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 $\leftrightarrow$ 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

\[ A(3,2) \Rightarrow A(2,3) \]
Tracking cycle: 1 \(-\) 3 \(-\) 4 \(-\) 2 \(-\) 1

\[ A(2,3,4) \Rightarrow A(3,4,2) \], 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

![Graph showing the comparison between In-Place and Two-Array methods. The x-axis represents the element size (N), and the y-axis represents the ratio of time for Two-Array to In-Place. The graph includes multiple lines for different array sizes and data types.]
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 \textbf{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 \textbf{global all-to-all exchange} of data blocks, each of size \( N_1(N_3/P)(N_2/P) \).

(G3) Do a \textbf{local transpose} on the local array

\[ A(N_1, N_3/P, N_2), \text{ viewed as } A(N_1N_3/P, N_2/P, P) \]

\[ \Rightarrow A(N_1N_3/P, P, N_2/P), \text{ viewed as } A(N_1, N_3, N_2/P). \]
Global all-to-all Exchange

! All processors simultaneously do the following:

\[
\text{do } q = 1, P - 1 \\
\text{send a message to destination processor } \text{destID} \\
\text{receive a message from source processor } \text{srcID} \\
\text{end do}
\]

! where destID = srcID = (myID XOR q)
Total Transpose Time (Pure MPI)

Use “latency + message-size / bandwidth” model

$$T_P = \frac{2MN_1N_2N_3}{P} + 2L(P-1) + \left[\frac{2N_1N_3N_2}{BP}\right]\frac{(P-1)}{P}$$

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

\[ N_{CPU} = N_{MPI} \times N_{threads} \]

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

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 $\text{ceiling}(n_{iters}/n_{thrds})$ iterations.

- **Affinity**: Loops are divided into $n_{thrds}$ partitions, each containing $\text{ceiling}(n_{iters}/n_{thrds})$ iterations. Then each partition is subdivided into chunks containing $\text{ceiling}(n_{left_iters\_in\_partition}/2)$ iterations.

- **Guided**: Loops are divided into progressively smaller chunks until the chunk size is 1. The first chunk contains $\text{ceiling}(n_{iter}/n_{thrds})$ iterations. Subsequent chunk contains $\text{ceiling}(n_{left_iters}/n_{thrds})$ 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 64x1000x500 and 32x100x25 with Different Schedules and Different Number of Threads Used within One IBM SP Node (Time in seconds)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>64x512x128</th>
<th>16x1024x256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 thrd</td>
<td>16 thrd</td>
</tr>
<tr>
<td>Static</td>
<td>34.1</td>
<td>15.3</td>
</tr>
<tr>
<td>Affinity</td>
<td>28.8*</td>
<td>10.8*</td>
</tr>
<tr>
<td>Guided</td>
<td>35.3</td>
<td>14.2</td>
</tr>
<tr>
<td>Dynamic,1</td>
<td>32.3</td>
<td>16.5</td>
</tr>
<tr>
<td>Dynamic,2</td>
<td>32.6</td>
<td>16.1</td>
</tr>
<tr>
<td>Dynamic,4</td>
<td>33.8</td>
<td>16.7</td>
</tr>
<tr>
<td>Dynamic,6</td>
<td>32.4</td>
<td>16.1</td>
</tr>
<tr>
<td>Dynamic,16</td>
<td>30.3</td>
<td>16.0</td>
</tr>
<tr>
<td>Dynamic,32</td>
<td>28.9</td>
<td>16.2</td>
</tr>
<tr>
<td>Dynamic,64</td>
<td>34.9</td>
<td>16.0</td>
</tr>
<tr>
<td>Dynamic,128</td>
<td>28.9</td>
<td>16.1</td>
</tr>
<tr>
<td>Dynamic,256</td>
<td>29.5</td>
<td>16.0</td>
</tr>
<tr>
<td>Dynamic,512</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic,1024</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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 8x1000x500 and 32x100x25 with Different Schedules and Different Number of Threads Used within One IBM SP Node (Time in seconds)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>8x1000x500</th>
<th></th>
<th></th>
<th></th>
<th>32x100x25</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 thrd</td>
<td>16 thrd</td>
<td>8 thrd</td>
<td>4 thrd</td>
<td>2 thrd</td>
<td>32 thrd</td>
<td>16 thrd</td>
<td>8 thrd</td>
</tr>
<tr>
<td>Static</td>
<td>57.9</td>
<td>58.8</td>
<td>93.3</td>
<td>158.3</td>
<td>261.5</td>
<td>18.7</td>
<td>2.84</td>
<td>1.49</td>
</tr>
<tr>
<td>Affinity</td>
<td>48.5</td>
<td>38.0</td>
<td>59.3</td>
<td>86.3</td>
<td>158.0</td>
<td>16.6</td>
<td>1.88</td>
<td>1.72</td>
</tr>
<tr>
<td>Guided</td>
<td>58.0</td>
<td>58.4</td>
<td>92.7</td>
<td>159.9</td>
<td>261.4</td>
<td>15.3*</td>
<td>3.99</td>
<td>1.71</td>
</tr>
<tr>
<td>Dynamic, 1</td>
<td>47.1*</td>
<td>32.7*</td>
<td>49.7*</td>
<td>84.0</td>
<td>147.2</td>
<td>19.3</td>
<td>0.81*</td>
<td>1.05*</td>
</tr>
<tr>
<td>Dynamic, 2</td>
<td>50.8</td>
<td>32.7*</td>
<td>51.9</td>
<td>82.5*</td>
<td>145.8</td>
<td>17.0</td>
<td>2.68</td>
<td>1.12</td>
</tr>
<tr>
<td>Dynamic, 4</td>
<td>56.9</td>
<td>37.8</td>
<td>53.9</td>
<td>83.7</td>
<td>144.3</td>
<td>17.0</td>
<td>3.45</td>
<td>2.03</td>
</tr>
<tr>
<td>Dynamic, 8</td>
<td>63.3</td>
<td>52.7</td>
<td>52.3</td>
<td>82.5*</td>
<td>144.1*</td>
<td>16.5</td>
<td>3.29</td>
<td>2.68</td>
</tr>
<tr>
<td>Dynamic, 16</td>
<td>107.9</td>
<td>92.1</td>
<td>92.2</td>
<td>90.2</td>
<td>148.6</td>
<td>18.7</td>
<td>4.28</td>
<td>3.20</td>
</tr>
<tr>
<td>Dynamic, 32</td>
<td>165.1</td>
<td>159.4</td>
<td>158.8</td>
<td>187.8</td>
<td>155.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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 \textit{affinity} optimizes performances for larger number of cycles and small cycle lengths. Schedule \textit{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.