Parallel Transposition of Sparse Data Structures

Hao Wang†, Weifeng Liu††, Kaixi Hou§, Wu-chun Feng‡
†Dept. of Computer Science, Virginia Tech, Blacksburg, VA, USA, {hwang121, kaixihou, wfeng}@vt.edu
§Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark, weifeng.liu@nbi.ku.dk
‡Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, UK

ABSTRACT
Many applications in computational sciences and social sciences exploit sparsity and connectivity of acquired data. Even though many parallel sparse primitives such as sparse matrix-vector (SpMV) multiplication have been extensively studied, some other important building blocks, e.g., parallel transposition for sparse matrices and graphs, have not received the attention they deserve.

In this paper, we first identify that the transposition operation can be a bottleneck of some fundamental sparse matrix and graph algorithms. Then, we revisit the performance and scalability of parallel transposition approaches on x86-based multi-core and many-core processors. Based on the insights obtained, we propose two new parallel transposition algorithms: ScanTrans and MergeTrans. The experimental results show that our ScanTrans method achieves an average of 2.8-fold (up to 6.2-fold) speedup over the parallel transposition in the latest vendor-supplied library on an Intel multi-core CPU platform, and the MergeTrans approach achieves on average of 3.4-fold (up to 11.7-fold) speedup on an Intel Xeon Phi many-core processor.

CCS Concepts
•Computing methodologies → Linear algebra algorithms; Parallel computing methodologies;

Keywords
Sparse Matrix; Transposition; CSR; SpMV; SpGEMM; Graph Algorithms; AVX; Intel Xeon Phi

1. INTRODUCTION
A large amount of applications in computational sciences and social sciences exploit sparsity and connectivity of acquired data. For example, finite element methods construct sparse matrices of linear systems and solve them by using direct or iterative methods [19, 40]; genomic workflows analyze the functional and connective structure of genomes assembled in various graph data structures [14, 23]. In addition, graph theory has been extensively used to analyze the sparsity of system matrices and processing them [2, 31]. Because of its importance, many sparse data structures (e.g., storage formats such as BRC [4], ACSR [3], CSR5 [34] and automatic selection tools [42]), basic linear algebra subprograms (BLAS) (e.g., sparse matrix-vector multiplication (SpMV) [8, 12, 34, 35] and sparse matrix-matrix multiplication (SpGEMM) [9, 10, 33, 38]), and graph algorithms (e.g., strongly connected components (SCC) [27, 37]) have been thoroughly researched on modern parallel platforms.

However, not all important sparse building blocks have received the attention they deserve. Computing the transposition of a sparse matrix or a graph is one such important building block. In particular, the parallel performance and scalability of the transposition operation have been largely ignored. As a result, some routines using transposition as a building block show degraded performance on parallel hardware. Such routines include, but are not limited to, transpose-based BLAS operations (e.g., sparse matrix-transpose-vector multiplication and sparse matrix-transpose-matrix multiplication) and transpose-based graph algorithms (e.g., finding strongly connected components). Later in this paper, we will further elaborate on the status of transposition in Section 2.2 and present some performance numbers in Figure 2.

In this paper, we present our research on the parallel transposition of sparse data structures, in particular, the mostly used compressed sparse row/column (CSR/CSC) formats. We first revisit the serial transposition scheme and two basic parallel transposition approaches: the atomic-based method and the sorting-based method. To overcome their shortcomings, we propose two new parallel transposition algorithms: ScanTrans and MergeTrans. We also implement the above four parallel methods on x86-based multi- and many-core systems with the latest parallel techniques. We evaluate these methods and compare with the multi-threaded sparse matrix transposition from the Intel MKL library on the Intel Haswell multi-core CPU and the Intel Xeon Phi many-core processor.

Our experimental results demonstrate that in all five aforementioned methods, ScanTrans has the best performance on Haswell, while MergeTrans has the best performance on Xeon Phi. Compared to the counterpart from the Intel MKL library, our ScanTrans method achieves up to 5.6-fold speedup and 6.2-fold speedup on Haswell for input data in single precision and double precision, respectively; while on Xeon Phi, our MergeTrans method achieves up to 11.7-fold speedup and 9.9-fold speedup, respectively.
This paper makes the following contributions:

- We identify that transposition operation can be a bottleneck of some basic sparse matrix and graph algorithms.
- We propose two new parallel transposition algorithms: ScanTrans and MergeTrans.
- We implement and optimize all known parallel transposition algorithms both on multi-core and on many-core processors.
- We conduct thorough experiments to demonstrate that the proposed methods can significantly improve the performance of parallel transposition on two very different x86 platforms, and the faster transposition can also largely accelerate higher-level algorithms.

The remainder of this paper begins by describing the CSR and CSC formats and analyzing the performance issues of existing sparse transposition method on modern multi-core processors in Section 2. Next, we present the algorithm of serial sparse transposition and two parallel algorithms: atomic-based and sorting-based in Section 3. We propose ScanTrans and MergeTrans in Section 4 as well as present the parallel implementation details in Section 5. We conduct the evaluations in Section 6 and discuss the relevant related work in Section 7. We conclude by summarizing this paper and arguing that higher-level routines should consider sparse transposition as a first-class concern in Section 8.

2. BACKGROUND AND MOTIVATION

2.1 Preliminaries

The compressed sparse row (CSR) and compressed sparse column (CSC) representations are the most widely used storage formats for sparse matrix and graph data. The CSR format consists of three arrays: (1) csrRowPtr stores the starting and ending pointers of nonzero elements of the rows. Its length is \( m + 1 \), where \( m \) is the number of the rows of the sparse matrix. (2) csrColIdx stores the column indices of nonzero elements. Its length is \( nnz \), where \( nnz \) is the number of nonzero elements in the sparse matrix. (3) csrVal array stores \( nnz \) values of the nonzero elements. The number of nonzero elements in a row \( i \), denoted as \( nnz_i \), can be computed as \( \text{csrRowPtr}[i + 1] - \text{csrRowPtr}[i] \). The column indices of nonzero elements in this row can be computed as \( \text{csrColIdx}[\text{csrRowPtr}[i]] \ldots \text{csrColIdx}[\text{csrRowPtr}[i + 1] - 1] \). Similar to the CSR format, the CSC format also uses three arrays, cscColPtr, cscRowIdx, and cscVal, for storing the starting and ending pointers of nonzero elements of the columns, the row indices of nonzero elements, and the corresponding values, respectively.

Matrix transposition transforms the \( m \times n \) matrix \( A \) to the \( n \times m \) matrix \( A^T \). For a matrix, the transposition is to transform the row-major storage to the column-major storage. Therefore, the CSC representation of a matrix is actually equivalent to the CSR representation of its transpose. Figure 1 illustrates the two formats for sparse matrices \( A \) of size \( 4 \times 6 \) and its transpose \( A^T \) of size \( 6 \times 4 \). We can see the equivalency in the CSR and CSC formats from both sides (Figures 1a and 1b). For brevity, in this paper, we use the statement “from CSR to CSC” to denote the transposition operation.

Because graphs can be represented by adjacency matrices and vice versa [16], they can share the same low-level algorithms [11]. Therefore, in this paper, we use the word “transposition” to denote sparse matrix transposition or graph transposition, graph conversion, or graph reversion.

2.2 Motivation

Many higher-level graph and linear algebra algorithms use sparse matrix transposition as a building block in their preprocessing and processing stages. While considering a scenario consuming a fixed graph or sparse matrix, the overhead of preprocessing (including matrix transposition) may be amortized within subsequent tens or hundreds of iterations. Some iterative linear system solvers, such as conjugate gradient (BiCG) [20] and standard quasi-minimal residual (QMR) [22], computing sparse matrix-vector multiplication (SpMV), both on \( A \) and \( A^T \), are examples of this scenario. Moreover, Buluç and Gilbert [10] utilized transposition as a part of preprocessing for fast sparse matrix-matrix multiplication (SpGEMM).

However, some other algorithms and applications do not have an iteration phase or have to process a changed sparse matrix in each iteration. For example, finding a graph’s strongly connected components (SCCs) [44] depends on its sparsity structure, thus merely runs once for an input. Some recently proposed fast SCC algorithms, such as FW-BW-Trim methods [27, 37], show good scalability, but require both original graph \( A \) and its transpose \( A^T \) as input. In this case, the transposition operation can be a bottleneck of detecting SCCs, if it scales not so well.

Another example is the simultaneous localization and mapping (SLAM) problem [32], which is one of the most important approaches to enable an autonomous robot to explore, map, and navigate in previously unknown environments. The method acquires and analyzes a new information matrix (sparse) in each step of a whole robot trajectory, which is commonly a long process that includes a large number of steps. In an efficient SLAM implementation proposed by Dellert and Kaess [17], the authors pointed out that com-
putting $A^T A$ (where $A$ is the information matrix constructed in each step) is about six times more expensive than QR factorization, thus dominating the overhead of SLAM. When SpGEMM scales well by running the latest fast methods [9, 38], computing $A^T A$ will limit the overall efficiency. (Note that SLAM demands as fast as possible, even real-time, data processing.)

To further demonstrate the motivation of this work, we benchmark the most used level 2 and level 3 sparse BLAS routines, SpMV and SpGEMM, in both non-transpose and transpose patterns and see how transposition influences the performance. For the SpMV operation, we evaluate three patterns: (1) ordinary SpMV $y_1 ← Ax$, (2) SpMV, T with implicit transposition $y_2 ← A^T x$, and (3) SpMV, T with an explicit transposition stage i.e., $B ← A^T$ then $y_2 ← Bx$. We also run the same three patterns for the SpGEMM operations.

Figure 2 shows the latest Intel MKL sparse BLAS performance of matrix web-Google, downloadable from the University of Florida Sparse Matrix Collection [16]. Note that the SpMV operations are iterated 50 times to simulate a real scenario in sparse iterative solvers such as BiCG. We can see that ordinary SpMV and SpGEMM operations scale well on the used 2×14-core Intel Xeon E5-2695 v3. However, the transposition operation itself does not show good scalability. Similarly, the implicit transposition versions of SpMV and SpGEMM do not scale well. For example, the implicit SpMV, T operation stops scaling when more than eight cores are used. However, if the matrix is explicitly transposed in advance, both SpMV, T and SpGEMM, T scale as well as the ordinary versions.

On the other hand, we can see that when more cores are utilized, the cost of the transposition operation is not negligible. Taking the SpGEMM operation as an example, a serial transposition uses 620 ms when a serial SpGEMM uses 2756 ms. However, when all 28 cores are utilized, the transposition still needs 139 ms but the SpGEMM requires only 260 ms. Consequently, if the transposition operation can scale better, the performance of SpMV, T and SpGEMM, T can be naturally improved.

## 3. EXISTING METHODS

### 3.1 Serial Transposition

Serial sparse matrix transposition is straightforward to implement. For a matrix $A$ of size $m × n$, the method uses the cscColPtr array to count the number of nonzero elements in each column. After that, the algorithm uses exclusive scan (i.e., prefix sum) on the histogram to set the starting and ending pointers in cscColPtr. Then, all nonzero elements are traversed again and moved to their final positions calculated by the column offsets in cscColPtr and the current relative positions in curr. Algorithm 1 illustrates the serial implementation.

### 3.2 Atomic-based Transposition

The serial algorithm can be parallelized with multithread programming models that support atomic operations on modern processors. In Algorithm 2, we show the implementation when applying OpenMP directives. In contrast to the serial algorithm, we allocate the array dloc of size $mn$ to record the relative position for each nonzero element in corresponding column. We can see that multiple threads may update the same relative position stored in dloc. Therefore, in the first for loop (that counts the number of nonzero elements in each column), we use the atomic operation fetch_and_add for updating dloc. As a result, all threads obtain conflict-free relative positions and can safely run the rest calculations.

Even though atomic operations have been used in some graph algorithms [6] to support parallelism, there still remains several potential performance bottlenecks. The first one is that atomic operations are inherently serial when multiple threads try to update the same memory address. In this case, the atomic operations actually degrade overall performance. Second, the atomic operation fetch_and_add cannot guarantee the relative order of nonzero elements in each column. For some algorithms that strictly require the order in each column [15], a compensation using the key-value sort [41] to reorder cscRowIdx and cscVal in each column is necessary. This leads to additional overhead.

### 3.3 Sorting-based Transposition

A sorting-based serial transposition method was first proposed by Gustavson [26]. This approach sorts column indices (i.e., the array cscColIdx in the CSR format) to gather all nonzero entries with the same column index to contiguously.
Algorithm 2: Atomic-based Parallel Sparse Transposition

```c
int csr2csc_atomic(m, n, nnz, csrRowPtr, csrColIdx, csrVal, cscColPtr, cscRowIdx, cscVal)
{
    // construct an array of size nnz to record the relative position of a nonzero element in corresponding column
    dloc = new int[nnz];
    #pragma omp parallel for schedule(dynamic)
    for i = 0; i < m; i++ do
        for j = csrRowPtr[i]; j < csrRowPtr[i+1]; j++ do
            dloc[j] = fetch_and_add(&cscColPtr[cscColIdx[j] + 1]), 1;
        prefix_sum(cscColPtr, n + 1);
    #pragma omp parallel for schedule(dynamic)
    for i = 0; i < m; i++ do
        for j = csrRowPtr[i]; j < csrRowPtr[i+1]; j++ do
            loc = cscColPtr[cscColIdx[j]] + dloc[j];
            cscRowIdx[loc] = i;
            cscVal[loc] = csrVal[j];
        sort cscRowIdx, cscVal in each column
    #pragma omp parallel for schedule(dynamic)
    for i = 0; i < n; i++ do
        begin = cscColPtr[i];
        end = cscColPtr[i+1];
        sort_keys_values(being, end, cscRowIdx, cscVal);
        delete dloc;
        return;
}
```

4. PROPOSED TRANSPOSITION METHODS

4.1 Performance Considerations

To realize an efficient transposition operation, we have to consider three aspects. First, the algorithm is required to be sparsity independent. That is to say, the nonzero entries should be evenly divided to threads. Thus transposing irregular data, such as power-law graphs, will not be affected by load imbalance (e.g., some of their very long rows are assigned to one single core and short ones to the others).

Second, because the atomic operations may serialize operations expected to be parallel, they should be avoided despite their simplicity. If required, some auxiliary arrays can be allocated to eliminate race conditions. Also, the relative order of nonzero entries should be guaranteed naturally to avoid an extra stage of sorting indices inside a row/column.

Third, the work complexity should be lower than $O(nnz \log nnz)$ of the sorting-based method. Consider the serial method has linear complexity $O(m + n + nnz)$, it may be possible to design parallel methods to achieve somewhere closer to it.

4.2 ScanTrans

We first introduce the scan-based sparse matrix transposition called ScanTrans. The basic idea of this algorithm is to partition nonzero elements evenly among threads, count the numbers of column indices as well as the relative position of each nonzero element in corresponding column by each thread, and finally get the absolute offset of a nonzero element in output cscRowIdx and cscVal after two rounds of scan. We construct two auxiliary arrays here: a two-dimensional array `inter` of size `(nthreads + 1) * n`, and the one-dimensional array `intra` of size `nnz`. Each row `i`, `i > 0`, in `inter` stores the number of column indices observed by the thread `i - 1`. Each element in `intra` is used to store the relative offset in the column of the corresponding nonzero element by the thread. After obtaining such histograms, the algorithm applies the column-wise scan, which is called `verticalScan` in Figure 3, on the `inter`, followed by the prefix sum on the last row of the `inter`. After that, the values in the last row of `inter` are the starting and ending pointers of row indices of the transposed matrix (cscColPtr). Finally, the algorithm calculates the absolute offset of each nonzero element in cscRowIdx and cscVal by adding the corresponding values in cscColPtr, `inter`, and `intra`.

Figure 3 shows the process of ScanTrans on the matrix $A$ shown in Figure 1. In the initial stage, the algorithm allocates memory for the `inter` and `intra` arrays and generates the row indices for nonzero elements in cscRowIdx. In the histogram step, each thread gets column indices from csrColIdx in its own partitions and sets the values of `inter` and `intra`. For example, the nonzero element $h$, whose column index is 3 and row index is 2, is the second nonzero element in column 3 of the partition proceeded by thread 1, where $f$ is the first element in column 3. Thus, the corresponding element in `intra` is set to 1 by thread 1. Because there are two nonzero elements, i.e., $f$ and $h$, in column 3 of this partition, the corresponding element 15 of `inter`, i.e., $(tid + 1) * n + colIdx = (1 + 1) * 6 + 3 = 15$, is set to 2. With the vertical scan on the two-dimensional array `inter`, the algorithm gets the total numbers of nonzero elements in each column, “1,3,3,4,2,2” as shown in the last row of `inter`. The vertical scan also generates the numbers of nonzero elements in each column proceeded by previous threads. For example, after the vertical scan, column 3 of `inter` is “0,1,3,3,4”, which means before thread 0, no element in column 3 is proceeded; before thread 1, there is one element in column 3 is proceeded; and so on. In the third step, the algorithm applies the prefix sum on the last row of `inter` and gets the start and end pointers of row indices in CSC (cscColPtr). After that, the algorithm can calculate the absolute offset in cscRowIdx and cscVal for any nonzero element. For example, for $h$ whose column index is 3 and position in csrColIdx is 7, the absolute offset in cscRowIdx and cscVal is $7 + 1 + 1 = 9$ calculated by the following equation:

$$off = cscColPtr[colIdx] + \text{inter}[tid * n + colIdx] + \text{intra}[pos].$$

Finally, its value and row index are written back to the position 9 of cscRowIdx and cscVal, respectively, which are
tagged by the gray color in the figure.

Compared to the atomic-based sparse matrix transposition, ScanTrans avoids the usage of atomic operations, and also keeps the order of nonzero elements in each column (thus no complementary sort stage is needed). However, this method cannot avoid the random memory access. After getting the absolute offset of a nonzero element, the method copies data from csrRowIdx and csrVal to cscRowIdx and cscVal. Even though the data read on csrRowIdx and csrVal is contiguous in each thread, having the spatial locality, the data write on cscRowIdx and cscVal is totally random. It is interesting to check whether there is improved performance if we change the sequential read, random write pattern to the random read, sequential write pattern. As a result, we implement an alternative of ScanTrans: different with Algorithm 3, instead of calculating the absolute offset in cscRowIdx and cscVal, each thread calculates the absolute position for a nonzero element in csrRowIdx and csrVal reversely. Then, the data access pattern is changed to the random read on nonzero elements in CSR, and sequential write on data in CSC.

4.3 MergeTrans

In order to mitigate the random memory access in ScanTrans and better leverage the shared LLC (Last Level Cache) in multi- and many-core processors, we design another parallel sparse matrix transposition, called MergeTrans, as shown in Figure 4. The MergeTrans method consists of two stages, the transposition stage and the merge stage. In the transposition stage, this method partitions the nonzero elements into multiple blocks (the block number is configurable), and transposes a block of nonzero elements from CSR to CSC by using the serial algorithm 3.1 or the single threaded sorting-based algorithm. For example, if we use the sorting-based method to transpose nonzero elements in the partition of thread 0, the method (1) sorts the column indices “1,0,1,3” to “0,1,1,3”; (2) accordingly moves their values from “a,b,c,d” to “c,a,d,b” in csrVal; (3) sets the row indices of nonzero elements to corresponding positions in cscRowIdx as “1,0,1,0”; and (4) based on the sorted column indices “0,1,1,3”, generates the cscColPtr “0,1,3,4,4,4” (one elements in the column 0, two elements in the column 1, and one elements in the column 3). If the size of a block is small enough, the random memory access can be mitigated because the data is expected to reside in the cache.

In the merge stage, the algorithm merges generated multiple CSC iteratively in parallel, until there is only one left. Two merge functions merge_mcsc_sthread and merge_pccs_mthread are designed to merge multiple CSC by a single thread and merge a pair of two CSC by multiple threads, respectively, as shown in Algorithm 4. In Figure 4, we set the number of blocks equal to the number of threads to simplify the figure. After getting multiple CSC, this method uses two rounds of merge_pccs_mthread to merge four CSC. The merge algorithm directly adds cscColPtr of two CSC to get the merged cscColPtr, and then moves nonzero elements in each column of cscRowIdx and cscVal to the column of the merged CSC with an interleaved manner. For example, in step 3, we use all four threads to merge two CSC, denoted as (t0, t1) and (t2, t3). The algorithm first gets the merged cscColPtr “0,1,4,7,11,13,15” by adding “0,1,3,5,8,8,8” and “0,0,1,2,3,5,7” correspondingly. The second and third elements “1,4” in the merged cscColPtr means there are three elements at the column 1. Two of them come from the CSC (t0, t1) because the corresponding elements in cscColPtr of CSC (t0, t1) are “1,3”, while one element comes from the CSC (t2, t3) because of “0,1” in the second and third positions of cscColPtr. Thus, the algorithm copies corresponding nonzero elements a, d from (t0, t1) and k from (t2, t3) as well as their row indices. In this figure, we use two colors to tag data from two CSC; and it shows the interleaved data access pattern. Furthermore, because the algorithm can move successive data, e.g., a, d, to contiguous positions of the merged CSC, such sequential read and write pattern can increase
the data locality.

In our implementation, several optimizations have been applied on. Multiple successive columns may be checked to enable the block data copy. For example, after “5,8,8,8” in cscColPtr of the CSC (t0, t1) is checked, we move all nonzero elements pointed by “2,3,5,7” in the CSC (t2, t3) instead of checking them column by column. Other optimizations, including dynamic binding of thread to task, will be discussed in Section 5.3.

Algorithm 4 illustrates the details of MergeTrans. The algorithm needs to construct auxiliary buffers to hold intermediate CSC. The total memory usage is $2 \times (nblocks \times (n + 1) + nnz)$ for cscColPtrA, cscRowIdxA, and cscValA, where ‘A’ means Auxiliary, and ‘2’ is for the double buffering mechanism because the algorithm is not the in-place transposition [43, 13]. Compared to other algorithms, MergeTrans has better memory access patterns both in the transposition and merge stage.

5. PARALLEL IMPLEMENTATION DETAILS

5.1 Parallel Prefix Sum

Prefix sum is an important building blocks used by multiple transposition methods, e.g., serial, atomic-based, and ScanTrans. However, due to the data dependency in successive elements, as shown in Line 7 of Algorithm 1, the compiler directives cannot optimize such a loop efficiently. As a result, we implement a parallel scan by following the scan-scan-add strategy [45] and vectorizing it with ISA intrinsics.

Manually programming with ISA intrinsics is non-trivial, and the code may not be portable across ISAs. For example, implementing the same functionality on Intel Haswell and Intel Xeon Phi requires 256-bit and 512-bit SIMD intrinsics, respectively. Thus, we implement a framework to automatically generate the vector codes for the prefix sum function on different ISAs. The basic idea is similar to the previous research [36, 28, 29]. We implement a template function for the prefix sum, and define the data reordering and computation patterns as building blocks of the template. The framework takes the data patterns as the input, searches corresponding ISA intrinsics, e.g., load, store, add, permutation, shuffle, etc., and selects intrinsics that can implement desired patterns to construct the building blocks. In current Intel architectures, a vector register consists multiple lanes, each of which has multiple elements, e.g., a 256-bit wide register of Intel Haswell has two lanes and each lane has four 32-bit elements. We use the intra-lane and inter-lane intrinsics to implement the desired pattern.

Figure 5 shows the data reordering and computation patterns of the prefix sum on vector registers. On the vector registers whose width is w, the in-register prefix sum needs $\log w$ steps and each step i consists of four stages: (1) intra-lane shuffle that rotates 2^i positions to left in each lane, (2) inter-lane permute that shifts lanes to left, (3) blend two vectors that will mask 2^i elements of the first vector, and (4) add two vectors. When w is 8, there are 3 steps to implement the in-register prefix sum. For the step 0, after loading 8 elements into the register v0, the stage 1 will rotate 2^0 po-
5.2 Parallel Sort

We implement the sorting-based transposition algorithm following a sort-merge manner: (1) halve data into two segments until the segment meets the predefined threshold, (2) sort each segment, and (3) merge segments with a multi-way merge manner until one segment is left. We implement each stage in parallel with multiple threads. When sorting a segment in one thread, we load data into vector registers and use the bitonic sort to sort data, because the vectorized bitonic sort has shown good performance and scalability on multi- and many-core processors in previous research [41, 30, 28]. In the merge stage, when the number of segments is larger than the number of threads, we use multiple threads to merge one pair of sorted segments. In this case, we use the mergepath method [24] to partition two segments, making each thread to merge same sized data and the merged data in each thread to be ordered among threads.

5.3 Dynamic Scheduling

In multiple transposition methods, e.g., the sorting-based method and MergeTrans method, we need to schedule threads for data chunks. The dynamic schedule directives of OpenMP are used to optimize the for loops in the atomic-based method and ScanTrans method, but they are not efficient to schedule threads for blocked tasks, e.g., the sort and merge tasks in the sorting-based method, and the transposition tasks and merge tasks in MergeTrans. For such cases, we use OpenMP tasking mechanism introduced in OpenMP 3.0 to dynamically schedule schedule threads to execute tasks. The tasking of the OpenMP runtime system has supported the dynamic task binding, even for the recursive algorithms. Better load balance is expected by using this mechanism.

Because the column indices in each row of CSR are already in order, we only need to merge rows to get the sorted column index array. However, we did not observe better performance of the merging-based method in our evaluation. The major reason is because the numbers of nonzero elements in different rows are usually not equal, this method has the imbalance problem even with the OpenMP tasking. As a result, we only show the performance numbers of the sorting-based method in Section 6.

6. EXPERIMENTS

6.1 Experimental Setup

In this paper, we evaluate five parallel transposition methods: (1) the sparse BLAS method mkl_sparce_convert_csr with the inspector-executor pattern from the latest Intel MKL 11.3, (2) the atomic-based method from the graph analysis package [27], (3) the sorting-based method optimized by using the parallel sort with SIMD [28], (4) the proposed ScanTrans method described in Section 4.2, and (5) the proposed MergeTrans method described in Section 4.3. We benchmark the above five methods on Intel Haswell (HSW) and Knights Corner (KNC), respectively. The HSW node is a dual-socket Intel Xeon E5-2695 v3 multi-core platform (2x14-core Haswell, 2.3 GHz, 128 GB ECC DDR4, 2x68.3 GB/s), and the KNC node has an Intel Xeon Phi 5110P many-core processor (60 cores, 1.05 GHz, 8 GB DDR5, 320 GB/s). All methods are compiled using Intel compiler icpc 16.1. On HSW, we use the compiler option -xCORE-AVX2 to enable AVX2. On KNC, we run the experiments using the native mode and compile the codes with -mnmic. All codes in our evaluations are optimized in the level of -O3.

To better cover various real-world scenarios, we transpose a given matrix stored in three data types: symbolic (i.e., no value thus only sparsity structure is transposed), 32-bit single precision and 64-bit double precision. But note that the Intel MKL parallel transposition routine does not support symbolic transposition thus this group of numbers are left blank.

6.2 Benchmark Suite

The University of Florida Sparse Matrix Collection [16] now includes near 3000 matrices from a variety of research fields, but most of them are symmetric. To benchmark our work described in this paper, we choose 21 unsymmetric matrices from a broad of applications such as computational fluid dynamics, communication network, economics, and DNA electrophoresis. We also add the matrix Dense to the suite to better understand performance behavior of transposition algorithms, even though it is not downloadable from the Collection and is actually symmetric. Table 1 shows the 22 matrices.

6.3 Transposition Performance

Figure 6 shows transposition performance, i.e., archived bandwidth in GB/s, of the above five methods on the multi-core CPU platform running for the 22 matrices in the three
data types. Figure 7 plots the same groups of performance numbers on the Intel Xeon Phi device.

We can see that the performance of atomic-based method highly depends on matrix sparsity structure. It can be very fast if the input matrix is relatively regular and has less write conflicts on the same memory addresses. Taking the matrix memchip as an example, the atomic-based method achieves the best performance and obtained 17.2% and 33.9% performance improvement over the best performance and obtained 17.2% and 33.9% per-

Table 1: The list of selected matrices

<table>
<thead>
<tr>
<th>Name</th>
<th>#Rows/Cols</th>
<th>#Nonzeros</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC_680k</td>
<td>682,862</td>
<td>3,871,773</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>cage14</td>
<td>1,505,785</td>
<td>27,130,349</td>
<td>Weighted directed graph</td>
</tr>
<tr>
<td>circuit5M</td>
<td>5,558,326</td>
<td>59,524,291</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>Dense</td>
<td>2,000</td>
<td>4,000,000</td>
<td>Dense matrix in CSR format</td>
</tr>
<tr>
<td>Economics</td>
<td>206,500</td>
<td>1,273,389</td>
<td>Economic</td>
</tr>
<tr>
<td>eu-2005</td>
<td>862,664</td>
<td>19,235,140</td>
<td>Directed graph</td>
</tr>
<tr>
<td>flickr</td>
<td>820,878</td>
<td>9,837,214</td>
<td>Directed graph</td>
</tr>
<tr>
<td>FullChip</td>
<td>2,987,012</td>
<td>26,621,990</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>language</td>
<td>399,130</td>
<td>1,216,334</td>
<td>Directed graph</td>
</tr>
<tr>
<td>memchip</td>
<td>2,707,524</td>
<td>14,810,202</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>para-4</td>
<td>153,226</td>
<td>5,326,228</td>
<td>Semiconductor device</td>
</tr>
<tr>
<td>rajat21</td>
<td>411,676</td>
<td>1,893,370</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>rajat29</td>
<td>643,994</td>
<td>4,866,270</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>smem3Dc</td>
<td>42,930</td>
<td>3,148,656</td>
<td>Structural problem</td>
</tr>
<tr>
<td>Stanford_Berkeley</td>
<td>683,446</td>
<td>7,583,376</td>
<td>Directed graph</td>
</tr>
<tr>
<td>stomach</td>
<td>213,360</td>
<td>3,021,648</td>
<td>2D/3D problem</td>
</tr>
<tr>
<td>torso1</td>
<td>116,158</td>
<td>8,516,500</td>
<td>2D/3D problem</td>
</tr>
<tr>
<td>transient</td>
<td>178,866</td>
<td>961,790</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>venkat01</td>
<td>62,424</td>
<td>1,717,792</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>1,000,005</td>
<td>3,105,536</td>
<td>Weighted directed graph</td>
</tr>
<tr>
<td>web-Google</td>
<td>916,428</td>
<td>5,105,039</td>
<td>Directed graph</td>
</tr>
<tr>
<td>wiki-Talk</td>
<td>2,394,385</td>
<td>5,021,410</td>
<td>Directed graph</td>
</tr>
</tbody>
</table>

Because the cache miss penalty on Intel Xeon Phi is much higher than that on multi-core CPU [39], the ScanTrans method has obvious performance degradation. Under such a circumstance, the MergeTrans method that mitigating the cache miss has become the overall winner. Compared to the counterpart from Intel MKL, the MergeTrans can achieve up to 11.7-fold and 9.9-fold speedups for the single precision and double precision both on wiki-Talk. Compared to its performance on Intel Xeon Phi, the MergeTrans performs relatively poor on Intel Haswell, as shown in Figure 6. This is because the MergeTrans uses multiple iterations to move nonzero elements from two input CSC to the merged one, and more memory accesses are needed compared to the ScanTrans. On Intel Haswell, the benefit of eliminating irregular memory access patterns and avoiding accompanying cache miss penalties in the MergeTrans is offset by extra memory access operations. This can also explain the observation on Intel Xeon Phi: when the matrices become dense, the ScanTrans will retake the leadership from the MergeTrans, as shown in smem3Dc, torso1, and dense of Figure 7.

Note that in our experiments, we tuned all five methods with different values of parameters, e.g., the thread affinity and the number of threads, and showed their best performance numbers in the figures. On Xeon Phi, the best performance could be achieved for all methods when we set the thread affinity type to balanced and the number of threads to 60. On Haswell, the thread affinity type compact could provide better performance than scatter, while the number of threads achieving the peak performance were diverse. We observed that the ScanTrans method could achieve the peak performance when using 16 - 22 cores, while the Intel MKL transposition stopped scaling after 8 - 14 cores. For example, for torso1 with single precision, the Intel MKL method scaled up to 14 cores, and the ScanTrans method could obtain 2.5-fold bandwidth when using 22 cores. Although the scalability of parallel transposition for sparse data structures is improved by the ScanTrans method, it is still an issue on multi-core processors and further optimizations are needed.

### 6.4 Transposition in Higher Level Routines

As sparse matrix transposition is normally used as a building block for higher level problems, we construct a routine suite composed of four representative graph and linear algebra algorithms requiring transposition operation: (1) sparse matrix-transpose-matrix addition, i.e., $C = A^T + A$, (2) sparse matrix-transpose-vector multiplication, i.e., $y = A^Tx$, (3) sparse matrix-transpose-matrix multiplication, i.e., $C = A^TA$, (4) strongly connected components problem, i.e., SCC(A). We briefly introduce them below.

1. The operation $A^T + A$ adds two sparse matrices $A$ and its transpose $A^T$. Applications of this fundamental operation include symmetrizing unsymmetric sparse matrices for graph partitioning [6] and sparsity preserving pre-ordering (such as the approximate minimum degree (AMD) ordering [2]) in prior to numerical factorizations.

2. The operation $y = A^Tx$ multiplies a matrix $A^T$ and a dense vector $x$. It is commonly used in iterative methods for sparse solvers such as BiCG and QMR. Because of this, we
choose a scenario requires 50 iterations and report 50 SpMV cost.

(3) The operation $A^T A$ multiplies a sparse matrix’s transpose $A^T$ and itself. Both $A^T A$ arises in sparse least-squares problems [7] (e.g., the above mentioned SLAM method [17] in robotics). Moreover, finite element assembly problem can also be implemented by calculating $A^T A$ [26].

(4) The operation SCC($A$) detects strongly connected components in a given directed graph $A$. An SCC is a maximal subgraph where there exists a path between any two vertices in the subgraph. The classic sequential SCC method designed by Tarjan [44] only works on $A$. But another classic serial algorithm from Kosaraju [1] and several efficient parallel SCC methods [27, 37] require connection information from both $A$ and $A^T$.

Figures 8 and 9 demonstrate performance improvement while using the ScanTrans and MergeTrans methods proposed in this paper. For brevity, we only select four matrices (Economics, stomach, venkat01 and web-Google) from our benchmark suite of 22 matrices. We compare the parallel transposition method in the MKL library with ScanTrans and MergeTrans methods on the CPU and Xeon Phi platforms, respectively. In Figure 2, we already show that the explicit methods (i.e., transpose in advance and conduct SpMV or SpGEMM computations) are much faster than the implicit methods. Therefore we only evaluate the explicit methods from the newest Intel MKL library.

We can see that all scenarios achieve better performance when using our methods. Taking matrix venkat01 as an example, the SCC operation obtains 202% performance improvement by our ScanTrans method on the CPU platform. On average, the four matrices achieve 26.8%, 30.7%, 26.2% and 62.7% improvement on the CPU device. On the Xeon Phi, the overall performance improvement is also noticeable.

6.5 Discussion

Symmetric Matrix: Except for the matrix Dense, all matrices used in our evaluations are unsymmetric. We did not show the performance numbers for symmetric matrices in the figures because sparse matrix algorithms usually use different solvers to handle symmetric and unsymmetric matrices, and those for symmetric matrices may process the transposition implicitly [21, 46]. Actually, in our observation, the proposed ScanTrans and MergeTrans can also provide competitive benefits for symmetric cases. For example, for the symmetric matrices delaunay_n21, hollywood-2009, and gupta3 from [16], ScanTrans can achieve 1.4-fold, 3.4-fold, and 4.0-fold speedups over the multi-threaded Intel MKL version on Haswell, respectively; and MergeTrans can deliver 5.4-fold, 3.6-fold, and 5.0-fold speedups on Xeon Phi.

Auto-Selection: Although ScanTrans and MergeTrans illustrate best performance on Intel Haswell and Xeon Phi, respectively, it is ideal to provide an adaptive method that can automatically select a proper transposition method for
more sparse matrices and hardware platforms. It is non-
trivial because the performance depends on combinations
of many configurable factors, including the characteristics
of the input sparse matrix $A$, e.g., m, n, nnz, sparsity, deviation,
etc., the targeted platforms, e.g., architecture of process-
ers, cache and memory hierarchy, etc., the parameters
in the transposition methods, e.g., predefined segment size
in the sorting-based method, configurable number of blocks
in MergeTrans, etc. We have observed many graph applica-
tions have put the sparse matrix transposition kernel inside
for loops before calling the SpMV and SpMM_T because
the $A$ and $A^T$ are updated within each iteration. For these
applications, it is possible to switch between different trans-
position methods by using performance data and statistical
data from earlier calls to the transposition function. We
leave the research for the adaptive transposition selection
mechanism in the future.

7. RELATED WORK

Parallel transposition of dense matrix [25, 13, 43]
has received much attention because of its widespread use
in leveraging the SIMD working pattern and accelerating
dense numerical linear algebra. Dotenko et al. [18] pointed out
that explicitly transposing dense data can significantly in-
crease the overall performance of parallel FFT. In this paper,
we show that transposing sparse data structures requires
more complex algorithm design and can improve many higher-
level routines as well.

A serial method for transposing sparse matrix has
been designed by Gustavson [26]. The approach uses sta-
ble sorting as the main primitive. Even though its parallel
implementation scales well on modern multi-core and many-
core machines when fast parallel sorting algorithm [28] is
used. It does not give the best observed performance in
our experiments because of inherently more memory trans-
actions.

Because the historically high cost of transposition, several
transposition-avoid algorithms and data structures
have been implemented. Freund developed a method [21]
to avoid SpMV_T in quasi-minimal residual (QMR) algo-
rum. Buluç et al. [8, 12] proposed a unified sparse matrix
format called compressed sparse blocks (CSB) to accelerate
both SpMV and SpMV_T operation in one single framework.
However, it may be hard to find general approaches to re-
design algorithms and data structures for avoiding transpo-
sition in a range of sparse matrix and graph applications.
In contrast, the work described in this paper transposes
the widely used CSR/CSC format thus can be easily adopted by
a majority of existing libraries.

Transposition can be implemented by using some parallel
primitives. Ashkiani et al. [5] recently designed a multisplit
method which in theory can be used for sparse matrix trans-
position. In their experiments, the new method behaves very
well when the number of buckets is 64 or smaller. When the
number of buckets grows to 65536, the new method is actu-
ally slower than sorting-based method. However, the number of buckets in sparse matrix transposition must be the number of rows/columns, which is up to multiple millions in our test suite. In this scenario, performance of using multisplit method decreases very fast and can be further slower than sorting-based method. In contrast, we show that the ScanTrans and MergeTrans methods proposed in this paper can be much faster than the sorting-based method. Moreover, we believe that our method can be potentially generalized to further accelerate the multisplit problem with a large amount of buckets.

8. CONCLUSIONS

In this paper, we proposed two new methods, ScanTrans and MergeTrans, for parallel sparse matrix and graph transposition, and implemented and optimized them for modern x86-based multi- and many-core processors. By using 22 sparse matrices from diverse application domains, we evaluated the two methods with the existing parallel transposition methods, including the atomic-based method, the sorting-based method, and the routine from the Intel MKL library, on Intel Haswell multi-core CPU and Xeon Phi many-core processor. Our experimental results showed that ScanTrans has the best performance on Intel Haswell CPUs, while MergeTrans has the best performance on Intel Xeon Phi. Compared to the counterpart from the Intel MKL library, the ScanTrans method achieved up to 5.6-fold and 6.2-fold speedups on Intel Haswell for single and double precision datatype, respectively; while on Intel Xeon Phi, the MergeTrans method achieved up to 11.7-fold and 9.9-fold speedups.

Because of the observed performance improvement on higher level routines using the new transposition methods, we believe more computation kernels for sparse matrix computations or graph processing can obtain benefits from this work.

9. ACKNOWLEDGEMENT

This research was supported in part by the NSF BIGDATA program via IIS-1247693, the NSF XPS program via CCF-1337131, the Department of Computer Science at Virginia Tech, and the Elizabeth and James Turner Fellowship. This research was also supported in part by the EU’s Horizon 2020 program under grant number 671633. Finally, we acknowledge Advanced Research Computing at Virginia Tech for access to high-performance computational resources and thank Kaiyong Zhao for helpful discussions about the simultaneous localization and mapping (SLAM) problem.

10. REFERENCES


