Parallel Transposition of Sparse Data Structures

Hao Wang†, Weifeng Liu§‡, Kaixi Hou†, Wu-chun Feng†

†Department of Computer Science, Virginia Tech
§Niels Bohr Institute, University of Copenhagen
‡Scientific Computing Department, STFC Rutherford Appleton Laboratory
### Sparse Matrix

- If most elements in a matrix are zeros, we can use sparse representations to store the matrix.

**Example Matrix:**

\[
A = \begin{bmatrix}
   0 & a & 0 & b & 0 & 0 \\
   c & d & e & f & 0 & 0 \\
   0 & o & 0 & g & h & i \\
   j & k & l & m & n & p \\
\end{bmatrix}
\]

**Compressed Sparse Row (CSR):**

- **csrRowPtr:** 0, 2, 6, 10, 15
- **csrColIdx:** 1, 3, 0, 1, 2, 3, 2, 3, 4, 5, 1, 2, 3, 4, 5
- **csrVal:** a, b, c, d, e, f, g, h, i, j, k, l, m, n, p

**Compressed Sparse Column (CSC):**

- **cscColPtr:** 0, 1, 4, 7, 11, 13, 15
- **cscRowIdx:** 1, 0, 1, 3, 1, 2, 3, 0, 1, 2, 3, 2, 3, 2, 3
- **cscVal:** c, a, d, k, e, g, l, b, f, h, m, i, n, j, p
**Spare Matrix Transposition**

- Sparse matrix transposition from $A$ to $A^T$ is an indispensable building block for higher-level algorithms

$$A = \begin{bmatrix}
  0 & a & 0 & b & 0 & 0 \\
  c & d & e & f & 0 & 0 \\
  0 & 0 & g & h & i & j \\
  0 & k & l & m & n & p
\end{bmatrix} \quad \rightarrow \quad A^T = \begin{bmatrix}
  0 & c & 0 & 0 \\
  a & d & 0 & k \\
  0 & e & g & l \\
  b & f & h & m \\
  0 & 0 & i & n \\
  0 & 0 & j & p
\end{bmatrix}$$

- CSR to CSC transformation:
  - csrRowPtr = [0, 2, 6, 10, 15]
  - csrColIdx = [1, 3, 0, 1, 2, 3, 2, 3, 4, 5, ...]
  - csrVal = [a, b, c, d, e, f, g, h, i, j, ...]
Spare Matrix Transposition

- Sparse transposition has not received the attention like other sparse linear algebra, e.g., SpMV and SpGEMM
  - Transpose $A$ to $A^T$ once and then use $A^T$ multiple times
  - Sparse transposition is fast enough on modern architectures
  - It is not always true!
Driving Cases

- Sparse transposition is inside the main loop
  - K-truss, Simultaneous Localization and Mapping (SLAM)
- Or, may occupy significant percentage of execution time
  - Strongly Connected Components (SCC)

<table>
<thead>
<tr>
<th>Examples</th>
<th>Description</th>
<th>Class</th>
<th>Build Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-truss</td>
<td>Detect sub graphs where each edge is part of at least k-2 triangles</td>
<td>Graph algorithm</td>
<td>SpMV, SpGEMM, Transposition</td>
</tr>
<tr>
<td>SLAM</td>
<td>Update an information matrix of an autonomous robot trajectory</td>
<td>Motion planning algorithm</td>
<td>SpGEMM, Transposition</td>
</tr>
<tr>
<td>SCC</td>
<td>Detect components where every vertex is reachable from every other vertex</td>
<td>Graph algorithm</td>
<td>Transposition, Set operations</td>
</tr>
</tbody>
</table>

Motivation

- **SpTrans and SpGEMM from Intel MKL Sparse BLAS**
  - SpGEMM no transposition: $C_1 = AA$
  - SpGEMM\_T (with implicit transposition\*): $y_2 = \text{trans}(A)A$
  - SpGEMM\_T (with explicit transposition): $B = \text{trans}(A)$ then $C_2 = BA$

Observations:
1. SpTrans and SpGEMM (implicit) did not scale very well
2. Time spending on SpTrans was close to SpGEMM if multiple cores were used

Implicit transposition can use $A$ as an input, but with a hint let higher-level computations operate on $A^T$. Supported by Intel.
Outlines

• Background
• Motivations
• **Existing Methods**
  – Atomic-based
  – Sorting-based
• Designs
  – ScanTrans
  – MergeTrans
• Experimental Results
• Conclusions
Atomic-based Transposition

1. Calculate the offset of each nonzero element in its column, set offset in auxiliary array \( \text{loc} \), and count how many nonzero elements in each column
   - Atomic operation \( \text{fetch}_\text{and}_\text{add}() \)
2. Use prefix-sum to count the start pointer for each column, i.e., \( \text{cscColPtr} \)
3. Scan CSR again to get the position of each nonzero element in \( \text{cscRowIdx} \) and \( \text{cscVal} \), and move it
4. An additional step, i.e., segmented sort, may be required to guarantee the order in each column
   - Offset of ‘\( f \)’ can be 0, 1, 2, 3, and final position of ‘\( f \)’ can be 7, 8, 9, 10
Sorting-based Transposition (First Two Steps)

1. Use key-value sort to sort csrColIdx (key) and auxiliary positions (value)

\[
A = \begin{bmatrix}
0 & a & 0 & b & 0 & 0 \\
c & d & e & f & 0 & 0 \\
0 & 0 & g & h & i & j \\
0 & k & l & m & n & p 
\end{bmatrix}
\]

csrColIdx = [1 3 0 1 2 3 2 3 4 5 1 2 3 4 5]

auxPos = [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]

 csrColIdx = [0 1 1 1 2 2 2 3 3 3 3 4 4 5 5]
 auxPos = [2 0 3 10 4 6 11 1 5 7 12 8 13 9 14]

↓ key-value sort

2. Set cscVal based on auxPos:

\[\text{cscVal}[x] = \text{csrVal}[%\text{auxPos}[x]]\]

For \(x = 7\), cscVal[7] = ?

auxPos[7] = 1

csrVal[1] = b

cscVal = [c a d k e g l b f h m i n j p]
Constraints in Existing Methods

• Atomic-based sparse transposition
  – Contention from the atomic operation `fetch_and_add()`
  – Additional overhead coming from the segmented sort

• Sorting-based sparse transposition
  – Performance degradation when the number of nonzero elements increases, due to $O(nnz \times \log(nnz))$ complexity
Outlines

• Background
• Motivations
• Existing Methods
  – Atomic-based
  – Sorting-based
• Designs
  – ScanTrans
  – MergeTrans
• Experimental Results
• Conclusions
Performance Considerations

• Sparsity independent
  – Performance should not be affected by load imbalance, especially for power-law graphs

• Avoid atomic operation
  – Avoid the contention of atomic operation
  – Avoid the additional stage of sorting indices inside a row/column

• Linear complexity
  – The serial method of sparse transposition has the linear complexity $O(m + n + nnz)$
  – Design parallel methods to achieve closer to it
Preprocess: extend csrRowPtr to csrRowIdx; partition csrVal evenly for threads

1. Histogram: count numbers of column indices per thread independently
2. Vertical scan (on inter)
3. Horizontal scan (on last row of inter)
A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix}

csrRowPtr = \begin{bmatrix} 0 & 2 & 6 & 10 & 15 \end{bmatrix}

csrColIdx = \begin{bmatrix} 1 & 3 & 0 & 1 & 2 & 3 & 2 & 3 & 4 & 5 \end{bmatrix}

csrVal = \begin{bmatrix} a & b & c & d & e & f & g & \textcolor{red}{h} & i & j & k & l & m & n & p \end{bmatrix}

csrRowIdx = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \end{bmatrix}

4. Write back

off = cscColPtr[\text{colIdx}] + inter[\text{tid}*n+\text{colIdx}] + intra[\text{pos}]  
csrVal = 0  
Analysis of ScanTrans

• Pros
  – Two round of Scan on auxiliary arrays to avoid atomic operation
  – Scan operations can be implemented by using SIMD operations

• Cons
  – Write back step has random memory access on cscVal and cscRowIdx
Preprocess: partition nonzero elements to multiple blocks

1. Each thread transpose one or several blocks to CSC format

2. Merge multiple blocks in parallel until one block left

① csr2csc_block

② multiple round merge
How to Mitigate Random Memory Access

<table>
<thead>
<tr>
<th>merge_pcsc_mthread</th>
<th>csc0</th>
<th>csc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscColPtr</td>
<td>0 1 3 5 8 8 8</td>
<td>0 0 1 2 3 5 7</td>
</tr>
<tr>
<td>cscRowIdx</td>
<td>1 0 1 1 2 0 1 2</td>
<td>3 3 3 2 3 2 3 3</td>
</tr>
<tr>
<td>cscVal</td>
<td>cadegbfh</td>
<td>klmijnjp</td>
</tr>
</tbody>
</table>

Merge two csc to one csc
1. Add two cscColPtr directly to get the output cscColPtr
2. For each column of output csc, check where the nonzero elements come from; and then move nonzero elements (cscVal and cscRowIdx) from input csc to output csc
   - Opt: only if two successive columns in both input csc change, we move the data
Analysis of MergeTrans

• Pros
  – Successive memory access on both input csc and output csc

• Cons
  – Performance is affected by the number of blocks
  – May have much larger auxiliary data \((2 \times nblocks \times (n + 1) + nnz)\) than ScanTrans and existing methods
Implementations and Optimizations

• SIMD Parallel Prefix-sum
  – Implement prefix-sum on x86-based platforms with Intrinsics
  – Support AVX2, and IMCI/AVX512
  – Apply on atomic-based method and ScanTrans

• SIMD Parallel Sort
  – Implement bitonic sort and mergesort on x86-based platform[4]
  – Support AVX, AVX2, and IMCI/AVX512
  – Apply on sorting-based method

• Dynamic Scheduling
  – Use OpenMP tasking (since OpenMP 3.0)
  – Apply on sorting-based method and MergeTrans

Outlines

• Background
• Motivations
• Existing Methods
  – Atomic-based
  – Sorting-based
• Designs
  – ScanTrans
  – MergeTrans
• Experimental Results
• Conclusions
Evaluation & Discussion

- Experimental Setup (Hardware)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CPU</th>
<th>MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Name</td>
<td>Intel Xeon E5-2695 v3</td>
<td>Intel Xeon Phi 5110P</td>
</tr>
<tr>
<td>Code Name</td>
<td>Haswell</td>
<td>Knights Corner</td>
</tr>
<tr>
<td># of Cores</td>
<td>2x14</td>
<td>60</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>2.3 GHz</td>
<td>1.05 GHz</td>
</tr>
<tr>
<td>L1/L2/L3 Cache</td>
<td>32 KB/ 256 KB/ 35 MB</td>
<td>32 KB/ 512 KB/ -</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB DDR4</td>
<td>8 GB GDDR5</td>
</tr>
<tr>
<td>Compiler</td>
<td>icpc 15.3</td>
<td>icpc 15.3</td>
</tr>
<tr>
<td>Compiler Options</td>
<td>-xCORE-AVX2 –O3</td>
<td>-mmic -O3</td>
</tr>
<tr>
<td>Vector ISA</td>
<td>AVX2</td>
<td>IMCI</td>
</tr>
</tbody>
</table>
Evaluation & Discussion

• Experimental Setup (Methods)
  – Intel MKL 11.3 mkl_sparse_convert_csr()
  – Atomic-based method (from SCC implementation, SC’13[3])
  – Sorting-based method (from bitonic-sort, ICS’15[4])
  – ScanTrans
  – MergeTrans

• Dataset
  – 22 matrices: 21 unsymmetric matrices from University of Florida Sparse Matrix Collection + 1 dense matrix
  – Single precision, Double precision, Symbolic (no value)

• Benchmark Suite
  – Strongly Connected Components: SCC($A$)

ICS'16, Istanbul, June 1-3, 2016
Transposition Performance on Haswell

- Compare to Intel MKL method, ScanTrans can achieve an average of \(2.8\times\) speedup.
- On wiki-Talk, the speedup can be pushed up to \(6.2\times\) for double precision.
Transposition Performance on MIC

- Compare to Intel MKL method, MergeTrans can achieve an average of 3.4x speedup.
- On wiki-Talk, the speedup can be pushed up to 11.7x for single precision.
Higher-level Routines on Haswell

![Graph 1: Execution Time (ms) for AT + A](#)

- **MKL_MT**
- **ScanTrans**

- *economics*
- *stomach*
- *venkato1*
- *web-Google*

![Graph 2: Execution Time (ms) for AT * x (#iters=50)](#

- **MKL_MT**
- **ScanTrans**

- *economics*
- *stomach*
- *venkato1*
- *web-Google*

**AT + A**

**AT * x (#iters=50)**

**SCC(A)**

202%
Outlines

• Background
• Motivations
• Existing Methods
  – Atomic-based
  – Sorting-based
• Designs
  – ScanTrans
  – MergeTrans
• Experimental Results
• Conclusions
Conclusions

• In this paper
  – We identify the sparse transposition can be the performance bottleneck
  – We propose two sparse transposition methods: ScanTrans and MergeTrans
  – We evaluate the atomic-based, sorting-based, and Intel MKL methods with ScanTrans and MergeTrans on Intel Haswell CPU and Intel MIC
  – Compare to the vendor-supplied library, ScanTrans can achieve an average of 2.8-fold (up to 6.2-fold) speedup on CPU, and MergeTrans can deliver an average of 3.4-fold (up to 11.7-fold) speedup on MIC

Thank you!