Locality-Aware Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication on Many-Core Processors

M. Ozan Karsavuran, Kadir Akbudak, and Cevdet Aykanat

Abstract—Sparse matrix-vector and matrix-transpose-vector multiplication ($\mathrm{SpMM^TV}$) repeatedly performed as $z \leftarrow A^T x$ and $y \leftarrow A\ z$ (or $y \leftarrow A\ w$) for the same sparse matrix A is a kernel operation widely used in various iterative solvers. One important optimization for serial $\mathrm{SpMM^TV}$ is reusing A-matrix nonzeros, which halves the memory bandwidth requirement. However, thread-level parallelization of $\mathrm{SpMM^TV}$ that reuses A-matrix nonzeros necessitates concurrent writes to the same output-vector entries. These concurrent writes can be handled in two ways: via atomic updates or thread-local temporary output vectors that will undergo a reduction operation, both of which are not efficient or scalable on processors with many cores and complicated cache-coherency protocols. In this work, we identify five quality criteria for efficient and scalable thread-level parallelization of $\mathrm{SpMM^TV}$ that utilizes one-dimensional (1D) matrix partitioning. We also propose two locality-aware 1D partitioning methods, which achieve reusing A-matrix nonzeros and intermediate z-vector entries; exploiting locality in accessing x-, y-, and z-vector entries; and reducing the number of concurrent writes to the same output-vector entries. These two methods utilize rowwise and columnwise singly bordered block-diagonal (SB) forms of A. We evaluate the validity of our methods on a wide range of sparse matrices. Experiments on the 60-core cache-coherent Intel Xeon Phi processor show the validity of the identified quality criteria and the validity of the proposed methods in practice. The results also show that the performance improvement from reusing A-matrix nonzeros compensates for the overhead of concurrent writes through the proposed SB-based methods.

Index Terms—Cache locality, sparse matrix, sparse matrix-vector multiplication, matrix reordering, singly bordered block-diagonal form, Intel Many Integrated Core Architecture (Intel MIC), Intel Xeon Phi

1 Introduction

The focus of this work is parallelization of sparse matrix-vector and matrix-transpose-vector multiplication ($\mathrm{SpMM^TV}$) operations on many-core processors. Several iterative methods perform repeated and consecutive computations of sparse matrix-vector (SpMV) and sparse matrix-transpose-vector ($\mathrm{SpM^TV}$) multiplications that involve the same sparse matrix A.

Typical examples include iterative methods for solving linear programming (LP) problems through interior point methods [1], [2]; the Biconjugate Gradient (BCG), the Conjugate Gradient for the Normal Equations (CGNE), the Conjugate Gradient for the Normal Residual (CGNR), and the Lanczos Biorthogonalization methods [3] for solving nonsymmetric linear systems; the LSQR method [4] for solving the least squares problem; the Surrogate Constraints method [5], [6] for solving the linear feasibility problem; the Hyperlink-Induced Topic Search (HITS) algorithm [7], [8] for rating web pages; and the Krylov-based balancing algorithms [9] used as preconditioners for sparse eigensolvers.

In the LP application, the SpMV operation immediately follows the $\mathrm{SpM^TV}$ operation in such a way that the output vector of $\mathrm{SpM^TV}$ becomes the input vector of SpMV. In CGNE, LSQR, Surrogate Constraints, and CGNR methods, the input vector of $\mathrm{SpM^TV}$ is obtained from the output vector of SpMV through linear vector operations. In BCG, Lanczos Biorthogonalization, HITS, and Krylov-based balancing algorithms, the SpMV and $\mathrm{SpM^TV}$ operations are totally independent.

The SpMV operation is known to be memory bound [10], [11], [12], [13] due to low operational intensity (flop-to-byte ratio, i.e., the ratio of the number of arithmetic operations to the number of memory accesses). Exploiting temporal locality through reusing input and/or output vector entries is expected to increase performance through reducing the memory bandwidth requirement of SpMV operations. Here, temporal locality refers to the reuse of data words (e.g., vector entries and matrix nonzeros) within a relatively small time duration, actually before eviction of the words from cache. As the $\mathrm{SpMM^TV}$ operation involves two SpMV operations with the same sparse matrix, reusing A-matrix nonzeros (together with their indices) is an opportunity towards further performance improvement over the opportunity of reusing input, output, and intermediate vector entries. Such data reuse opportunities become much more important on cache-coherent architectures involving large number of cores, such as the Xeon Phi processor.

In this work, we propose and discuss efficient parallel $\mathrm{SpMM^TV}$ algorithms that utilize the above-mentioned data

• *The authors are with the Department of Computer Engineering, Bilkent University, 6800, Ankara, Turkey.*
E-mail: {ozan.karsavuran, kadir, aykanat}@cs.bilkent.edu.tr.

Manuscript received 30 Nov. 2014; revised 23 May 2015; accepted 23 June 2015. Date of publication 7 July 2015; date of current version 18 May 2016. Recommended for acceptance by D. Trystram.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2453970

reuse opportunities. The proposed algorithms are directly applicable to the LP application as well as to the applications in which SpMV and $\mathrm{SpM^TV}$ operations are independent. The proposed algorithms are also applicable to the remaining applications as long as the intermediate linear vector operations that incur dependency between SpMV and $\mathrm{SpM^TV}$ operations do not require synchronization in parallelization. The applicability issues for these applications are discussed in Section 3.5.

We investigate four parallel $\mathrm{SpMM^TV}$ approaches that utilize one-dimensional (1D) rowwise and columnwise partitioning of A and A^T matrices. We identify five quality criteria for efficient thread-level parallelization of $\mathrm{SpMM^TV}$ based on the utilization of different data reuse opportunities. Four out of the five quality criteria refer to data reuse opportunities on reusing A-matrix nonzeros, reusing the intermediate vector entries, and data reuse in accessing vector entries during individual SpMVs. Reusing A-matrix nonzeros introduces the crucial problem of concurrent writes to either the intermediate or the output vector. The fifth quality criterion refers to the trade-off between the reuse of A-matrix nonzeros and concurrent writes.

We propose permuting A and A^T matrices into dual singly bordered block-diagonal (SB) forms to satisfy all five quality criteria simultaneously. We obtain two distinct parallel $\mathrm{SpMM^TV}$ algorithms by permuting A into a rowwise SB form, which induces a columnwise SB form of A^T, and permuting A into a columnwise SB form, which induces a rowwise SB form of A^T. We show that the objective of minimizing the size of the row or column border in the SB form of A corresponds to minimizing the number of concurrent writes in the respective parallel $\mathrm{SpMM^TV}$ algorithm.

We evaluate the validity of our proposed $\mathrm{SpMM^TV}$ algorithms on a single Xeon Phi processor for a wide range of sparse matrices. Although we experiment with Xeon Phi, our contributions are also viable for other cache-coherent shared memory architectures. The experimental results show that the performance improvement from reusing A-matrix nonzeros compensates for the overhead of concurrent writes through the proposed SB-form scheme. To our knowledge, this is the first work that successfully achieves reusing matrix nonzeros in $\mathrm{SpMM^TV}$ operations on many-core architectures.

The rest of the paper is organized as follows: Four viable parallel $\mathrm{SpMM^TV}$ approaches that utilize 1D rowwise and columnwise partitioning of A and A^T matrices are discussed in Section 2. The five quality criteria for efficient thread-level parallelization of $\mathrm{SpMM^TV}$ are presented in Section 3.1 and the two proposed SB-based $\mathrm{SpMM^TV}$ schemes achieving all these criteria are described in Section 3.2. Section 3.3 discusses the existing hypergraph-partitioning (HP)-based method utilized for permuting a sparse matrix into an SB form. The merits of using an SB form are discussed in Section 3.4 and the applicability of the proposed schemes to the iterative methods are investigated in Section 3.5. We present the experimental results in Section 4. The related work on SpMV on Xeon Phi and $\mathrm{SpMM^TV}$ algorithms is reviewed in Section 5. Finally, we conclude the paper in Section 6.

2 Parallel $\mathrm{SpMM^TV}$ Algorithms Based on 1D Matrix Partitioning

Consider an iterative algorithm involving $\mathrm{SpMM^TV}$ operations of the form $y \leftarrow AA^Tx$, which are performed as two successive SpMV operations $z \leftarrow A^Tx$ and $y \leftarrow A\ z$.

Based on 1D matrix partitioning, there are two viable parallel SpMV algorithms, namely row parallel and column parallel. The row-parallel SpMV algorithm utilizes rowwise partitioning, where each thread is held responsible for performing the submatrix-vector multiplication associated with a distinct row slice (submatrix). The column-parallel SpMV algorithm utilizes columnwise partitioning, where each thread is held responsible for performing the submatrix-vector multiplication associated with a distinct column slice.

In terms of inter-dependence among threads, the row-parallel algorithm incurs concurrent reads of the same input-vector entries, whereas the column-parallel algorithm incurs concurrent writes to the same output-vector entries. Concurrent reads do not incur any race conditions, however, concurrent writes do incur race conditions, which must be handled via either atomic updates or thread-local temporary output vectors. So, due to the more-expensive concurrent write operations, the row-parallel SpMV can be considered to be more advantageous than the column-parallel SpMV on shared memory architectures.

There are four viable parallel $\mathrm{SpMM^TV}$ algorithms based on 1D partitioning of A and A^T matrices:

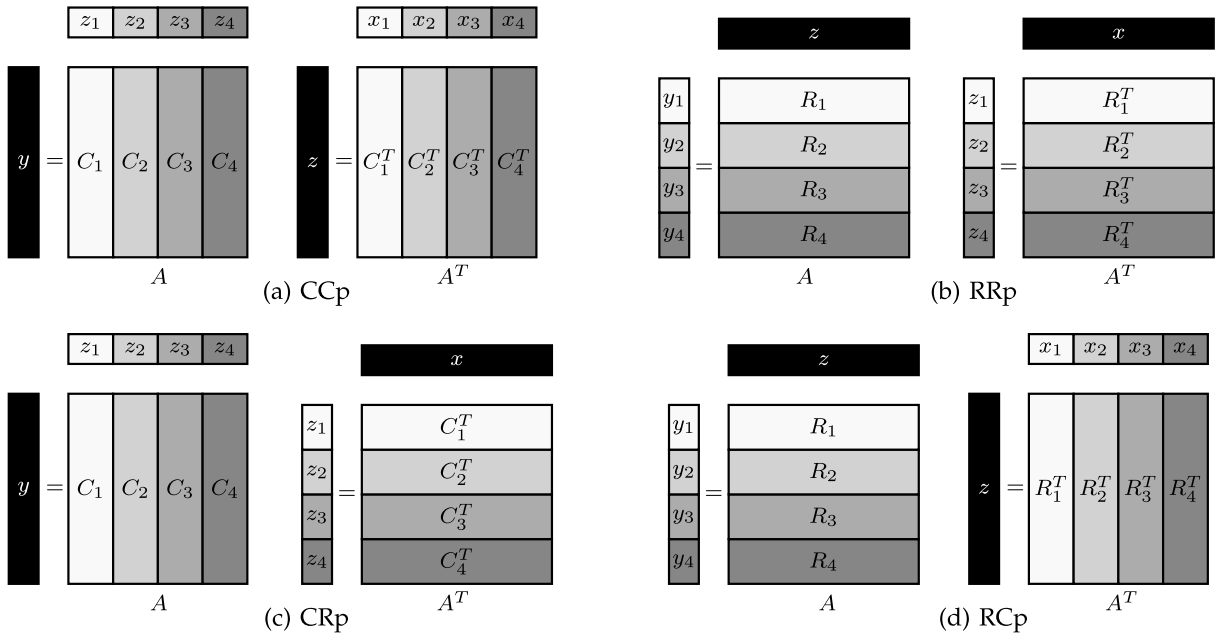
- Column-Column parallel (CCp)
- Row-Row parallel (RRp)
- Column-Row parallel (CRp)
- Row-Column parallel (RCp)

The CCp algorithm utilizes the column-parallel SpMV in both $y \leftarrow A\ z$ and $z \leftarrow A^Tx$, whereas the RRp algorithm utilizes the row-parallel SpMV in both $y \leftarrow A\ z$ and $z \leftarrow A^Tx$. The CRp algorithm utilizes the column-parallel SpMV for $y \leftarrow A\ z$ and row-parallel SpMV for $z \leftarrow A^Tx$, whereas the RCp algorithm utilizes the row-parallel SpMV for $y \leftarrow A\ z$ and column-parallel SpMV for $z \leftarrow A^Tx$.

Fig. 1 illustrates the four $\mathrm{SpMM^TV}$ algorithms for a parallel system with four threads. In each figure, the $z \leftarrow A^Tx$ computation involving the matrix on the right is performed first, then the $y \leftarrow A\ z$ computation involving the matrix on the left is performed. In the figure, a gray scale tone indicates the data exclusively used and/or computed by a single thread. The black color on the x, y, and z vectors indicates the data concurrently read and/or written by multiple threads.

In Fig. 1, a horizontal vector on the top of a matrix denotes the input vector of the respective SpMV computation, whereas a vertical vector denotes the output vector of the respective SpMV computation. Note that the intermediate z vector appears twice in each figure, vertical as the output vector of the first SpMV and horizontal as the input vector of the second SpMV. For example, in the $\mathrm{SpM^TV}$ computation of the RRp algorithm, each of the four z subvectors is exclusively computed by threads, whereas the whole z vector is concurrently read by all threads in the SpMV computation.

All the above-mentioned $\mathrm{SpMM^TV}$ methods are viable on distributed-memory architectures, however, CCp is not viable on cache-coherent many-core processors because it requires



A gray scale tone denotes exclusive accesses by a single thread, whereas black color denotes concurrent accesses by multiple threads.

Fig. 1. Four baseline SpMMTV algorithms for computing $y \leftarrow A\ z$ after $z \leftarrow A^T x$ by four threads.

expensive concurrent writes in both SpMV operations. For this reason, CCp is not investigated in the rest of the paper.

3 The Proposed SpMM$^{\mathrm{T}}$V Methods

3.1 Quality Criteria for Efficient Parallelization

We identify five quality criteria given in Table 1 for efficient thread-level parallelization of the above-mentioned SpMM$^{\mathrm{T}}$V algorithms, which utilize 1D matrix partitioning.

The RRp algorithm has the nice property of avoiding expensive concurrent writes in both SpMVs, therefore, it automatically satisfies quality criterion (e). However, RRp fails to satisfy criterion (b) since it necessitates storing A and A^T matrices separately. RRp also fails to satisfy criterion (a) since all threads require the whole z vector during the row-parallel SpMV $y \leftarrow A\ z$.

The CRp algorithm automatically satisfies quality criterion (a), whereas RCp fails to satisfy this criterion. Both CRp and RCp have the potential of satisfying quality criterion (b) since neither of them necessitates storing A and A^T separately. Both CRp and RCp fail to satisfy quality criterion (e) since both of them contain expensive concurrent writes to the whole output vector in one of the two SpMVs.

In this work, we utilize a special form of sparse matrices – namely SB form – to develop parallel CRp and RCp algorithms that satisfy all quality criteria. We will show that SB forms together with the proposed CRp and RCp algorithms automatically satisfy criteria (a) and (b). Two 1D matrix partitioning schemes are adopted to find row/column reorderings to permute A and A^T matrices into the desired SB forms. In both partitioning schemes, the partitioning objective corresponds to satisfying the remaining three criteria (c), (d), and (e).

TABLE 1
Quality Criteria for Efficient Parallelization of SpMM$^{\mathrm{T}}$V Algorithms

Quality criteria
(a) Reusing z-vector entries generated in $z \leftarrow A^T x$ and then read in $y \leftarrow A\ z$
(b) Reusing matrix nonzeros (together with their indices) in $z \leftarrow A^T x$ and $y \leftarrow A\ z$
(c) Exploiting temporal locality in reading input vector entries in row-parallel SpMVs
(d) Exploiting temporal locality in updating output vector entries in column-parallel SpMVs
(e) Minimizing the number of concurrent writes performed by different threads in column-parallel SpMVs

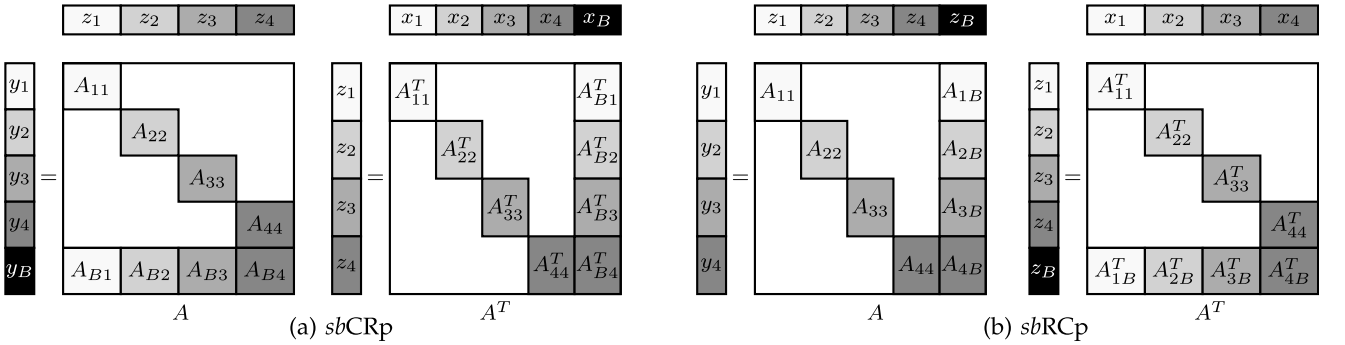
3.2 CRp and RCp Algorithms Based on SB Forms

We propose two methods for SpMM$^{\mathrm{T}}$V: The first method uses the CRp algorithm with the rowwise SB form of A, whereas the second one utilizes the RCp algorithm together with the columnwise SB form of A. Here and hereafter, the first and second methods will be respectively referred to as *sb*CRp and *sb*RCp. The following two subsections, Sections 3.2.1 and 3.2.2, present the proposed 1D matrix partitioning schemes together with the associated SpMM$^{\mathrm{T}}$V algorithms. More details are given for *sb*CRp in Section 3.2.1, and *sb*RCp is summarized in Section 3.2.2 because *sb*RCp can be considered a dual form of *sb*CRp.

3.2.1 The sbCRp Method

The Parallel Algorithm. Consider a row/column reordering that permutes matrix A into a rowwise SB form as:

$$\begin{aligned}\hat{A} &= A_{rSB} = P_r A P_c \\ &= \begin{bmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{KK} \\ A_{B1} & A_{B2} & \dots & A_{BK} \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_K \\ R_B \end{bmatrix} \\ &= [\ C_1 \quad C_2 \quad \dots \quad C_K\ ]. \end{aligned} \tag{1}$$



A gray scale tone denotes exclusive accesses by a single thread, whereas black color denotes concurrent accesses by multiple threads.

Fig. 2. Proposed SB-based SpMM$^{\mathrm{T}}$V algorithms for computing $y \leftarrow A\ z$ after $z \leftarrow A^T x$ by four threads.

In (1), P_r and P_c respectively denote the row and column permutation matrices. A_{kk} denotes the kth diagonal block of A_{rSB}. R_k and C_k respectively denote the kth row and column slices of A_{rSB} for $k = 1, 2, \ldots, K$. R_B denotes the row border as follows:

$$R_B = [\, A_{B1} \;\; A_{B2} \;\; \ldots \;\; A_{BK} \,]. \tag{2}$$

Here, A_{Bk} denotes the kth border block in the row border R_B. In A_{rSB}, the columns of diagonal blocks are coupled by rows in the row border. That is, each coupling row in R_B has nonzeros in the columns of at least two diagonal blocks. A coupling row $r_i \in R_B$ is said to have a connectivity of $\lambda(r_i)$ if and only if $r_i \in R_B$ has at least one nonzero at each of the $\lambda(r_i)$ A_{Bk} submatrices.

The rowwise SB form of A given in (1) induces the columnwise SB form of A^T as follows:

$$\begin{aligned}(A_{rSB})^T = \hat{A}^T = A^T_{cSB} &= P_c A^T P_r \\ &= \begin{bmatrix} A_{11}^T & & & & A_{B1}^T \\ & A_{22}^T & & & A_{B2}^T \\ & & \ddots & & \vdots \\ & & & A_{KK}^T & A_{BK}^T \end{bmatrix} = \begin{bmatrix} C_1^T \\ C_2^T \\ \vdots \\ C_K^T \end{bmatrix} \\ &= \begin{bmatrix} R_1^T & R_2^T & \ldots & R_K^T & R_B^T \end{bmatrix}. \end{aligned} \tag{3}$$

In (3), A_{kk}^T denotes the kth diagonal block of A^T_{cSB}. C_k^T and R_k^T respectively denote the kth row and column slice of A^T_{cSB} for $k = 1, 2, \ldots, K$. R_B^T denotes the column border as follows:

$$R_B^T = \begin{bmatrix} A_{B1}^T \\ A_{B2}^T \\ \vdots \\ A_{BK}^T \end{bmatrix}. \tag{4}$$

Here, A_{Bk}^T denotes the kth border block in the column border R_B^T. In A^T_{cSB}, the rows of diagonal blocks are coupled by columns in the column border. That is, each coupling column in R_B^T has nonzeros in the rows of at least two diagonal blocks. A coupling column $c_j \in R_B^T$ is said to have a connectivity of $\lambda(c_j)$ if and only if $c_j \in R_B^T$ has at least one nonzero at each of $\lambda(c_j)$ A_{Bk}^T submatrices.

In the proposed partitioning scheme, K is selected in such a way that the size of each column slice C_k together with the associated input and output subvectors is below the size of the cache of a single core. Both submatrix-transpose-vector multiplication $z_k \leftarrow C_k^T\ x$ and submatrix-vector multiplication $y \leftarrow C_k\ z_k$ are considered as an atomic task, which is assigned to a thread executed by a single core of the Xeon Phi architecture. So this partitioning and task-to-thread assignment scheme leads to the *sb*CRp method, as shown in Algorithm 1. In this algorithm, as well as in Algorithm 2, the "for ... in parallel do" constructs mean that each iteration of the for loop can be executed in parallel. Fig. 2a displays the matrix view of the parallel *sb*CRp method given in Algorithm 1. In this figure, the x_B and y_B vectors are also referred to as "border subvectors" throughout the paper. In this figure, as well as in Fig. 2b, concurrently accessed subvectors are colored black.

Algorithm 1. The Proposed *sb*CRp Method

Require: A_{kk} and A_{Bk} matrices; x, y, and z vectors
1: **for** $k \leftarrow 1$ **to** K **in parallel do**
2: $\quad z_k \leftarrow A_{kk}^T\ x_k$ } $z_k \leftarrow C_k^T\ x$
3: $\quad z_k \leftarrow z_k + A_{Bk}^T\ x_B$ }
4: $\quad y_k \leftarrow A_{kk}\ z_k$ } $y \leftarrow C_k\ z_k$
5: $\quad y_B \leftarrow y_B + A_{Bk}\ z_k$ ▷ Concurrent writes }
6: **end for**

The row-parallel SpMV algorithm incurs input dependency, whereas the column-parallel SpMV algorithm incurs output dependency among threads. As seen in Algorithm 1, the proposed method utilizing the SB form enables both input and output independency among threads for SpMV computations on diagonal blocks and their transposes. That is, SpMV computations $z_k \leftarrow A_{kk}^T\ x_k$ and $y_k \leftarrow A_{kk}\ z_k$ in lines 2 and 4 are performed concurrently and independently by threads. Note that the write-read dependency between these two SpMV computations incurs only intra-thread dependency due to the z_k vector. The $z_k \leftarrow z_k + A_{Bk}^T\ x_B$ computation in line 3 incurs input dependency among threads via the border subvector x_B. The $y_B \leftarrow y_B + A_{Bk}\ z_k$ computation in line 5 incurs output dependency among threads via the border subvector y_B.

Quality criteria coverage. The working set of every for-loop iteration fits into the cache of a single core due to the partitioning and task-to-thread assignment scheme adopted in the *sb*CRp method. Hence, the *sb*CRp method achieves both quality criteria (a) and (b) by enabling the reuse of z-vector entries and matrix nonzeros, respectively, between $z_k \leftarrow C_k^T\ x$ and $y \leftarrow C_k\ z_k$ computations. Under a fully

associative cache assumption, there will be no cache misses during the $y \leftarrow A\ z$ computation, neither in reading z-vector entries nor in accessing A-matrix nonzeros. Note that misses in a fully associative cache are compulsory or capacity misses and are not conflict misses. Also note that quality metric (b) can only be achieved by storing the A matrix only once, that is, A and A^T matrices are not stored separately. Therefore, reusing nonzeros of diagonal blocks in $z_k \leftarrow A_{kk}^T\ x_k$ and $y_k \leftarrow A_{kk}\ z_k$ computations can be achieved by storing A_{kk} in compressed sparse columns (CSC) format which corresponds to storing A_{kk}^T in compressed sparse rows (CSR) format, or vice versa. Similarly, reusing nonzeros of border blocks in $z_k \leftarrow z_k + A_{Bk}^T\ x_B$ and $y_B \leftarrow y_B + A_{Bk}\ z_k$ computations can also be achieved by storing A_{Bk} in CSC format, which corresponds to storing A_{Bk}^T in CSR format, or vice versa. To some extent, quality metric (b) can be achieved by storing A in the Coordinate (COO) format, however COO is likely to have lower performance due to its higher index-storage overhead.

We make the following notes for achieving quality criteria (c) and (d): For CSR-/CSC-based sequential SpMV, reordering the rows/columns with similar sparsity patterns nearby is likely to increase temporal locality in accessing input-/output-vector entries as mentioned in [14]. So, for row-/column-parallel SpMV, temporal locality in accessing input-/output-vector entries can be exploited by clustering rows/columns with similar sparsity patterns to the same row/column blocks. In the following two paragraphs, we show how the SB forms of matrix A are utilized to achieve quality criteria (c) and (d) in parallel $\mathrm{SpMM^TV}$ operations. Also note that, in the rest of the paper, the analyses on the number of data accesses are given under single-word line-size assumption. The upper bounds given in these analyses are still valid for multiple-word line-sizes.

For achieving quality criterion (c), temporal locality in accessing input-vector (x-vector) entries in row-parallel SpMV s $z_k \leftarrow C_k^T\ x$ can be exploited by utilizing the columnwise SB form of the A^T matrix as follows: The diagonal block computations $z_k \leftarrow A_{kk}^T\ x_k$ (line 2 of Algorithm 1) share no x_k-vector entries due to the structure of the columnwise SB form. Hence, under a fully associative cache assumption, only one cache miss will occur for each of such x_k-vector entries. On the other hand, the border block computations $z_k \leftarrow z_k + A_{Bk}^T\ x_B$ (line 3 of Algorithm 1) do share x_B-vector entries because of the coupling columns in R_B^T. Under a fully associative cache assumption, column $c_j \in R_B^T$ with a connectivity of $\lambda(c_j)$ will incur at most $\lambda(c_j)$ cache misses due to accessing x_j. $\lambda(c_j)$ is an upper bound on the number of cache misses because of the possibility of reusing x_j, which can only occur when the SpMV computations associated with two border blocks having nonzeros in column c_j are executed consecutively on the same core. Therefore, this upper bound is rather tight, and hence, minimizing the sum of the connectivities of the coupling columns in R_B^T closely relates to minimizing the number of cache misses in performing $z \leftarrow A^T x$.

For achieving quality criterion (d), temporal locality in updating the output-vector (y-vector) entries in column-parallel SpMVs $y \leftarrow C_k\ z_k$ can be exploited by utilizing the rowwise SB form of the A matrix as follows: The diagonal block computations $y_k \leftarrow A_{kk}\ z_k$ (line 4 of Algorithm 1) share no y_k-vector entries due to the structure of the rowwise SB form. Hence, under a fully associative cache assumption, at most two cache misses (for reading and writing) will occur for each of such y_k-vector entries. On the other hand, the border block computations $y_B \leftarrow y_B + A_{Bk}\ z_k$ (line 5 of Algorithm 1) do share y_B-vector entries because of the coupling rows in R_B. Under a fully associative cache assumption, row $r_i \in R_B$ with a connectivity of $\lambda(r_i)$ will incur at most $2\lambda(r_i)$ cache misses due to updating y_i. $2\lambda(r_i)$ is an upper bound on the number of cache misses because of the possibility of reusing y_i, which can only occur when the SpMV computations associated with two border blocks having nonzeros on row r_i are executed consecutively on the same core. Therefore, this upper bound is rather tight, and hence, minimizing the sum of the connectivities of the coupling rows in R_B closely relates to minimizing the number of cache misses in performing $y \leftarrow A\ z$.

We make the following notes on how criteria (b), (c), and (d) are related. Consider an $\mathrm{SpMM^TV}$ algorithm that achieves criterion (b). If this algorithm achieves criterion (c), it automatically achieves criterion (d) and vice versa. This is because criterion (b) can be achieved by storing the A matrix only once and a rowwise SB form of A induces a columnwise SB form of A^T and vice versa.

For achieving quality criterion (e), minimizing the number of concurrent writes to the output-vector (y-vector) entries in column-parallel SpMV s $y \leftarrow C_k\ z_k$ can be accomplished by utilizing the rowwise SB form of the A matrix as follows: Concurrent writes occur only during the border block computations. Consider two possible implementations of each border block computation $y_B \leftarrow y_B + A_{Bk}\ z_k$, namely CSC and CSR schemes. The CSC scheme will incur one concurrent write for each nonzero of the row border R_B, whereas the CSR scheme will incur only one concurrent write for each nonzero row of the A_{Bk} matrices of R_B. That is, to update $y_i \in y_B$, the CSC and CSR schemes will respectively incur $nnz(r_i)$ and $\lambda(r_i)$ concurrent writes. Here, $nnz(r_i)$ and $\lambda(r_i)$ respectively denote the number of nonzeros and connectivity of row $r_i \in R_B$. Hence, the CSR scheme is selected since it requires a much smaller number of concurrent writes. Therefore, each border block A_{Bk} is stored in CSR format, which corresponds to storing A_{Bk}^T in CSC format. Hence, under this implementation scheme, minimizing the sum of the connectivities of coupling rows in R_B exactly corresponds to minimizing the number of concurrent writes in performing $y \leftarrow A\ z$. This one-to-one correspondence is valid when concurrent writes are handled via atomic updates, however, this correspondence depends on the algorithm used in reducing the temporary vectors when concurrent writes are handled using thread-local temporary vectors.

We should mention here that criteria (d) and (e) are closely related for column-parallel SpMV. Criterion (e) is given to clarify the conditions under which achieving criterion (d) implies achieving criterion (e) of minimizing the number of concurrent writes in a parallel implementation. These conditions are storing row border submatrices in CSR format and identifying the output vector entries that

will incur concurrent writes prior to execution of the $\mathrm{SpMM^TV}$ algorithm.

As a consequence of the above-mentioned discussions about quality coverage, the proposed *sb*CRp method can achieve criterion (c) by minimizing the sum of the connectivities of the coupling columns in R_B^T and can achieve both quality criteria (d) and (e) by minimizing the sum of the connectivities of the coupling rows in R_B. Note that the former and latter minimization objectives are equivalent since the columns of R_B^T correspond to the rows of R_B. Thus, the *sb*CRp method can achieve all three quality criteria (c), (d), and (e) by permuting matrix A into the rowwise SB form with the objective of minimizing the sum of the connectivities of the coupling rows. Consequently, *sb*CRp satisfies all quality criteria since CRp already achieves quality criteria (a) and (b).

3.2.2 The sb*RCp Method*

The Parallel Algorithm. The *sb*RCp method utilizes the columnwise SB form of matrix A as follows:

$$\begin{aligned}\hat{A} &= A_{cSB} = P_r A P_c \\ &= \begin{bmatrix} A_{11} & & & & A_{1B} \\ & A_{22} & & & A_{2B} \\ & & \ddots & & \vdots \\ & & & A_{KK} & A_{KB} \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_K \end{bmatrix} \\ &= [\ C_1 \quad C_2 \quad \dots \quad C_K \quad C_B\ ].\end{aligned} \tag{5}$$

This columnwise SB form of A induces the rowwise SB form of A^T as follows:

$$\begin{aligned}(A_{cSB})^T = \hat{A} = A_{rSB}^T &= P_c A^T P_r \\ &= \begin{bmatrix} A_{11}^T & & & \\ & A_{22}^T & & \\ & & \ddots & \\ & & & A_{KK}^T \\ A_{1B}^T & A_{2B}^T & & A_{KB}^T \end{bmatrix} = \begin{bmatrix} C_1^T \\ C_2^T \\ \vdots \\ C_K^T \\ C_B^T \end{bmatrix} \\ &= [\ R_1^T \quad R_2^T \quad \dots \quad R_K^T\ ].\end{aligned} \tag{6}$$

In this partitioning scheme, K is selected in such a way that the size of each row slice R_k together with the associated input and output subvectors is below the size of the cache of a single core. The *sb*RCp method is presented in Algorithm 2. Similar to the *sb*CRp method, the *sb*RCp method given in Algorithm 2 enables both input and output independency among threads for SpMV computations on diagonal blocks and their transposes (as shown in lines 2 and 4). In a dual manner to the *sb*CRp method, the $z_B \leftarrow z_B + A_{kB}^T\ x_k$ computation in line 3 incurs output dependency among threads via border subvector z_B, whereas the $y_k \leftarrow y_k + A_{kB}\ z_B$ computation in line 7 incurs input dependency among threads via border subvector z_B. The existence of both output and input dependencies on the same border subvector z_B incurs additional synchronization overhead, as depicted by the two consecutive "for ... in parallel do" constructs in Algorithm 2. Fig. 2b displays the matrix view of the parallel *sb*RCp method given in Algorithm 2.

Algorithm 2. The Proposed *sb*RCp Method

Require: A_{kk} and A_{kB} matrices; x, y, and z vectors
1: **for** $k \leftarrow 1$ **to** K **in parallel do**
2: $\quad z_k \leftarrow A_{kk}^T\ x_k$
3: $\quad z_B \leftarrow z_B + A_{kB}^T\ x_k$ $\quad \triangleright$ Concurrent writes $\quad \}\ z \leftarrow R_k^T\ x_k$
4: $\quad y_k \leftarrow A_{kk}\ z_k$
5: **end for**
6: **for** $k \leftarrow 1$ **to** K **in parallel do**
7: $\quad y_k \leftarrow y_k + A_{kB}\ z_B$ $\quad \}\ y_k \leftarrow R_k\ z$
8: **end for**

For the BCG, Lanczos Biorthogonalization, HITS, and Krylov-based balancing algorithms, the two for loops of *sb*RCp given in Algorithm 2 can be fused since there is no input/output dependency between the $z \leftarrow A^T x$ and $y \leftarrow A\ w$ operations. On the average, the fusion of these two for loops provides 10 percent performance improvement over the non-fused case for the nonsymmetric square matrices given in Table 4.

Quality criteria coverage. The matrix partitioning scheme of the proposed *sb*RCp method can be considered a dual form of the *sb*CRp method given in Section 3.2.1. So the discussion for the *sb*RCp method , in general, follows the discussion given for the *sb*CRp method in a dual manner. Here, we will briefly discuss the *sb*RCp method, focusing on the differences. The *sb*RCp method satisfies all quality criteria in general. It satisfies quality criteria (c)–(e) for all respective computations. It also satisfies quality criteria (a) and (b) for the $z_k \leftarrow A_{kk}^T\ x_k$ and $y_k \leftarrow A_{kk}\ z_k$ computations. However, it fails to satisfy quality criteria (a) and (b) for the $z_B \leftarrow z_B + A_{kB}^T\ x_k$ and $y_k \leftarrow y_k + A_{kB}\ z_B$ computations.

Despite the above-mentioned disadvantages, *sb*RCp can still be preferred since some of the nonsymmetric square and rectangular matrices may be more suitable for permuting into a columnwise SB form , whereas some other matrices may be more suitable for permuting into a rowwise SB form . This property is because of the differences in row and column sparsity patterns of a given nonsymmetric square or rectangular matrix.

3.3 Permuting Matrices into SB Form

For the proposed *sb*CRp and *sb*RCp algorithms, we utilize the hypergraph-partitioning-based methods in [15] for permuting matrices into rowwise and columnwise SB forms via row/column reordering. Here, we will briefly discuss this HP approach for *sb*CRp, where a dual discussion applies for *sb*RCp.

In *sb*CRp, the HP-based method utilizes the row-net hypergraph model [16] for obtaining a rowwise SB form. In the HP problem, the partitioning constraint is to maintain balance on the weights of the parts and the partitioning objective is to minimize the cutsize, defined as the sum of the connectivities of the cut nets. For *sb*CRp, the partitioning constraint encapsulates balancing the nonzero counts of the column slices, which in turn corresponds to maintaining balance on computational loads of K submatrix-vector and submatrix-transpose-vector multiplications. The partitioning objective encapsulates minimizing the sum of the

TABLE 2
Quality Criteria Coverage of $\mathrm{SpMM^TV}$ Algorithms

Quality Criteria	RRp	CRp	RCp	*sb*CRp	*sb*RCp
(a) z-vector reuse	×	✓	×	✓	✓[1]
(b) A-matrix nonzero reuse	×	✓	×	✓	✓[2]
(c) temporal locality in row-parallel SpMV	×[3]	×[3]	×[3]	✓	✓
(d) temporal locality in col-parallel SpMV	–	×[3]	×[3]	✓	✓
(e) minimizing concurrent writes	✓	×	×	✓	✓

✓: satisfied
–: not applicable
×: not satisfied
✓[1]: satisfied except z_B border subvectors
✓[2]: satisfied except A_{kB} border submatrices
×[3]: may be satisfied through row/column reordering

connectivities of the coupling rows in the row border R_B of A_{rSB}, which in turn corresponds to satisfying quality criteria (c), (d), and (e).

3.4 Merits of SB Forms in CRp and RCp Algorithms

Table 2 displays the comparison of the $\mathrm{SpMM^TV}$ algorithms in terms of quality criteria coverage. As seen in the table, RRp does not satisfy quality criteria (a) or (b), as mentioned in Section 3.1. On the other hand, separate storages of A and A^T enable avoiding concurrent writes, which in turn corresponds to satisfying (e). Note that RRp can be enhanced to satisfy criterion (c), and both CRp and RCp can be enhanced to satisfy criteria (c) and (d) via adopting intelligent cache-aware row/column reordering methods such as the Reverse Cuthill-McKee (RCM) algorithm [12], [17].

CRp satisfies only criteria (a) and (b). Utilizing the SB form of A in the proposed *sb*CRp method extends the quality coverage of CRp so that criteria (c), (d), and (e) are satisfied by *sb*CRp. These merits of the SB form can also be observed by comparing the black vectors/subvectors in Figs. 1c and 2a. In the CRp algorithm, whole x and y vectors are concurrently accessed by all threads, whereas in the *sb*CRp method, only x_B and y_B subvectors are concurrently accessed by all threads, and each of the other subvectors is accessed by one distinct thread.

RCp satisfies none of the quality criteria. Utilizing the SB form of A in the proposed *sb*RCp method enables reusing z-vector entries and matrix nonzeros in the diagonal submatrix-vector multiplies, whereas it cannot achieve the reuse in the border submatrix-vector multiplies. So *sb*RCp partially satisfies criteria (a) and (b) while satisfying criteria (c), (d), and (e). These merits of the SB form can also be observed by comparing Figs. 1d and 2b. In the RCp algorithm, the whole z vector is concurrently accessed by all threads, whereas in the *sb*RCp method, only the z_B subvector is concurrently accessed by all threads, and each of other subvectors is accessed by one distinct thread.

3.5 Applicability of the Methods

The discussion given in Sections 3.2.1 and 3.2.2 about the quality criteria coverage of the proposed methods is based on the assumption that the second SpMV immediately follows the first SpMV with no intermediate computation, where the input of the second SpMV is equal to the output of the first SpMV. The proposed methods are therefore directly applicable to linear programming problems.

TABLE 3
Coverage of Quality Criteria (a) and (b) of the $\mathrm{SpMM^TV}$ Algorithms for Several Iterative Methods

Iterative Methods		CRp	RCp	*sb*CRp	*sb*RCp
Directly applicable					
LP [1], [2]	$z \leftarrow A^T x$ $y \leftarrow Az$	✓	×	✓	✓
Directly (no dependency since inner product can be delayed)					
CGNE [3]	$z \leftarrow q - Ax$ $\beta \leftarrow (z,z)/(q,q)$ $y \leftarrow A^T z$	✓	×	✓	✓
Directly (linear vector operations without synchronization)					
LSQR [4]	$z \leftarrow Ax$ $w \leftarrow f(z)$ $y \leftarrow A^T w$	✓	×	✓	✓
Surrogate Constraints [5], [6]	$z \leftarrow Ax$ $w \leftarrow f(z)$ $y \leftarrow A^T w$	✓	×	✓	✓
Independent SpMV s (the two for loops of *sb*RCp can be fused.)					
BCG [3]	$z \leftarrow Ax$ $y \leftarrow A^T w$	✓	×	✓	✓
Lanczos Biorthogonalization [3]	$z \leftarrow Ax$ $y \leftarrow A^T w$	✓	×	✓	✓
HITS [7], [8]	$z \leftarrow Ax$ $y \leftarrow A^T w$	✓	×	✓	✓
Krylov-based Balancing [9]	$z \leftarrow Ax$ $y \leftarrow A^T x$	✓	×	✓	✓
Not applicable due to inner product and inter-dependency					
CGNR [3]	$z \leftarrow Ax$ $\alpha \leftarrow \|\|y\|\|_2^2 / \|\|z\|\|_2^2$ $y \leftarrow A^T \alpha w$	×	×	×	×

The quality coverage of the proposed methods is still valid if the input of the second SpMV is obtained from the output of the first SpMV via linear vector operations that incur no thread synchronization. If, however, intermediate parallel computations incur synchronization, they disable the reuse of z-vector entries and A-matrix nonzeros, thus preventing the proposed methods from satisfying quality criteria (a) and (b). Typically, an intermediate synchronization point is incurred by an inner-product computation that involves the output of the first SpMV, the scalar result of which is used in computing the input of the second SpMV via other linear vector operations. Note that such intermediate synchronization points do not adversely affect the coverage of quality criteria (c), (d), or (e) for the $\mathrm{SpMM^TV}$ algorithms. Table 3 summarizes the computational structure of several iterative algorithms to discuss coverage of quality criteria (a) and (b). In the table, $f(\cdot)$ denotes a set of linear vector operations that do not involve explicit synchronization.

As seen in Table 3, the quality coverage of (a) and (b) for the $\mathrm{SpMM^TV}$ algorithms are disturbed only in the CGNR method. This is expected since CGNR involves an intermediate synchronization point due to the inner-product computation. Note that although CGNE has an intermediate inner-product computation [3], this is not a problem since

TABLE 4
Properties of Test Matrices and K Values of Their SB Forms

		Number of			Nnz in row		Nnz in col		K	
Matrix	Kind	rows	cols	nonzeros	avg	max	avg	max	A_{rSB}	A_{cSB}
ASIC_680ks	circuit simulation	682,712	682,712	2,329,176	3	210	3	210	469	469
circuit5M_dc	circuit simulation	3,523,317	3,523,317	19,194,193	5	27	5	25	3,730	3,730
dbic1	linear programming	43,200	226,317	1,081,843	25	1,453	5	38	201	212
degme	linear programming	185,501	659,415	8,127,528	44	624,079	12	18	1,500	1,488
fome21	linear programming	67,596	216,350	465,294	7	96	2	3	90	99
Freescale1	circuit simulation	3,428,755	3,428,755	18,920,347	6	27	6	25	3,674	3,674
image_interp	computer graphics	232,485	120,000	711,683	3	5	6	6	145	138
kneser_10_4_1	combinatorial	349,651	330,751	992,252	3	16	3	3	204	202
LargeRegFile	circuit simulation	2,111,154	801,374	4,944,201	2	4	6	655,876	866	955
lp_osa_14	linear programming	2,337	54,797	317,097	136	38,336	6	6	59	47
lp_osa_30	linear programming	4,350	104,374	604,488	139	72,555	6	6	111	89
lung2	computational fluid	109,460	109,460	492,564	4	8	4	8	97	97
memchip	circuit simulation	2,707,524	2,707,524	14,810,202	5	27	5	27	2,878	2,878
neos	linear programming	479,119	515,905	1,526,794	3	29	3	16,220	308	312
ohne2	semiconductor device	181,343	181,343	11,063,545	61	3,441	61	3,441	2,037	2,037
para-6	semiconductor device	155,924	155,924	5,416,358	35	6,931	35	6,931	1,002	1,002
pds-100	linear programming	156,016	514,577	1,096,002	7	101	2	3	211	233
pds-80	linear programming	128,954	434,580	927,826	7	96	2	3	178	197
scircuit	circuit simulation	170,998	170,998	958,936	6	353	6	353	187	187
sgpf5y6	linear programming	246,077	312,540	831,976	3	61	3	12	168	172
Stanford	directed graph	261,588	281,731	2,312,497	9	38,606	8	255	440	425
Stanford_Berkeley	directed graph	615,384	678,711	7,583,376	12	83,448	11	249	1,427	1,348
stat96v1	linear programming	5,995	197,472	588,798	98	666	3	18	109	120
stat96v2	linear programming	29,089	957,432	2,852,184	98	3,232	3	12	525	581
watson_2	linear programming	352,013	677,224	1,846,391	5	93	3	15	360	380
web-BerkStan	directed graph	680,486	617,094	7,600,595	11	249	12	84,208	1,347	1,430
web-Stanford	directed graph	281,731	261,588	2,312,497	8	255	9	38,606	426	440
wheel_601	combinatorial	902,103	723,605	2,170,814	2	602	3	3	453	442

the inner-product computation can be deferred to the end of the second SpMV.

4 Experiments

4.1 Data sets

The validity of the proposed methods are tested on 28 non-symmetric square and rectangular sparse matrices arising in different application domains. All test matrices are selected from the University of Florida Sparse Matrix Collection [18]. Twelve of the test matrices in the set are LP constraint matrices since the $\mathrm{SpMM^TV}$ operation is directly used in solving LP problems. Web-link matrices are also included since PageRank computations implemented via using Krylov-subspace methods (e.g., BCG) [19] and the HITS algorithm [7], [8] operate on such matrices.

Table 4 displays properties of A matrices in the test set. The matrices are listed in alphabetical order by name. In the table, "avg" and "max", respectively, denote the average and the maximum number of nonzeros per row/column. Table 4 also displays the number K of parts of the SB forms of the test matrices in the last two columns.

4.2 Experimental Framework

The performances of the proposed SB-based *sb*CRp and *sb*RCp methods are evaluated against the RRp, CRp, and RCp algorithms. Recall that *sb*CRp and *sb*RCp given in Algorithms 1 and 2 utilize rowwise and columnwise SB forms of A, as illustrated in Figs. 2a and 2b, respectively. Performance comparisons of *sb*CRp against CRp and *sb*RCp against RCp are reported to experimentally validate the merits (see Section 3.4) of the SB form in the CRp and RCp algorithms. The performance of baseline algorithms CRp and RCp are tested according to two different row/column orderings of A: original ordering and RCM ordering. The former and latter schemes will be respectively referred to as *org*CRp, *org*RCp and *rcm*CRp, *rcm*RCp. *rcm*CRp and *rcm*RCp respectively enable CRp and RCp to satisfy quality criteria (c) and (d), as shown by "$\times^3$" in Table 2.

RRp is also selected as a baseline algorithm for performance comparison. The performance of RRp is also tested according to the original and RCM orderings of A, which will be referred to as *org*RRp and *rcm*RRp, respectively. *rcm*RRp enables RRp to satisfy quality criterion (c), as also shown by "$\times^3$" in Table 2. In addition to our RRp implementation, a vendor-provided SpMV routine, `mkl_dcsrmv`, of Intel Math Kernel Library (MKL) [20], is also utilized to implement a variant of RRp. The performance of MKL is also tested according to the original and RCM orderings of A, which will be referred to as *org*MKL and *rcm*MKL, respectively.

In all of the baseline implementations, except MKL, either the original matrix or the RCM-ordered matrix is partitioned either rowwise or columnwise in such a way that the size of each row/column slice is just above the cache size threshold. For obtaining these partitionings, we utilize a simple heuristic that assigns successive rows/columns to

TABLE 5
Performance Results Normalized wrt Those of RRp for Original Order and Normalized Border Sizes

Matrix	Parallelization schemes																
	Row-Row (RRp)				Col.-Row (CRp)			Row-Col. (RCp)			Border sizes		The best of				
	org. (ms)	RCM	MKL		org.	RCM	SB	org.	RCM	SB	A_{rSB}	A_{cSB}	RRp	CRp/RCp			
			org.	RCM										org.	RCM	SB	
ASIC_680ks	2.067	0.88	1.73	1.38	1.68	1.54	**0.70**	1.79	1.66	0.79	0.06	0.06	0.88	1.68	1.54	**0.70**	
circuit5M_dc	10.535	0.93	2.20	2.14	1.54	1.37	**0.83**	1.62	1.64	0.90	0.06	0.06	0.93	1.54	1.37	**0.83**	
dbic1	1.367	0.93	1.79	1.48	1.18	0.84	**0.50**	2.84	2.23	1.91	0.28	0.67	0.93	1.18	0.84	**0.50**	
degme	15.295	0.89	0.82	0.80	0.43	0.29	**0.20**	7.09	6.71	6.44	0.53	0.97	0.80	0.43	0.29	**0.20**	
fome21	1.002	0.64	0.95	0.77	1.05	0.81	**0.62**	2.02	1.36	0.75	0.24	0.15	0.64	1.05	0.81	**0.62**	
Freescale1	12.514	0.78	1.92	1.75	1.49	1.14	**0.70**	1.61	1.37	0.76	0.06	0.06	0.78	1.49	1.14	**0.70**	
image_interp	0.712	0.83	0.86	0.82	1.43	1.42	**0.73**	2.28	1.23	0.89	0.06	0.12	0.82	1.43	1.23	**0.73**	
kneser_10_4_1	2.455	0.52	0.99	0.63	1.66	0.73	**0.48**	1.64	1.04	0.73	0.07	0.13	0.52	1.64	0.73	**0.48**	
LargeRegFile	18.158	0.96	2.55	1.15	8.15	8.55	5.74	0.21	0.39	**0.17**	0.74	0.01	0.96	0.21	0.39	**0.17**	
lp_osa_14	0.949	0.97	2.95	1.79	0.64	0.68	**0.59**	20.42	17.46	11.38	0.64	0.96	0.97	0.64	0.68	**0.59**	
lp_osa_30	1.637	1.01	4.61	1.37	0.55	0.55	**0.48**	18.49	11.59	11.65	0.61	0.96	1.00	0.55	0.55	**0.48**	
lung2	0.417	1.08	0.96	1.06	1.62	2.34	**0.88**	1.78	2.36	1.04	0.01	0.01	0.96	1.62	2.34	**0.88**	
memchip	7.559	1.04	2.31	2.27	1.61	1.52	**0.94**	1.71	1.83	1.03	0.07	0.08	1.00	1.61	1.52	**0.94**	
neos	1.769	0.96	3.47	2.00	5.01	3.61	3.02	1.18	1.39	**0.76**	0.51	0.04	0.96	1.18	1.39	**0.76**	
ohne2	3.637	**0.77**	1.38	0.87	2.11	1.50	1.40	2.24	1.62	1.48	0.98	0.98	**0.77**	2.11	1.50	1.40	
para-6	2.824	**0.70**	1.48	0.86	2.12	1.50	1.42	2.23	1.61	1.62	0.88	0.88	**0.70**	2.12	1.50	1.42	
pds-100	1.578	0.64	1.13	0.82	1.05	1.00	**0.56**	1.89	1.49	0.75	0.25	0.16	0.64	1.05	1.00	**0.56**	
pds-80	1.565	0.63	1.02	0.72	0.91	0.82	**0.58**	1.65	1.49	0.75	0.26	0.16	0.63	0.91	0.82	**0.58**	
scircuit	1.141	0.61	1.08	0.63	1.72	1.06	**0.58**	1.83	1.14	0.67	0.08	0.08	0.61	1.72	1.06	**0.58**	
sgpf5y6	1.605	0.50	0.63	1.12	2.94	0.98	**0.41**	0.65	1.08	0.59	0.06	0.16	0.50	0.65	0.98	**0.41**	
Stanford	6.992	0.26	1.12	0.73	1.40	0.32	**0.20**	2.77	1.84	2.37	0.06	0.49	0.26	1.40	0.32	**0.20**	
Stanford_Berkeley	4.113	1.01	1.60	1.90	1.03	1.17	**0.76**	5.00	8.96	8.20	0.08	0.59	1.00	1.03	1.17	**0.76**	
stat96v1	0.387	1.07	1.17	1.17	1.93	1.73	**0.88**	1.93	2.87	1.19	0.29	0.03	1.00	1.93	1.73	**0.88**	
stat96v2	1.335	1.05	1.40	1.30	1.24	1.43	**0.83**	2.33	3.20	1.18	0.28	0.03	1.00	1.24	1.43	**0.83**	
watson_2	1.711	1.01	1.02	1.41	2.23	1.46	**0.76**	1.41	2.28	0.78	0.05	0.08	1.00	1.41	1.46	**0.76**	
web-BerkStan	4.124	1.02	1.62	2.23	4.55	10.36	7.94	1.12	1.26	**0.83**	0.59	0.08	1.00	1.12	1.26	**0.83**	
web-Stanford	6.796	0.27	1.16	0.66	2.66	2.06	1.91	1.47	0.35	**0.25**	0.49	0.06	0.27	1.47	0.35	**0.25**	
wheel_601	4.290	0.51	0.91	1.60	1.69	0.88	**0.36**	1.58	1.00	1.06	0.03	0.43	0.51	1.58	0.88	**0.36**	
Average	–	0.76	1.42	1.16	1.61	1.28	0.81	2.09	1.88	1.20	0.17	0.14	0.74	1.16	0.96	0.58	

a new row/column slice until the size of the slice exceeds the cache size threshold.

The SpMV operation associated with each row/column slice is treated as an atomic task that will be executed by a thread on a single core of the Xeon Phi architecture. This scheme in general obtains fine-grained tasks so that OpenMP scheduler can easily maintain balance on computational loads of threads using dynamic scheduling. This scheme also enables CRp to satisfy quality criteria (a) and (b).

The partitioning scheme described above for the baseline algorithms is not used for MKL. Instead, the whole A or A^T matrix is given as input to `mkl_dcsrmv`, which utilizes rowwise parallelization. The latter scheme is preferred because of the dramatic performance degradation experimentally observed for the former scheme due to calling `mkl_dcsrmv` for each individual row/column slice. The CSB library [21] is not utilized as another baseline algorithm since it is experimentally observed that the CSB library does not perform well on Xeon Phi when the latter scheme is adopted.

The proposed *sb*CRp and *sb*RCp algorithms are implemented as described in Sections 3.2.1 and 3.2.2, respectively. The matrices are permuted into SB forms using the HP-based method, as described in Section 3.3. The HP tool PaToH is used with the `PATOH_SUGPARAM_SPEED` parameter, which is reported in [22] as producing reasonably good partitions faster than the default parameter. The allowed maximum imbalance ratio is set to 10 percent. K is selected according to the cache size threshold, as in the baseline algorithms. Since PaToH utilizes randomized algorithms, each matrix is permuted into rowwise and columnwise SB forms ten times for the *sb*CRp and *sb*RCp methods, respectively; and average performance results are reported in Table 5 and Fig. 4, given in Section 4.3.

The experiments are carried out on a system with an Intel Xeon Phi P5110 coprocessor. The coprocessor is used in offload mode so 59 out of 60 cores are used. The Level 2 (L2) cache of each core is private to the core, eight-way set associative, and 512 KB in size. Each core can handle up to four hardware threads, so the effective cache size per thread can be estimated as 128 KB when four threads run on the same core. In the experiments, the cache size threshold utilized by the partitioning algorithms for the baseline and proposed methods is selected as 64 KB, which is half of the effective cache size per thread.

All of the proposed and baseline parallel $\mathrm{SpMM^TV}$ algorithms are implemented in C using OpenMP and compiled with `icc` (Intel's C Compiler) version 13.1.3 with the `O1` optimization flag. Double precision arithmetic is used. In Table 5, the best performance result for 59, 118, 177, and 236 threads are reported; and in Fig. 4, performance results for

1, 10, 20, 30, 40, 59, 118, 177, and 236 threads are reported. We utilize dynamic scheduling with chunksizes 1, 2, 4, and 8; and the best results are given. Environment variables for the coprocessor are set as follows: `KMP_AFFINITY=granularity=fine,balanced` to prevent OpenMP threads from floating between different thread contexts and to provide balanced assignments of threads to cores; and `MKL_DYNAMIC=false` for forcing the library to not automatically determine and change the number of threads [20]. For each $\mathrm{SpMM^TV}$ computation, we report the average execution time of 1,000 iterations after performing 10 iterations as a warm-up. OpenMP's `atomic` construct is used for handling concurrent writes.

4.3 Performance Evaluation

Table 5 displays the running times of the $\mathrm{SpMM^TV}$ algorithms on the Xeon Phi processor for the 28 test matrices given in Table 4. In the table, running times of *org*RRp are given in terms of milliseconds, whereas running times of all other algorithms are displayed as normalized values. Each normalized value is calculated through dividing the running time of the respective algorithm for a given matrix by that of the *org*RRp algorithm for the same matrix. The last row of Table 5 displays the average of the normalized running times of each algorithm over all test matrices. The averages are computed using the geometric mean. Note that the average normalized running time of *org*RRp is effectively 1.00. In each row of the table, a bold value indicates the minimum normalized running time attained for the respective matrix.

Table 5 also displays the normalized border sizes of the SB forms of the test matrices in the "Border size" columns. For the rowwise SB form A_{rSB} of a given matrix, a normalized value is calculated through dividing the number of rows in the border by the total number of rows of the same matrix. Similarly, for the columnwise SB form A_{cSB} of a given matrix, a normalized value is calculated through dividing the number of columns in the border by the total number of columns of the same matrix.

As seen in Table 5, RCM ordering substantially improves the running times of all baseline algorithms. On average, RCM ordering respectively improves the running times of RRp, MKL, CRp, and RCp by 24, 18.3, 20.5, and 10 percent. These experimental findings show the validity of quality criteria (c) and (d) (temporal locality in SpMV operations) on the performance of $\mathrm{SpMM^TV}$.

Although both *rcm*CRp and *sb*CRp satisfy quality criteria (c) and (d), as seen in Table 2, *sb*CRp performs 36.7 percent faster than *rcm*CRp on average, as seen in Table 5. This significant performance improvement of *sb*CRp over *rcm*CRp mainly stems from the topological property of the SB form, which minimizes the number of costly concurrent writes. A similar discussion holds for the performance difference between *rcm*RCp and *sb*RCp. These experimental findings show the importance of quality criterion (e) on minimizing concurrent writes for increasing the performance of parallel $\mathrm{SpMM^TV}$ operations.

As seen in Table 5, out of the 28 test matrices, the proposed methods attain the highest performance for 26 test matrices, whereas *rcm*RRp attains the highest performance for only two matrices. These results can be attributed to the quality of the SB forms of these two matrices, `ohne2` and `para-6`. An SB form of a given matrix is said to be "good" if its border submatrix is small. Good rowwise SB forms do not exist for matrices that have many dense columns. Good columnwise SB forms do not exist for matrices that have many dense rows. Neither good rowwise nor good columnwise SB forms exist for matrices that have both many dense columns and rows. Note that the number of nonzeros in a dense column/row of a given matrix already establishes a lower bound on the size of the row/column border of the rowwise/columnwise SB form. As seen in Table 4, `ohne2` and `para-6` have dense columns and dense rows and a relatively large average number of nonzeros per column and row. As also seen in the "Border size" column of Table 5, 98 percent of `ohne2`'s rows and 88 percent of `para-6`'s rows constitute the row borders of their respective rowwise SB forms; and the same percents of columns constitute the column borders of their respective columnwise SB forms. These consequences explain the inferior performance of the SB-based methods for these two matrices.

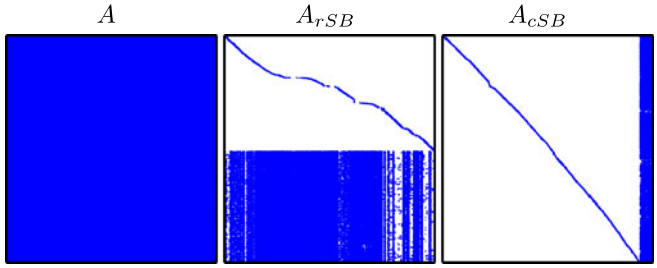


Fig. 3. The `web-Stanford` matrix A and its rowwise and columnwise SB forms A_{rSB} and A_{cSB}.

As seen in Table 5, out of the 26 test matrices for which SB-based methods show superior performance, *sb*CRp attains the highest performance for 22 test matrices. The inferior performance of *sb*RCp is already expected since it incurs an additional synchronization point, as seen in Algorithm 2 and partially satisfies quality criteria (a) and (b), as shown in Table 2. Despite these disadvantages, *sb*RCp attains the highest performance for the four matrices `LargeRegFile`, `neos`, `web-BerkStan`, and `web-Stanford`. These matrices have dense columns but not dense rows. As also seen in Table 5, the normalized border sizes of the columnwise SB forms of these matrices are significantly smaller than those of their rowwise SB forms. As an example, Fig. 3 displays the significant quality difference between rowwise and columnwise SB forms of the `web-Stanford` matrix.

As discussed above, despite the advantages of *sb*CRp over *sb*RCp, *sb*CRp may show inferior performance for some matrices. For this reason, it will be more meaningful to compare the best results obtained by *sb*CRp and *sb*RCp against RRp. In Table 5, "The-best-of-CRp/RCp-SB" column displays the minimum of the running times of *sb*CRp and *sb*RCp for each matrix. For the sake of a fair comparison, a similar "Best-of" approach is adopted for all baseline algorithms. For each matrix, the minimum of the running times of *org*RRp, *rcm*RRp, *org*MKL, and *rcm*MKL is displayed in the "RRp" column, the minimum of the running times of *org*CRp and *org*RCp is displayed in the "org." column, and the minimum of the running times of *rcm*CRp and *rcm*RCp is displayed in the "RCM" column.

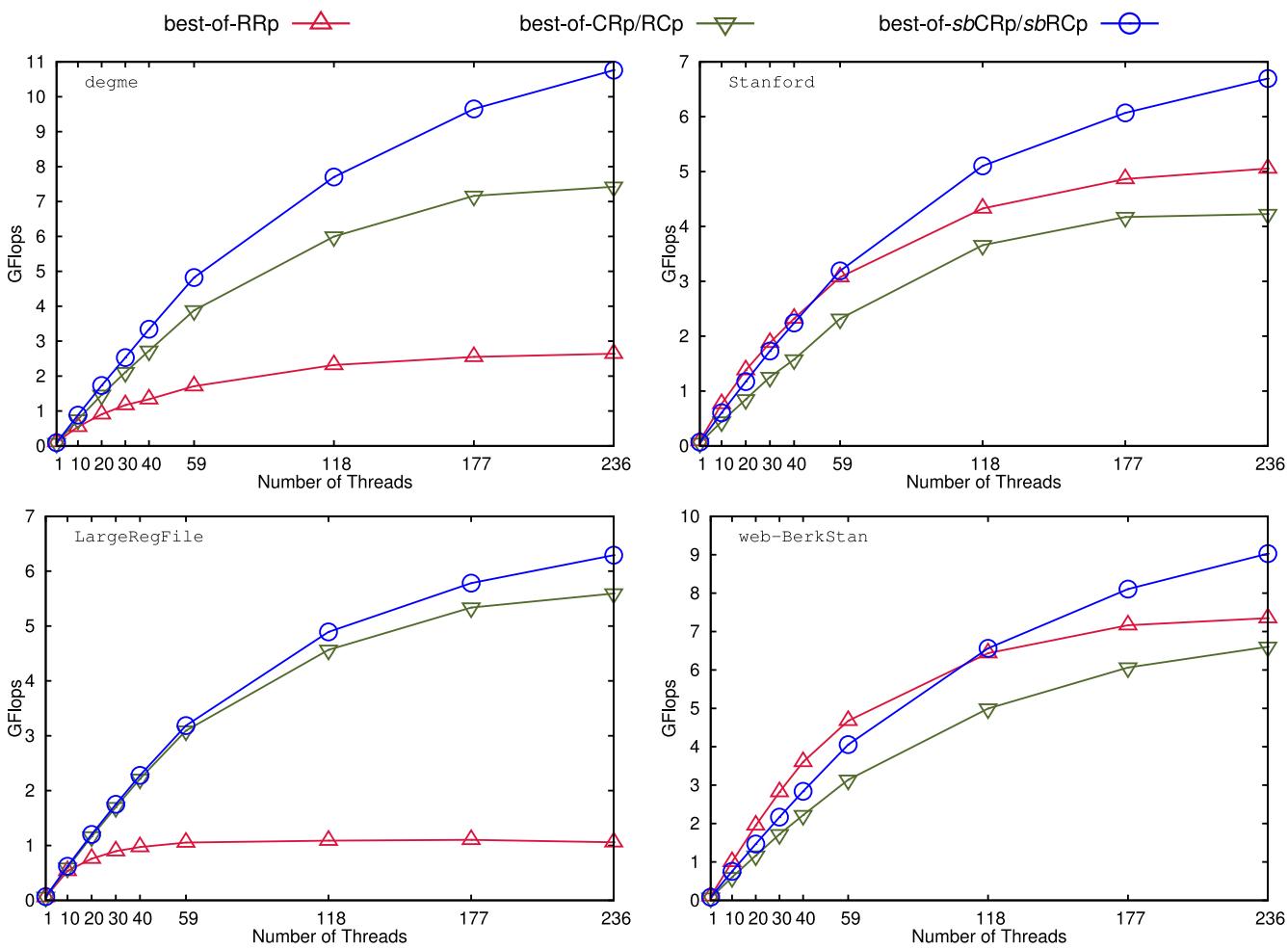


Fig. 4. Speedup curves for four test matrices: `degme`, `Stanford`, `LargeRegFile`, and `web-BerkStan`.

As seen in Table 5, reporting the best performance results of the CRp/RCp algorithms significantly improves the average normalized performances of the original, RCM, and SB schemes from 1.61/2.09 to 1.16, from 1.28/1.88 to 0.96, and from 0.81/1.20 to 0.58, respectively. Considering that the best-of-RRp achieves the average normalized performance of 0.74, these results show that neither the original nor RCM ordering of the best-of-CRp/RCp can attain better average performance than the best-of-RRp, whereas the best-of-*sb*CRp/*sb*RCp attains significantly better average performance (22 percent better) than the best-of-RRp. These results in turn show the validity of the proposed SB approach in the *sb*CRp and *sb*RCp methods.

There are two distinct approaches for deciding on the best of the *sb*CRp and *sb*RCp algorithms for a given matrix. The first is to run both algorithms for a few iterations and decide on the best one, as adopted in OSKI's offline optimization [23]. Although this scheme works fine in practice, it suffers from doubling the preprocessing overhead due to row/column reordering and partitioning. The second is to devise a simple recipe based on analyzing the sparsity pattern of matrix A, which is beyond the scope of this work.

Fig. 4 shows the strong scaling results as speedup curves for the best-of-RRp, best-of-CRp/RCp, and best-of-*sb*CRp/*sb*RCp schemes on four matrices in terms of giga flops per second (GFlops). As seen in the figure, the proposed best-of-*sb*CRp/*sb*RCp scheme shows good scalability and outperform the baseline best-of-RRp and best-of-CRp/RCp schemes.

The preprocessing overhead of the methods is also investigated. For each matrix, PaToH's running time on the host system is divided by the best $\mathrm{SpMM^TV}$ time on the Xeon Phi processor. On the average, the overhead of the best of *sb*CRp and *sb*RCp is 1,121 kernel invocations, whereas the overhead of the RCM algorithm [24] is 84 invocations. The relatively high overhead of the proposed sophisticated methods over the simple RCM algorithm are expected to amortize for large number of repeated $\mathrm{SpMM^TV}$ computations that involve matrix A with the same sparsity pattern.

5 Related Work

5.1 SpMV

For thread-level parallelism of SpMV on the recently-released Xeon Phi processor, Cramer et al. [25] experiment with the conjugate gradient method, which involves SpMV, and analyze the experimental results on the Xeon Phi architecture according to the Roofline model [10].

Saule et al. [12] evaluate the performance of SpMV with multiple dense right-hand-side vectors (SpMM) on the Xeon Phi. They also investigate the performance effect of row/column reordering for exploiting cache locality via utilizing the widely used bandwidth-reduction method, RCM.

Liu et al. [13] use the ELLPACK Sparse Block (ESB) format for SpMV operation on the Xeon Phi. The ESB format uses bitmasks for storing the sparsity pattern and sorts rows within blocks according to their nonzero counts – instead of sorting whole matrix – with the purpose of increasing the nonzero density of the columns of compressed row blocks, as well as reducing disturbance of the input-vector locality provided by the original matrix. The column blocks are multiplied in parallel, so thread-local temporary output vectors are used and hence a reduction operation is performed after the computation. The authors use three schedulers for load balancing: partitioning based on cache-miss simulation; hybrid scheduling consisting of sharing and stealing tasks; and 1D partitioning [26] of rows according to their computational loads, which are determined via executing SpMV operations.

Sarıyüce et al. [27] use an SpMM-based formulation for the closeness centrality of a given graph to fully utilize the vector units of Xeon processors and the Xeon Phi architecture. The SpMM-based formulation is obtained via processing multiple vertices of the graph in simultaneous breadth-first search operations.

Pichel and Rivera [28] analyze the effects of SpMV optimization techniques (reordering, blocking, and compression) on an experimental processor with 48 cores connected through a mesh network. They report that the subject architecture benefits considerably from locality improvements in parallel SpMV computations.

Park et al. [29] run the newly established high-performance conjugate gradient (HPCG) benchmark [30] on a Xeon Phi cluster. They apply optimization techniques such as task scheduling and matrix reordering.

Kreutzer et al. [11] show that a unified storage format can perform ideally for a wide range of matrix types involved in parallel SpMV operation on various architectures, including CPU, GPU, and Xeon Phi.

5.2 SpMM$^{\mathrm{T}}$V

For improving performance of the sequential SpMM$^{\mathrm{T}}$V operation, Vuduc et al. [31] propose reusing A-matrix nonzeros. The intermediate subvector z_k is computed as $z_k \leftarrow C_k^T\ x$, using nonzeros of the kth column slice C_k, then partial results for output vector y is computed as $y \leftarrow C_k\ z_k$, using the same nonzeros. A naive parallelization of this scheme (our CRp scheme) results in concurrent writes to the whole output vector, so scalability of this parallel algorithm is limited.

For the parallel SpMM$^{\mathrm{T}}$V operation on many-core processors, Buluç et al. [21] propose a blocking method called Compressed Sparse Blocks (CSB) with a scheduling algorithm for dynamically assigning the blocks to threads. Using the same data structure for performing both $y \leftarrow A\ z$ and $z \leftarrow A^T x$ with no performance degradation is non-trivial, and this is successfully achieved by CSB via using a triple for each nonzero and using indices relative to the block. Their method performs $y \leftarrow AA^T x$ as two separate row-parallel SpMV s (with no A-matrix nonzero reuse) and recursively divides row-slices into blocks, yielding a two-dimensional parallelization scheme for reducing load imbalance due to irregularity in sparsity pattern of the input matrix. Their method handles concurrent writes via using temporary arrays for each thread that contributes to the same output subvector. Although the CSB scheme uses a single storage of matrix A, authors' multiplication algorithm cannot simultaneously perform $y \leftarrow A\ z$ and $z \leftarrow A^T x$. Our proposed methods perform these two SpMV operations simultaneously, and hence satisfy the quality criterion of reusing A-matrix nonzeros as well as other quality criteria via casting the efficient parallelization of SpMM$^{\mathrm{T}}$V as a combinatorial optimization problem. The CSB scheme can also be integrated into our proposed methods for handling diagonal blocks.

Yang et al. [8] propose a tiling-based method to increase data reuse in GPUs for data mining algorithms such as PageRank, HITS, and Random Walk with Restart. These data mining algorithms utilize SpMV operations that involve sparse matrices representing power-law graphs. In [8], A and A^T are stored separately, i.e., A-matrix nonzeros are not reused. One of the baseline SpMV algorithms assigns each row to a thread, which in turn corresponds to our RRp algorithm with a row-level thread assignment.

Martone [32] compares the performance of Recursive Sparse Blocks (RSB) format [33], which is based on space filling curves, against those of MKL and CSB [21] for SpMV and SpMM$^{\mathrm{T}}$V operations. Although a single storage of matrix A is used in [32], SpMV and SpM$^{\mathrm{T}}$V operations are performed separately, without any nonzero reuse. The scheduling algorithm in [32] does not assign submatrices that will write to the subvector, which is currently being written by another thread. It is reported in [32] that this scheduling algorithm is not scalable because critical sections are used in assigning blocks to threads. The use of space-filling curves is expected to further increase data locality while multiplying each block in our proposed SpMM$^{\mathrm{T}}$V algorithms.

The above-mentioned works on parallel SpMV and SpMM$^{\mathrm{T}}$V generally aim to increase utilization of vector units, and some also provide algorithmic contributions. Although one must use vectorization to efficiently use vector units, vectorization provided by compilers or hand-tuned code is not used in the current work because its scope is mainly to achieve data reuse. Data structures and kernels based on vectorization can also benefit such locality improvements.

Our work differs from the above-mentioned works on SpMM$^{\mathrm{T}}$V in that we encode the efficient parallelization of SpMM$^{\mathrm{T}}$V as a combinatorial optimization problem via using the identified five quality criteria. None of the existing parallel SpMM$^{\mathrm{T}}$V algorithms can reuse matrix nonzeros. Additionally, to our knowledge, there is no previous study on parallel SpMM$^{\mathrm{T}}$V algorithms for the Xeon Phi processor.

6 Conclusion

For cache-coherent many-core processors, we presented a parallel sparse matrix-vector and matrix-transpose-vector multiplication (SpMM$^{\mathrm{T}}$V) framework based on one-dimensional matrix partitioning, and identified five quality criteria that affect performance. In various iterative methods, SpMM$^{\mathrm{T}}$V operations are repeatedly performed

as consecutive sparse matrix-vector and sparse matrix-transpose-vector multiplication operations on the same matrix. We proposed two novel methods based on permuting sparse matrices into singly bordered block-diagonal forms. The two SB-based methods simultaneously achieve reducing the memory bandwidth requirement of $\mathrm{SpMM^TV}$ operations via utilizing data reuse opportunities and minimizing the number of concurrent writes.

We tested the validity of the identified quality criteria and the proposed methods within the framework on a wide range of sparse matrices. Experiments on a 60-core cache-coherent Intel Xeon Phi processor verified the validity of the proposed framework through showing the performance improvements from data reuse opportunities on processors with many cores and complex cache-coherency protocols. The experiments also showed that reusing matrix nonzeros compensates for the overhead of concurrent writes through the proposed SB-based methods.

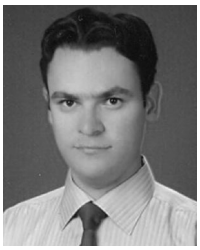
Acknowledgments

This work was partially supported by the PRACE 4IP project funded in part by Horizon 2020 The EU Framework Programme for Research and Innovation (2014-2020) under grant agreement number 653838.

References

[1] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proc. 16th Annu. ACM Symp. Theory Comput.*, 1984, pp. 302–311.
[2] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM J. Optimization*, vol. 2, no. 4, pp. 575–601, 1992.
[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.
[4] C. C. Paige and M. A. Saunders, "LSQR: An algorithm for sparse linear equations and sparse least squares," *ACM Trans. Math. Softw.*, vol. 8, no. 1, pp. 43–71, 1982.
[5] K. Yang and K. G. Murty, "New iterative methods for linear inequalities," *J. Optimization Theory Appl.*, vol. 72, no. 1, pp. 163–185, 1992.
[6] B. Uçar, C. Aykanat, M. Ç. Pınar, and T. Malas, "Parallel image restoration using surrogate constraint methods," *J. Parallel Distrib. Comput.*, vol. 67, no. 2, pp. 186–204, 2007.
[7] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, 1999.
[8] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proc. VLDB Endowment*, vol. 4, no. 4, pp. 231–242, Jan. 2011.
[9] T.-Y. Chen and J. W. Demmel, "Balancing sparse matrices for computing eigenvalues," *Linear Algebra Appl.*, vol. 309, no. 13, pp. 261–287, 2000.
[10] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[11] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, "A unified sparse matrix data format for efficient general sparse Matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.
[12] E. Saule, K. Kaya, and U. V. Catalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Proc. 10th Int. Conf. Parallel Process. Appl. Math.*, 2014, pp. 559–570.
[13] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. Int. Conf. Supercomput.*, 2013, pp. 273–282.
[14] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph partitioning based models and methods for exploiting cache locality in sparse Matrix-vector multiplication," *SIAM J. Sci. Comput.*, vol. 35, no. 3, pp. C237–C262, 2013.
[15] C. Aykanat, A. Pinar, and U. V. Catalyurek, "Permuting sparse rectangular matrices into Block-diagonal form," *SIAM J. Sci. Comput.*, vol. 25, pp. 1860–1879, 2002.
[16] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.
[17] M. Krotkiewski and M. Dabrowski, "Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs," *Parallel Comput.*, vol. 36, no. 4, pp. 181–198, 2010.
[18] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.
[19] G. M. D. Corso, A. Gull, and F. Romani, "Comparison of Krylov subspace methods on the PageRank problem," *J. Comput. Appl. Math.*, vol. 210, no. 12, pp. 159–166, 2007.
[20] (2015). MKL [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl/
[21] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Symp. Parallelism Algorithms Archit.*, 2009, pp. 233–244.
[22] U. V. Catalyürek and C. Aykanat, "PaToH: A multilevel hypergraph partitioning tool, version 3.0," Dept. Comput. Eng., Bilkent University, Ankara, 1999.
[23] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *J. Phys.: Conf. Series*, vol. 16, no. 1, p. 521, 2005.
[24] J. Burkardt. (2003). Reverse Cuthill McKee ordering [Online]. Available: http://people.sc.fsu.edu/ ~jburkardt/cpp_src/rcm/rcm.html, 2003.
[25] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison," in *Proc. Many-Core Appl. Res. Community Symp. RWTH Aachen Univ.*, pp. 38–44, 2012.
[26] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distrib. Comput.*, vol. 64, no. 8, pp. 974–996, 2004.
[27] A. E. Sarıyüce, E. Saule, K. Kaya, and U. V. Catalyürek, "Hardware/software vectorization for closeness centrality on multi-/many-core architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 1386–1395.
[28] J. C. Pichel and F. F. Rivera, "Sparse Matrix–vector multiplication on the Single-chip cloud computer many-core processor," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1539–1550, 2013.
[29] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 945–955.
[30] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia Rep., SAND2013-4744, vol. 312, 2013.
[31] R. Vuduc, A. Gyulassy, J. Demmel, and K. Yelick, *Memory Hierarchy Optimizations and Performance Bounds for Sparse* $\mathrm{A^T}$ Ax, ser. Lecture Notes in Computer Science, P. Sloot, D. Abramson, A. Bogdanov, Y. Gorbachev, J. Dongarra, and A. Zomaya, Eds. Springer Berlin Heidelberg, 2003, vol. 2659.
[32] M. Martone, "Efficient multithreaded untransposed, transposed or symmetric sparse Matrix-vector multiplication with the recursive sparse blocks format," *Parallel Comput.*, vol. 40, no. 7, pp. 251–270, 2014.
[33] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication- preliminary considerations," in *Proc. 25th Int. Conf. Comput. Their Appl.*, 2010, pp. 300–305.

M. Ozan Karsavuran received the BS and MS degrees in 2012 and 2014, respectively, in computer engineering from Bilkent University, Ankara, Turkey, where he is currently working toward the PhD degree. His research interests include parallel computing, cache-aware methods, and high-performance computing.



Kadir Akbudak received the BS and MS degrees in 2007 and 2009, respectively, in computer engineering from Hacettepe and Bilkent Universities, Ankara, Turkey. He is currently working toward the PhD degree at the Computer Engineering Department, Bilkent University. His research interests include locality-aware partitioning and scheduling methods for exascaling irregular applications, cache-locality exploiting methods for scientific computational kernels.

Cevdet Aykanat received the BS and MS degrees from Middle East Technical University, Ankara, Turkey, both in electrical engineering, and the PhD degree from Ohio State University, Columbus, in electrical and computer engineering. He was at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects. (co)authored about 80 articles published in academic journals indexed in ISI and his publications received above 700 citations in ISI indexes. He received the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as a member of IFIP Working Group 10.3 (Concurrent System Technology) since 2004 and as an associate editor of *IEEE Transactions of Parallel and Distributed Systems* between 2009 and 2013.