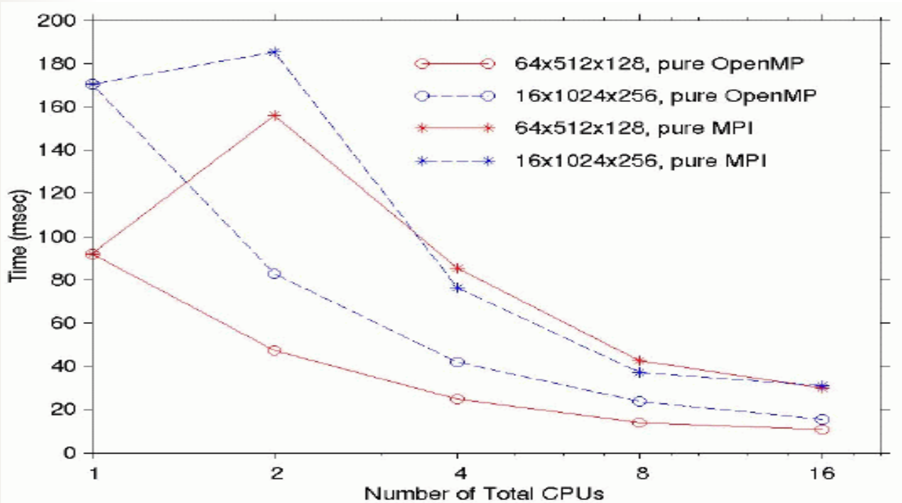
Table 2: Timing for Array Sizes 8×1000×500 and 32×100×25 with Different Schedules and Different Number of Threads Used within One IBM SP Node (Time in seconds)

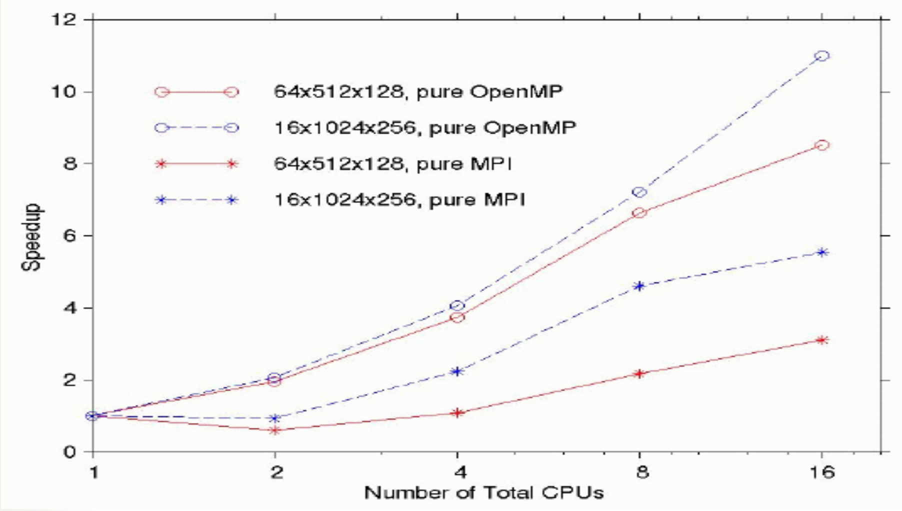
Schedule	8×1000×500					32×100×25				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	57.9	58.8	93.3	158.3	261.5	18.7	2.84	1.49	1.34	1.10
Affinity	48.5	38.0	59.3	86.3	158.0	16.6	1.88	1.72	0.95	1.31
Guided	58.0	58.4	92.7	159.9	261.4	15.3*	3.99	1.71	1.93	1.08*
Dynamic,1	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	-	-	-	-	-

8x1000x500: N_cycles = 132, cycle_lengths = 8890, 1778, 70, 14, 5

32x100x25: N_cycles = 42, cycle_lengths = 168, 24, 21, 8, 3.

Pure MPI and Pure OpenMP Within One Node

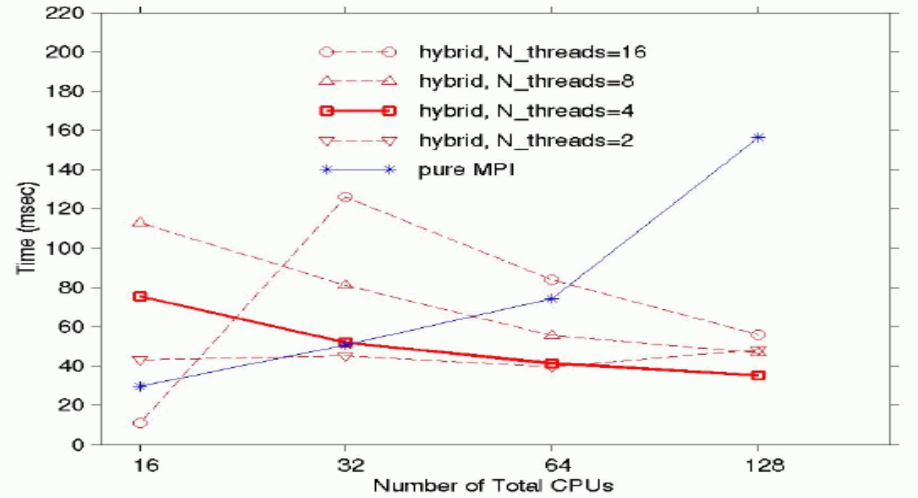


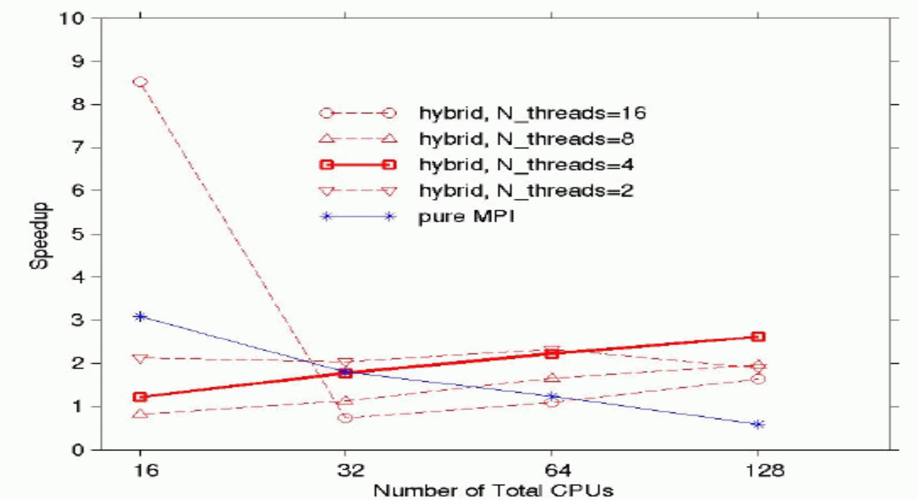


OpenMP *vs.* MPI (16 CPUs)

64x512x128: 2.76 times faster
16x1024x256:1.99 times faster

Pure MPI and Hybrid MPI/ OpenMP Across Nodes





With 128 CPUs, n_thrds=4 hybrid MPI/OpenMP performs faster than n_thrds=16 hybrid by a factor of 1.59, and faster than pure MPI by a factor of 4.44.

Conclusions

- In-place vacancy tracking method outperforms 2-array method. It could be explained by the elimination of copy back and memory access volume and pattern.
- Independency and non-overlapping of tracking cycles allow multi-threaded parallelization.
- SMP schedule *affinity* optimizes performances for larger number of cycles and small cycle lengths. Schedule *dynamic* for smaller number of cycles and larger or uneven cycle lengths.
- The algorithm could be parallelized using pure MPI with the combination of local vacancy tracking and global exchanging.

Conclusions (cont’d)

- Pure OpenMP performs more than twice faster than pure MPI within one node. It makes sense to develop a hybrid MPI/OpenMP algorithm.
- Hybrid approach parallelizes the local transposes with OpenMP, and MPI is still used for global exchange across nodes.
- Given the total number of CPUs, the number of MPI tasks and OpenMP threads need to be carefully chosen for optimal performance. In our test runs, a factor of 4 speedup is gained compared to pure MPI.
- This paper gives a positive experience of developing hybrid MPI/OpenMP parallel paradigms.