Cartesian Partitioning Models for 2D and 3D Parallel SpGEMM Algorithms

Gunduz Vehbi Demirci and Cevdet Aykanat

Abstract—The focus is distributed-memory parallelization of sparse-general-matrix-multiplication (SpGEMM). Parallel SpGEMM algorithms are classified under one-dimensional (1D), 2D, and 3D categories denoting the number of dimensions by which the 3D sparse workcube representing the iteration space of SpGEMM is partitioned. Recently proposed successful 2D- and 3D-parallel SpGEMM algorithms benefit from upper bounds on communication overheads enforced by 2D and 3D cartesian partitioning of the workcube on 2D and 3D virtual processor grids, respectively. However, these methods are based on random cartesian partitioning and do not utilize sparsity patterns of SpGEMM instances for reducing the communication overheads. We propose hypergraph models for 2D and 3D cartesian partitioning of the workcube for further reducing the communication overheads of these 2D- and 3D- parallel SpGEMM algorithms. The proposed models utilize two- and three-phase partitioning that exploit multi-constraint hypergraph partitioning formulations. Extensive experimentation performed on 20 SpGEMM instances by using upto 900 processors demonstrate that proposed partitioning models significantly improve the scalability of 2D and 3D algorithms. For example, in 2D-parallel SpGEMM algorithm on 900 processors, the proposed partitioning model respectively achieves 85 and 42 percent decrease in total volume and total number of messages, leading to 1.63 times higher speedup compared to random partitioning, on average.

Index Terms—Sparse matrix-matrix multiplication, SpGEMM, sparse SUMMA SpGEMM, split-3D-SpGEMM, hypergraph partitioning, communication cost, bandwidth, latency

1 Introduction

Sparse general matrix multiplication (SpGEMM) of the form $C = AB$ is a kernel operation in many scientific computing applications such as finite element simulations [1], molecular dynamics [2], [3], linear programming (LP) [4], [5] and linear solvers [6], [7]. Additionally, SpGEMM is also utilized in high-performance graph computations such as graph contraction [8], betweenness centrality computation [9], Markov clustering [10], triangle counting [11] and graph traversal [12].

Extensive research is made for parallelizing SpGEMM on shared memory [13], [14], [15], [16], [17] and distributed memory [18], [19], [20] architectures. There also exist several works that propose graph/hypergraph partitioning models for improving the performance of the parallel SpGEMM [13], [21], [22], [23]. These works utilize the sparsity structure of the matrices for reducing the communication overhead of the parallel SpGEMM algorithms. The proposed graph/hypergraph partitioning models incur preprocessing overhead. Hence, applications, that involve repeated SpGEMM, in which the sparsity patterns of input matrices remain the same, benefit more from these models. As discussed in [21], [22], similarity join [24], collaborative filtering [25] and interior point methods used for solving linear-programming problems [4], [5], [26] constitute such applications.

• *The authors are with the Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey.*
E-mail: {gunduz.demirci, aykanat}@cs.bilkent.edu.tr.

Manuscript received 29 Oct. 2019; revised 3 May 2020; accepted 1 June 2020. Date of publication 8 June 2020; date of current version 30 June 2020.
(Corresponding author: Cevdet Aykanat.)
Recommended for acceptance by P. Balaji.
Digital Object Identifier no. 10.1109/TPDS.2020.3000708

Iteration space of SpGEMM operation can be visualized as a sparse three-dimesnsional (3D) cube (workcube) and parallel SpGEMM algorithms are categorized according to the partitioning of this workcube [19]. In this categorization, 1D, 2D and 3D algorithms are defined according to the number of dimensions by which the workcube is partitioned. There exist efficient implementations of 1D-, 2D- and 3D-parallel algorithms in literature [18], [19], [21]. An important drawback of 1D-parallel algorithms is that these algorithms face communication bottlenecks, since the volume/number of messages handled by processors may drastically increase with increasing number of processors because of dense rows and/or columns of the input matrices.

The drawbacks of 1D-parallel SpGEMM algorithms can be significantly reduced by utilizing additional dimensions in processor grids and partitioning the iteration space along multiple dimensions. Successful 2D- and 3D-parallel algorithms are recently proposed (e.g., [18], [19]). These multi-dimensional algorithms benefit from nice upper bounds enforced on the communications requirements of processors by 2D and 3D cartesian partitioning of the workcube on 2D and 3D virtual processor grids, respectively. However, these methods are based on random cartesian partitioning and hence do not utilize sparsity patterns of SpGEMM instances for reducing the communication overheads.

We fill this literature gap by proposing hypergraph partitioning models for improving the performance of 2D-parallel [18] and 3D-parallel [19] SpGEMM algorithms. The proposed hypergraph models attain 2D and 3D cartesian partitioning of the workcube through two- and three-phase partitioning frameworks, respectively. The proposed models utilize multi-constraint partitioning formulations for encoding computational load-balancing within the multi-phase

partitioning framework. In the first and second phases of both 2D and 3D partitioning models, the partitioning objective of minimizing cutsize encodes the minimization of total volume during the communication of B-matrix rows and A-matrix columns, respectively. In the third phase of 3D partitioning model, minimizing cutsize encodes the minimization of total volume during the communication of partial results for C-matrix nonzeros. Hence, the proposed models further improve the communication performances of the successful 2D- and 3D-parallel SpGEMM algorithms.

The partitioning objective of the proposed models also encode the minimization of local memory requirements of processors due to communication buffers. This enables all processors to concurrently perform single-stage sparse communications at each phase, whereas existing 2D- and 3D-parallel implementations use multi-stage communications at each phase. We utilize this fact to develop new efficient 2D- and 3D-parallel SpGEMM implementations.

We conduct extensive experiments to evaluate our partitioning models on 20 SpGEMM instances arising from real-world applications. Experimental results demonstrate that our models provide significant improvements for the algorithms given in [18], [19] and improve these algorithms' scalability and efficiency on real-world datasets. For 2D-parallel SpGEMM on 900 processors, the proposed partitioning model respectively achieves 85 and 42 percent decrease in total volume and total number of messages, leading to 1.63x higher speedup compared to random partitioning, on average. For 3D-parallel SpGEMM on 900 processors, these improvements respectively become 62 and 31 percent, leading to 1.31x higher speedup.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes the 2D- and 3D-parallel SpGEMM algorithms. Sections 4.2 and 4.3 describe our proposed hypergraph models for 2D and 3D cartesian partitioning of the workcube. Section 5 presents experimental results. Section 6 concludes the paper. In the supplemental material, Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.3000708, contains sensitivity experiments on multi-constraint partitioning quality, Appendix B, available in the online supplemental material, discusses partitioning overhead and amortization and Appendix C, available in the online supplemental material, contains speedup curves for 20 realistic instances.

2 Related Work

Parallel SpGEMM algorithms for various shared-memory parallel architectures are extensively studied in the literature. Intel's math kernel library (MKL) [27] provides an efficient SpGEMM implementation. Patwary *et al*. [28] studies various partitioning and cache optimization techniques which significantly improve the performance of MKL. Akbudak and Aykanat [13] propose hypergraph and bipartite graph partitioning models to exploit spatial and temporal locality in row-by-row parallel SpGEMM on many core Intel Xeon Phi processor architecture. Studies considering GPU architectures also exist [14], [15], [16].

SpGEMM algorithms for distributed-memory systems are also studied extensively. Trilinos [29] and Combinatorial BLAS (CombBLAS) [9] are publicly available libraries which offer distributed-memory SpGEMM implementations. CombBLAS library includes an implementation of sparse SUMMA algorithm [18]. This algorithm operates on a 2D virtual processor grid and utilizes 2D block partitioning of input and output matrices. This algorithm utilizes a sequential SpGEMM kernel based on heap and doubly-compressed-sparse-column data structures to perform local computations. Azad *et al*. [19] propose an extension for sparse SUMMA algorithm by considering an additional third dimension in the virtual processor grid. Their algorithm also extends the sequential SpGEMM kernel in sparse SUMMA by considering multi-threaded execution. Theoretical lower bounds on communication costs of parallel SpGEMM algorithms are studied in [30], [31]. A survey of parallel SpGEMM algorithms and their theroetical analysis of expected communication costs on random matrices, is given in [32]. Sparsity-dependent communication lower bounds for parallel SpGEMM algorithms are given in [23].

Akbudak and Aykanat [22] consider an outer-product formulation for distributed SpGEMM and propose hypergraph-partitioning-based models to employ efficient task and data distribution. Hypergraph and bipartite graph partitioning models for outer-product, inner-product and row-by-row-product formulations for distributed SpGEMM are studied in [21]. Ballard *et al*. [23] propose a fine-grain hypergraph model. However, the fine-grain hypergraph model is presented as a theoretical approach [23] and it is found to be impractical [19], [21] due to the significantly large size of the hypergraph.

The graph/hypergraph models given in [13], [21], [22] partition the iteration space of SpGEMM along a single dimension and can be considered as 1D-parallel SpGEMM algorithms. The fine-grain model [23] represents each nonzero scalar multiplication as well as each nonzero entry in the input and output matrices by different vertices. Then, it achieves multi-dimensional partitioning on the 3D workcube by first coarsening the fine-grain hypergraph and then partitioning the coarsened hypergraph. Since the coarsened hypergraph is directly partitioned into the total number of processors, this partitioning model actually assumes a 1D virtual processor grid and does not exploit the nice upper bounds on communication overhead achieved by 2D- and 3D-parallel algorithms.

The proposed hypergraph partitioning models are significantly different from the fine-grain hypergraph model [23] as follows: The proposed models aim at 2D/3D cartesian workcube partitioning that matches the virtual processor-grid dimensions of 2D- and 3D-parallel SpGEMM algorithms in order to utilize the nice upper bounds on the communication requirements of those algorithms. For this purpose, the proposed models utilize multi-phase and multi-constraint partitioning framework. Multi-constraint formulation is utilized in order to maintain workload balance among processors within the multi-phase partitioning framework. Furthermore, in the proposed models, data partitioning is inferred from the task partitioning induced by workcube partitioning instead of introducing vertices for nonzero entries of input/output matrices. This enables our models to be much smaller than the fine-grain model.

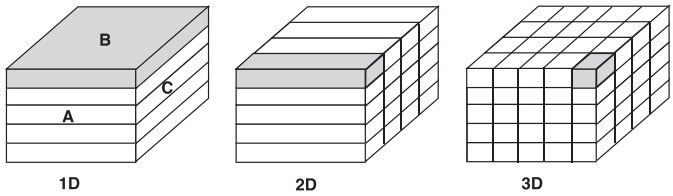

Fig. 1. 1D, 2D, and 3D cartesian workcube partitioning for 1D-, 2D-, and 3D-parallel SpGEMM algorithms on 5, 5×4, and $5\times5\times4$ processor grids. Gray shaded areas show a horizontal block, a fiber block, and a cuboid.

3 SpGEMM Algorithms

3.1 Workcube Representation

Given matrices $A\in\mathbb{R}^{m\times\ell}$, $B\in\mathbb{R}^{\ell\times n}$ and $C\in\mathbb{R}^{m\times n}$, iteration space of the SpGEMM operation $C=AB$ can be visualized as a sparse 3D cube (workcube) W of size $m\times\ell\times n$ [19], [23]. In W, each nontrivial scalar multiplication $A(i,k)B(k,j)$ (i.e., both $A(i,k)\neq0$ and $B(k,j)\neq0$) is represented by a voxel $W(i,j,k)$ whose projections onto A- and B-faces contain nonzero entries. Projections of these voxels onto C-face determine the nonzero pattern of matrix C. Subcubes of W obtained by fixing one and two indices are respectively called "layers" and "fibers". So, $W(i,:,:), W(:,j,:)$ and $W(:,:,k)$ denote the ith horizontal, jth frontal and kth lateral layers, respectively. Intersections of layers along different dimensions produce fibers which are denoted by $W(i,j,:)$, $W(i,:,k)$ and $W(:,j,k)$. For instance, the intersection of ith horizontal and jth frontal layers is fiber $W(i,j,:)$.

The voxels in the ith horizontal layer $W(i,:,:)$ represent computations using the nonzeros of the ith row of A and thus representing the task of computing ith row of C. The voxels in the jth frontal layer $W(:,j,:)$ represent computations using the jth column of B and thus representing the task of computing the jth column of C. The voxels in the kth lateral layer $W(:,:,k)$ represent computations associated with the outer product of the kth column of A with the kth row of B.

The middle part of Fig. 3 displays the workcube of the sample SpGEMM instance given at the top part. In the figure, A, B and C faces of the workcube represent the sparsity patterns of the respective matrices. The sparsity pattern of the workcube is shown by displaying individual voxels on the horizontal, frontal and lateral layers.

In [19], parallel SpGEMM algorithms are categorized according to the partitioning of the workcube among processors. In this categorization, 1D, 2D and 3D algorithms are defined with respect to the number of dimensions by which the workcube is partitioned (see Fig. 1). 1D, 2D and 3D algorithms perform communication on one, two and all three of the matrices, respectively. Communication on an input matrix refers to expand-type (i.e., multicast-type) of communication of the nonzero entries of the respective input matrix. Communication on the output matrix refers to fold-type (i.e., reduce-type) communication on the partial results produced by different processors for the same output matrix entries [21]. The following two subsections summarize 2D- and 3D-parallel SpGEMM algorithms for which we propose intelligent partitioning models.

3.2 2D: Sparse SUMMA Algorithm [18]

Sparse SUMMA performs multiplication $C=AB$ on a 2D virtual $p_x\times p_y=p$ processor grid. Let $P_{x,:}$ and $P_{:,y}$ respectively denote the processors in the xth processor-row and the yth processor-column of the 2D grid. So, $P_{x,y}$ denote the processor in the xth row and the yth column of the grid.

2D block partitioning is applied to A, B and C matrices in such a way that each matrix is partitioned rowwise among p_x processor-rows and columnwise among p_y processor-columns. The rows of A are partitioned conformably with the rows of C and the columns of B are partitioned conformably with the columns of C. Each processor $P_{x,y}$ stores submatrices $A_{x,y}$ and $B_{x,y}$ and computes submatrix $C_{x,y}$. Let m_x and ℓ_x respectively denote the number of A-/C-matrix rows and B-matrix rows assigned to the processors in processor-row $P(x,:)$. Also let n_y and ℓ_y respectively denote the number of B-/C-matrix columns and A-matrix columns assigned to the processors in the processor-column $P(:,y)$. Then, submatrices $A_{x,y}$, $B_{x,y}$ and $C_{x,y}$ are of sizes $m_x\times\ell_y$, $\ell_x\times n_y$ and $m_x\times n_y$, respectively.

With this data partitioning strategy, each processor $P_{x,y}$ needs all submatrices along the xth row block of A, which are stored in processor-row $P_{x,:}$. Processor $P_{x,y}$ also needs all submatrices along the yth column block of B, which are stored in processor-column $P_{:,y}$. To satisfy the input submatrix requirements of all processors for local SpGEMM computations, any processor $P_{x',y'}$ should broadcast its local submatrix $A_{x',y'}$ along the processors of processor-row $P_{x',:}$ as well as local submatrix $B_{x',y'}$ along processor-column $P_{:,y'}$. This 2D partitioning strategy has the nice property of providing an upper bound on the volume and the number of messages involved in these operations, since it confines these collective communication operations along the rows and columns of the processor grid.

The parallel sparse SUMMA algorithm proposed in [18] achieves the collective communication operations in stages in order to reduce the local memory requirements of the processors. Although this algorithm has the nice property of reducing processors' local memory requirement, it suffers from the increase in the latency overhead because of increased number of collective operations.

In this algorithm, the workcube is partitioned into $p_x\times p_y$ fiber blocks, where fiber block $W_{x,y,:}$ is of size $m_x\times\ell\times n_y$. Then each processor $P_{x,y}$ is responsible of computing the tasks/voxels in the fiber block $W_{x,y,:}$. In other words, the C-face of the workcube is decomposed into $p_x\times p_y$ 2D block submatrices, each denoted by $C_{x,y}$, where $C_{x,y}$ is locally computed by $P_{x,y}$. Fig. 2 displays a sample workcube partitioning and the corresponding matrix partitioning on a 2D grid of size $3\times4=12$.

3.3 3D: Split-3D-SpGEMM Algorithm [19]

Split-3D-SpGEMM algorithm [19] extends 2D algorithm [18] by considering a 3D $p_x\times p_y\times p_z$ virtual processor grid. Let $P_{x,:,:}$, $P_{:,y,:}$ and $P_{:,:,z}$ respectively denote the processors in the xth horizontal, yth frontal and zth lateral layers of the 3D grid. Also let $P_{x,y,:}$, $P_{x,:,z}$ and $P_{:,y,z}$ denote the processor-fibers in the intersections of the respective processor-layers. So $P_{x,y,z}$ denote the processor in the xth horizontal, yth frontal and zth lateral layers of the grid.

This algorithm relies on splitting the 2D blocks of 2D-partitioned A, B and C matrices into disjoint subblocks along different third dimensions. That is, 2D partitioning is applied

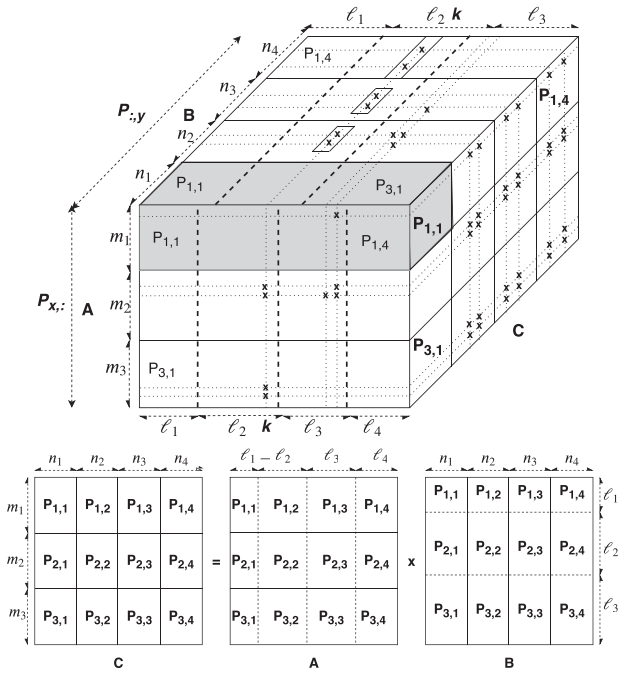

Fig. 2. Sparse Summa (2D) algorithm. Top: Partitioning of the workcube on a 3×4 2D grid. Bottom: Data distribution. Solid lines show that A-matrix rows and B-matrix columns are partitioned conformably with the workcube/task partition. Dotted lines show that B-matrix rows and A-matrix columns are partitioned independent from the task partitioning."x" denotes a nonzero entry in the respective matrix.

on A, B and C matrices among $p_x\times p_z$, $p_z\times p_y$ and $p_x\times p_y$ processors in the yth frontal layer $P_{:,y,:}$, the xth horizontal layer $P_{x,:,:}$ and the zth lateral layer $P_{:,:,z}$ of the grid, respectively. Then, each 2D block of A, B and C matrices are split into subblocks among p_y, p_x and p_z processors along the y, x and z dimensions, respectively. For example, each 2D block $A_{x,z}$ of A is split into p_y subblocks $A_{x,1,z}, A_{x,2,z}, \ldots, A_{x,y,z}$ which are respectively assigned to processors $P_{x,1,z}, P_{x,2,z}, \ldots, P_{x,y,z}$ of processor-fiber $P_{x,:,z}$.

This novel scheme enables splitting the broadcast of $A_{x,z}$ from a single processor $P_{x,z}$ in 2D algorithm among p_y processors. That is, the broadcast of A-matrix subblocks $A_{x,1,z}, A_{x,2,z}, \ldots, A_{x,y,z}$ are concurrently performed by processors $P_{x,1,z}, P_{x,2,z}, \ldots, P_{x,y,z}$, respectively. In a similar manner, the broadcast of $B_{y,z}$ is split among p_x processors. Since C-matrix blocks are also split along the z dimension, final C-block results are obtained from local partial C-subblock results through reduce (fold) type of operation along the z dimension. Compared to the 2D algorithm, the 3D algorithm is more scalable with better upper bounds on communication overhead [19] at the expense of reduce operations along the z dimension.

In this algorithm, the workcube is partitioned into $p_x\times p_y\times p_z$ cuboids, where each cuboid $W_{x,y,z}$ is of size $m_x\times\ell_z\times n_y$. Then each processor $P_{x,y,z}$ is responsible of computing the tasks/voxels in the cuboid $W_{x,y,z}$. In other words, the C-face of the workcube is decomposed into $p_x\times p_y$ 2D block submatrices, each denoted by $C_{x,y}$, where $C_{x,y}$ is collectively computed among the processors of the processor-fiber $P_{x,y,:}$. Fig. 4 displays a sample workcube partitioning and the corresponding matrix partitioning among $3\times4\times4=48$ processors.

4 Partitioning Models

Here, we first present background material on hypergraph partitioning and then present proposed hypergraph models

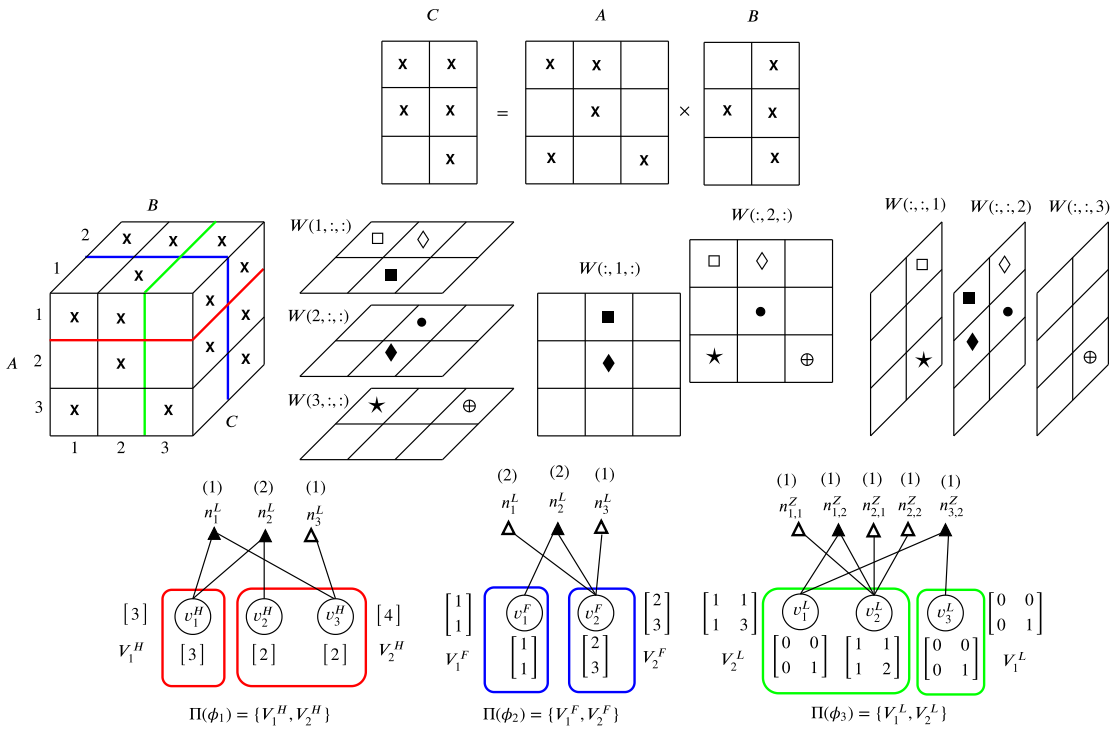

Fig. 3. Top: 3×3 A-matrix is multiplied by 3×2 B-matrix to produce 3×2 C-matrix. Middle:The corresponding $3\times2\times2$ workcube and its decomposition into horizontal, frontal and lateral layers where each of the 7 voxels is denoted by a different legend. Bottom: Hypergraphs built in phases ϕ_1, ϕ_2, ϕ_3. Partition $(\Pi(\phi_1),\Pi(\phi_2),\Pi(\phi_3))$ obtained on hypergraphs $H(\phi_1)$, $H(\phi_2)$, and $H(\phi_3)$ induces a $2\times2\times2$ partition on the workcube. Empty triangles denote internal nets whereas solid triangles denote cut nets.

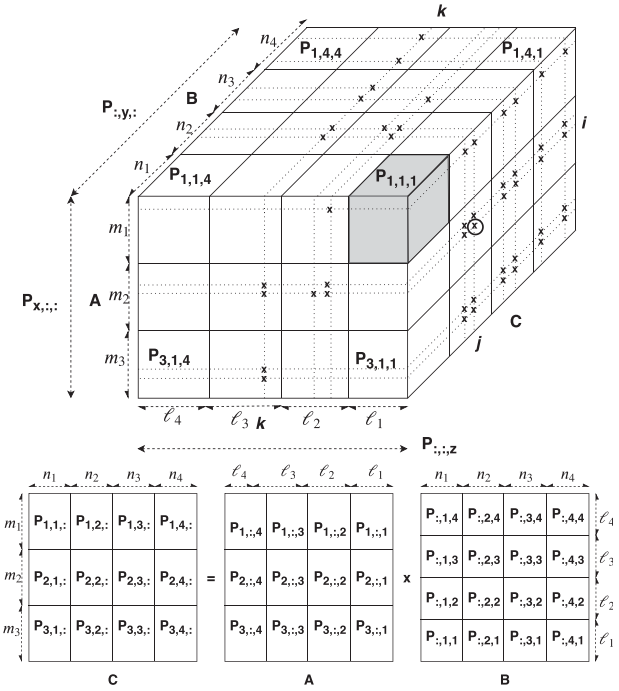

Fig. 4. Split-3D-SpGEMM algorithm (3D) algorithm: Top: Partitioning of the workcube on a $3\times4\times4$ 3D processor grid. Bottom: Data distribution. Solid lines show that rows and columns of all matrices are partitioned conformably with the workcube/task partition."x" denotes a nonzero entry in the respective matrix.

for 2D and 3D cartesien partitioning of the workcube. The former and the latter partitioning models are proposed for improving the performance of 2D- and 3D-parallel algorithms described in Sections 3.2 and 3.3, respectively.

4.1 Hypergraph Partitioning

A hypergraph $H=(V,N)$ is defined as a two-tuple of vertex set V and net set N. A hypergraph is the generalization of a graph, where each net connects possibly more than two vertices and the set of vertices connected by a net n_j is represented by $pins(n_j)$. Each vertex $v_i\in V$ is associated with C weights $w^c(v_i)$ for $c=1,\ldots,C$, and each net $n_j\in N$ is associated with $cost(n_j)$.

$\Pi=\{V_1,V_2\ldots V_K\}$ is a K-way partition of H if vertex parts are mutually disjoint and exhaustive. In Π, a net n_j connecting at least one vertex in a part is said to connect that part. The connectivity set $\Lambda(n_j)$ and the connectivity $\lambda(n_j)=|\Lambda(n_j)|$ of net n_j are respectively defined as the set of parts and the number of parts connected by n_j. A net n_j that connects more than one part (i.e., $\lambda(n_j)>1$) is said to be cut and uncut otherwise. The partitioning objective is to minimize the connectivity cut size defined as

$$\text{CutSize}(\Pi)=\sum_{n_j\in N} cost(n_j)\times(\lambda(n_j)-1). \tag{1}$$

The partitioning constraint is to maintain the balance criteria

$$W^c(V_k)=\sum_{v_i\in V_k} w^c(v_i)\le W^c_{avg}(1+\epsilon^c), \tag{2}$$

for all $V_k\in\Pi$ and $c=1\ldots C$. Here, $W^c(V_k)$ denotes the cth weight of part V_k, $W^c_{avg}=\sum_{v_i\in V} w^c(v_i)/K$ denotes the average part weight on the cth vertex weights and ϵ is the maximum allowed imbalance ratio on part weights.

4.2 2D Cartesian Partitioning of Workcube

The proposed model contains two partitioning phases ϕ_1 and ϕ_2. In phases ϕ_1 and ϕ_2, the horizontal and frontal layers of the workcube are partitioned into p_x and p_y parts each of which is assigned to a distinct row and column of the processor grid, respectively.

For phase ϕ_1, we define a hypergraph $H(\phi_1)=\{V^H,N^L\}$ with m vertices, ℓ nets and $nnz(A)$ pins, where $nnz(\cdot)$ refers to the number nonzeros in the respective matrix. $H(\phi_1)$ contains a vertex $v_i^H\in V^H$ for each horizontal layer $W(i,:,:)$. So, vertex v_i^H represents the task of computing the ith row of C. $H(\phi_1)$ contains a net $n_k^L\in N^L$ for the kth lateral layer of W. Net n_k^L, which represents the kth row of matrix B, connects each vertex v_i^H for which horizontal layer $W(i,:,:)$ has at least one voxel in fiber $W(i,:,k)$. Formally

$$pins(n_k^L)=\{v_i^H\mid\exists W(i,j,k)\in W(i,:,:)\cap W(:,:,k)\}.$$

Alternatively, in matrix theoretic view

$$pins(n_k^L)=\{v_i^H\mid\exists k\in cols(A(i,:))\},$$

where $cols(A(i,:))$ denote the column indices of the nonzeros in the ith row of A.

Each vertex v_i^H is assigned a weight $w(v_i^H)$ equal to the number of voxels in the ith horizontal layer $W(i,:,:)$, i.e.,

$$w(v_i^H)=|W(i,:,:)|\quad=\sum_{k\in cols(A(i,:))} nnz(B(k,:)).$$

Here, $|\cdot|$ denotes the number of voxels in the respective subcube of W. Each net is assigned a cost equal to the number of nonzeros in the respective B-matrix row, i.e.,

$$cost(n_k^L)=nnz(B(k,:)).$$

A p_x-way partition $\Pi_{p_x}(\phi_1)=\{V_1^H,V_2^H,\ldots,V_{p_x}^H\}$ of $H(\phi_1)$ is decoded as follows for task partitioning: Without loss of generality, all tasks associated with vertices in $V_x^H\in\Pi_{p_x}(\phi_1)$ are assigned to the processors of processor-row $P_{x,:}$. That is, the task of computing an individual row of matrix C is confined among the processors of the same row of the grid. This partitioning can also be considered as partitioning the horizontal layers of W and then utilizing this partition as inducing a partial reordering on the horizontal layers so that the horizontal layers belonging to the same part are reordered consecutively (in any order) to form a horizontal block. Here and hereafter, this partially reordered workcube will be referred to as $\overline{W}$. Then, the xth horizontal block $\overline{W}_{x,:,:}$ of $\overline{W}$ is assigned to processor-row $P_{x,:}$.

The weight of part V_x^H computed according to (2) is equal to the number of voxels in the horizontal block $\overline{W}_{x,:,:}$. So, the partitioning constraint of maintaining balance on part weights encodes balance on the voxel counts of the p_x horizontal blocks thus encoding computational balance among p_x processor-rows.

Consider a cut-net n_k^L with $\Lambda(n_k^L)$. Each part $V_x^H\in\Lambda(n_k^L)$ corresponds to the processor-row $P_{x,:}$ which is assigned the horizontal layers whose intersection with the kth lateral

layer has at least one voxel. In other words, for each part $V_x^H \in \Lambda(n_k^L)$, the tasks assigned to the processors of the processor-row $P_{x,:}$ in $\Lambda(n_k^L)$ require the B-matrix row $B(k,:)$.

In Fig. 3, the first two subfigures of the bottom part show $H(\phi_1)$ and $H(\phi_2)$ respectively for the sample workcube given in the middle part of the figure. $H(\phi_1)$ contains three vertices and three nets corresponding to the three horizontal and three lateral layers, respectively. $H(\phi_2)$ contains two vertices and three nets corresponding to the two frontal and three lateral layers, respectively. In these two subfigures, $[\cdot]$ below each vertex shows the weight(s) associated with that vertex. $[\cdot]$ besides each part denotes the weight of that part. $(\cdot)$ above each net shows the cost of that net. For example, in $H(\phi_1)$, $w(v_1^H)=3$ since $|W(1,:,:)|=3$ and $cost(n_1^L)=1$ since $nnz(B(1,:))=1$. In $H(\phi_2)$, $w^1(v_2^F)=2$ and $w^2(v_2^F)=3$ since $|W(:,2,:) \cap W_{1,:,:}|=2$ and $|W(:,2,:) \cap W_{2,:,:}|=3$, respectively. In $H(\phi_2)$, $cost(n_1^L)=2$ since $nnz(A(:,1))=2$. In the sample 2-way partition of $H(\phi_1)$, n_3^L is internal to part V_2^H, whereas nets n_1^L and n_2^L are cut thus incurring a communication volume of 3 words during the expand phase of B-matrix entries. In the sample 2-way partition of $H(\phi_2)$, n_1^L and n_3^L are internal to part V_2^F, whereas only net n_2^L is cut thus incurring a communication volume of 2 words during the expand phase of A-matrix entries.

Here, we define a B-matrix row distribution to be consistent with task partition $\Pi(\phi_1)$ if each B-matrix row k is assigned to one of the processor-rows in $\Lambda(n_k^L)$. The distribution of nonzeros of a B-matrix row among the processors of the respective processor-row will be determined by the partition to be obtained in phase ϕ_2. Under this consistent data distribution, assume that processor-row $P_{x,:}$ in $\Lambda(n_k^L)$ stores B-matrix row k. So, processor-row $P_{x,:}$ will expand B-matrix row k to all processor-rows in $\Lambda(n_k^L) - \{P_{x,:}\}$ so that cut-net n_k^L will incur the communication of

$$nnz(B(k,:)) \times \big(|\Lambda(n_k^L)| - 1\big),$$

words. So, the total communication volume associated with B-matrix rows between processor-rows is

$$\text{ExpVol}(B) = \sum_{n_k^L \in N^L} nnz(B(k,:)) \times \big(|\Lambda(n_k^L)| - 1\big).$$

Therefore, the partitioning objective of minimizing the cut-size according to (1) encodes the minimization of the total communication volume during the expand type communications on B-matrix rows.

For phase ϕ_2, we define a hypergraph $H(\phi_2)=\{V^F, N^L\}$ with n vertices, ℓ nets and $nnz(B)$ pins. $H(\phi_2)$ contains a vertex $v_j^F \in V^F$ for each frontal layer $W(:,j,:)$. So vertex v_j^F represents the task of computing the jth column of C. $H(\phi_2)$ contains a net $n_k^L \in N^L$ for the kth lateral layer of W. Net n_k^L, which represents the kth column of matrix A, connects each vertex v_j^F for which frontal layer $W(:,j,:)$ has at least one voxel in fiber $W(:,j,k)$. Formally

$$pins(n_k^L) = \{v_j^F \mid \exists W(i,j,k) \in W(:,j,:) \cap W(:,:,k)\}.$$

Alternatively, in matrix view

$$pins(n_k^L) = \{v_j^F \mid \exists k \in rows(B(:,j))\},$$

where $rows(B(:,j))$ denote the row indices of the nonzeros in the jth column of B. Each net n_k^L is associated with a cost equal to the number of nonzeros in the respective A-matrix column, i.e.,

$$cost(n_k^L) = nnz(A(:,k)).$$

A p_y-way partition $\Pi_{p_y}(\phi_2)=\{V_1^F, V_2^F, \ldots, V_{p_y}^F\}$ of hypergraph $H(\phi_2)$ is decoded as follows for task partitioning: All tasks associated with vertices in $V_y^F \in \Pi_{p_y}(\phi_2)$ are assigned to the processors of processor-column $P_{:,y}$. That is, the task of computing an individual column of matrix C is confined among the processors of the same column of the grid. This partitioning can also be considered as partitioning the frontal layers of W and then utilizing this partition as inducing a partial reordering on the frontal layers in such a way that the frontal layers belonging to the same part are reordered consecutively (in any order) to form a frontal block. Then, the yth frontal block $\overline{W}_{:,y,:}$ of the reordered workcube is assigned to processor-column $P_{:,y}$.

The p_x-way horizontal partition Π_{p_x} obtained in phase ϕ_1 together with p_y-way partition Π_{p_y} obtained in phase ϕ_2 can be considered as forming fiber blocks such that fiber block $\overline{W}_{x,y,:}$ contains voxels in the intersection of the xth horizontal block $\overline{W}_{x,:,:}$ and the yth frontal block $\overline{W}_{:,y,:}$ of the reordered workcube. So, partition $(\Pi(\phi_1), \Pi(\phi_2))$ is decoded as assigning fiber block $\overline{W}_{x,y,:}$ to processor $P_{x,y}$.

In phase ϕ_2, a multi-constraint partitioning formulation is proposed in order to maintain balance on the voxel counts of the fiber blocks. For this purpose, each vertex v_j^F of V^F is assigned p_x weights $w^c(v_j^F)$ for $c=1,2,\ldots,p_x$. Here, $w^c(v_j^F)$ is set equal to the number of voxels of the frontal layer $W(:,j,:)$ in the horizontal block $\overline{W}_{c,:,:}$ induced by the vertex part V_c^H of $\Pi_{p_x}(\phi_1)$. That is

$$w^c(v_j^F) = |W(:,j,:) \cap \overline{W}_{c,:,:}|.$$

Alternatively, in matrix view

$$w^c(v_j^F) = \sum_{v_i^H \in V_c^H} |\{k \mid k \in cols(A(i,:)) \wedge k \in rows(B(:,j))|.$$

For a given partition $\Pi_{p_y}(\phi_2)=\{V_1^F, V_2^F, \ldots, V_{p_y}^F\}$ of $H(\phi_2)$, the sum of the cth weights of the vertices in part $V_y^F \in \Pi_{p_y}(\phi_2)$ is equal to the number of voxels in the fiber block $\overline{W}_{c,y,:}$ That is

$$\begin{aligned} W^c(V_y^F) &= \sum_{v_j^F \in V_y^F} w^c(v_j^F) = \sum_{v_j^F \in V_y^F} |W(:,j,:) \cap \overline{W}_{c,:,:}| \\ &= |\overline{W}_{:,y,:} \cap \overline{W}_{c,:,:}|. \end{aligned}$$

So, the cth partitioning constraint (2) of maintaining balance on the cth weights of the parts encodes maintaining balance on the voxel counts of the fiber blocks in the cth horizontal block. Recall that the horizontal partition $\Pi_{p_x}(\phi_1)$ of W obtained in the first phase ϕ_1 already produces horizontal blocks with roughly equal number of voxels. Hence, the single partitioning constraint in ϕ_1 together with the p_x partitioning constraints in ϕ_2 encodes maintaining balance on the voxel counts in the individual fiber blocks. Since each fiber block is assigned to a distinct processor, the proposed

multi-constraint partitioning formulation encodes computational load-balance among processors.

Consider a cut-net n_k^L with $\Lambda(n_k^L)$. Each part $V_y^F \in \Lambda(n_k^L)$ corresponds to the processor-column $P_{:,y}$ which is assigned the frontal layers whose intersection with the kth lateral layer has at least one voxel. That is, for each part $V_y^F \in \Lambda(n_k^L)$, the tasks assigned to the processors of the processor-column $P_{:,y}$ in $\Lambda(n_k^L)$ require the A-matrix column $A(:,k)$.

Here, we define an A-matrix column distribution to be consistent with task partition $\Pi(\phi_2)$ if each A-matrix column k is assigned to one of the processor-columns in $\Lambda(n_k^L)$. Under this consistent data distribution, assume that processor-column $P_{:,y}$ in $\Lambda(n_k^L)$ stores A-matrix column k. So, processor-column $P_{:,y}$ will expand A-matrix column k to all processor-columns in $\Lambda(n_k^L) - \{P_{:,y}\}$ so that cut-net n_k^L will incur the communication of

$$nnz(A(:,k)) \times \big(|\Lambda(n_k^L)| - 1\big),$$

words. So, the total communication volume associated with A-matrix columns between processor-columns is

$$\text{ExpVol}(A) = \sum_{n_k^L \in N^L} nnz(A(:,k)) \times \big(|\Lambda(n_k^L)| - 1\big).$$

Therefore, the partitioning objective of minimizing the cut size according to (1) encodes the minimization of the total communication volume during the expand type communications on A-matrix columns.

As discussed earlier, task partition $\Pi(\phi_1)$ induces a consistent distribution of B-matrix rows among processor-rows. The distribution of nonzeros of a B-matrix row among the processors of the respective processor-row is determined by $\Pi(\phi_2)$ as follows: Assume that row $B(k,:)$ is assigned to a processor-row $P_{x,:} \in \Lambda(n_k^L)$ by utilizing $\Pi(\phi_1)$. Each nonzero $B(k,j)$ of row $B(k,:)$ is assigned to a processor $P_{x,y}$ if $v_j^F \in V_y^F$ in $\Pi(\phi_2)$. That is, the nonzero distribution of row $B(k,:)$ among processors of $P_{x,:}$ follows the partitioning obtained on frontal layers. Expanding a B-matrix row from a processor-row is realized collectively by the processor(s) of that grid row via expanding disjoint B-matrix nonzero row segment(s) along the respective processor-columns of the grid. Although the nonzero distribution of a B-matrix row along a processor-row may change the number of messages, it does not change the total communication volume for expanding nonzeros of that B-matrix row.

As also discussed earlier, task partition $\Pi(\phi_2)$ determines a consistent distribution of A-matrix columns among processor-columns. The distribution of nonzeros of an A-matrix column among the processors of the respective processor-column is determined by $\Pi(\phi_1)$ as follows: Assume that column $A(:,k)$ is assigned to a processor-column $P_{:,y}$ in $\Lambda(n_k^L)$ by utilizing $\Pi(\phi_2)$. Each nonzero $A(i,k)$ of column $A(:,k)$ is assigned to a processor $P_{x,y}$ if $v_i^H \in V_x^H$ in $\Pi(\phi_1)$. That is, the nonzero distribution of column $A(:,k)$ in processor-column $P_{:,y}$ follows the partitioning obtained on horizontal layers. Expanding an A-matrix column from a processor-column is realized collectively by the processor(s) of that column via expanding disjoint A-matrix nonzero column segment(s) along the respective processor-rows of the grid. Although the nonzero distribution of an A-matrix column along a processor-column may change the number of messages, it does not change the total communication volume for expanding nonzeros of that A-matrix column.

The discussion given in the above two paragraphs imply the following: The total communication volume on B-matrix rows is determined by the partitioning of A-matrix rows, whereas it is independent from the partitioning of B-matrix columns. The total communication volume on A-matrix columns is determined by the partitioning of B-matrix columns, whereas it is independent from the partitioning of A-matrix rows.

Fig. 2 depicts the proposed partitioning model. In phase ϕ_1, the kth lateral layer and row $B(k,:)$ are represented by a cut-net n_k^L with $\Lambda(n_k^L) = \{P_{2,:}, P_{3,:}\}$ since horizontal-layer blocks $\overline{W}_{2,:,:}$ and $\overline{W}_{3,:,:}$ contain voxels in the intersection with the kth lateral layer (i.e., voxels induced by the outer-product of $A(:,k)$ with $B(k,:)$). Hence, processors $P_{2,2}$, $P_{2,3}$ and $P_{2,4}$ in processor-row $P_{2,:} \in \Lambda(n_k^L)$ store nonzero row segments of row $B(k,:)$ and they expand these *three* row segments (drawn in *three* parallelograms) along their processor-columns. For instance, processor $P_{2,2}$ expands its nonzero row segment to processor $P_{3,2}$ since fiber blocks $\overline{W}_{2,2,:}$ and $\overline{W}_{3,2,:}$ have voxels and require this row segment. In phase ϕ_2, the kth lateral layer and column $A(:,k)$ are represented by a net n_k^L with $\Lambda(n_k^L) = \{P_{:,2}, P_{:,3}, P_{:,4}\}$ since frontal-layer blocks $\overline{W}_{:,2,:}$, $\overline{W}_{:,3,:}$ and $\overline{W}_{:,4,:}$ contain voxels in the intersection with kth lateral layer. Hence, processors $P_{2,2}$ and $P_{3,2}$ in processor-column $P_{:,2} \in \Lambda(n_k^L)$ store nonzero column segments of column $A(:,k)$ and responsible for expanding these column segments along their processor-rows. For instance, processor $P_{3,2}$ expands its nonzero column segment to processors $P_{3,3}$ and $P_{3,4}$ since fiber blocks $\overline{W}_{3,2,:}$, $\overline{W}_{3,3,:}$ and $\overline{W}_{3,4,:}$ have voxels and require this column segment.

4.3 3D Cartesian Partitioning of Workcube

The proposed model contains three partitioning phases ϕ_1, ϕ_2 and ϕ_3. In phases ϕ_1, ϕ_2 and ϕ_3, the horizontal, frontal and lateral layers of the workcube are partitioned into p_x, p_y and p_z parts each of which is assigned to a distinct horizontal, frontal and lateral layer of the processor grid, respectively. The hypergraph models $H(\phi_1)$ and $H(\phi_2)$ for the first two phases are exactly the same with those for the 2D partitioning model proposed in Section 4.2.

For phase ϕ_3, we define a hypergraph $H(\phi_3) = \{V^L, N^Z\}$ with ℓ vertices, $nnz(C)$ nets and $|W|$ pins. $H(\phi_3)$ contains a vertex $v_k^L \in V^L$ for each lateral layer $W(:,:,k)$. So vertex v_k^L represents the task of computing the outer-product of column $A(:,k)$ with row $B(k,:)$. $H(\phi_3)$ contains a net $n_{i,j}^Z \in N^Z$ for each fiber $W(i,j,:)$ that has a voxel (partial product) contributing to nonzero entry $C(i,j)$. Net $n_{i,j}^Z$, which represents nonzero $C(i,j)$ of matrix C, connects each vertex v_k^L for which lateral layer $W(:,:,k)$ has a voxel in fiber $W(i,j,:)$ (i.e., W contains voxel $W(i,j,k)$). Formally

$$pins(n_{i,j}^Z) = \{v_k^L \mid \exists W(i,j,k) \in W(i,j,:) \cap W(:,:,k)\}.$$

Alternatively, in matrix view

$$pins(n_{i,j}^Z) = \{v_k^L \mid \exists k \in cols(A(i,:)) \wedge \exists k \in rows(B(:,j))\}.$$

Each net $n_{i,j}^Z$ is associated with $cost(n_{i,j}^Z) = 1$.

A p_z-way partition $\Pi_{p_z}(\phi_3)=\{V_1^L, V_2^L, \ldots, V_{p_z}^L\}$ of hypergraph $H(\phi_3)$ is decoded as follows for task partitioning: All tasks associated with vertices in $V_z^L \in \Pi_{p_z}(\phi_3)$ are assigned to the processors of the zth lateral layer $P_{:,:,z}$ of the processor grid. That is, the task of computing an individual outer-product of a column of matrix A with the respective row of matrix B is confined among the processors of the same lateral layer of the grid. This partitioning can also be considered as partitioning the lateral layers of W and then utilizing this partition as inducing a partial reordering on the lateral layers in such a way that the lateral layers belonging to the same part are reordered consecutively (in any order) to form a lateral block. Then, the zth lateral block $\overline{W}_{:,:,z}$ of the reordered workcube is assigned to the zth lateral layer of the processor grid $P_{:,:,z}$.

The $p_x \times p_y$ horizontal fiber-block partition/assignment induced by $(\Pi(\phi_1), \Pi(\phi_2))$ in the first two phases together with the p_z-way partition Π_{p_z} obtained in the third phase can be considered as forming a $p_x \times p_y \times p_z$ cuboid partition such that a cuboid $\overline{W}_{x,y,z}$ contains voxels in the intersection of the horizontal fiber block $\overline{W}_{x,y,:}$ and the zth lateral block $\overline{W}_{:,:,z}$ of the reordered workcube. So, the partition $(\Pi(\phi_1), \Pi(\phi_2), \Pi(\phi_3))$ is decoded as assigning cuboid $\overline{W}_{x,y,z}$ to processor $P_{x,y,z}$.

For maintaining balance on the voxel counts of the cuboids, each vertex v_k^L of V^L is assigned $p_x \times p_y$ weights $w^{c,d}(v_k^L)$ for $c=1,2,\ldots,p_x$ and $d=1,2,\ldots,p_y$. For the sake clarity of presentation, constraints are presented in 2D-array format (c,d), whereas they are stored as 1D vectors to be conformable with the input format of the multi-constraint partitioners. Here, $w^{c,d}(v_k^L)$ is set equal to the number of voxels of the lateral layer $W(:,:,k)$ in the fiber block $\overline{W}_{c,d,:}$ induced by the vertex part V_c^H of $\Pi_{p_x}(\phi_1)$ and the vertex part V_d^F of $\Pi_{p_y}(\phi_2)$. That is

$$w^{c,d}(v_k^L) = |W(:,:,k) \cap \overline{W}_{c,d,:}|.$$

Alternatively, in matrix view

$$w^{c,d}(v_k^L) = |\{C(i,j) \,|\, v_i^H \in V_c^H \wedge v_j^F \in V_d^F \wedge\ k \in cols(A(i,:)) \wedge k \in rows(B(:,j))\}|.$$

For a given partition $\Pi_{p_z}(\phi_3)=\{V_1^L, V_2^L, \ldots, V_{p_z}^L\}$ of $H(\phi_3)$, the sum of the weights $w^{c,d}(v_k^L)$ of the vertices in part $V_z^L \in \Pi_{p_z}(\phi_3)$ is equal to the number of voxels in the cuboid $\overline{W}_{c,d,z}$ That is

$$W^{c,d}(V_z^F) = \sum_{v_k^L \in V_z^L} w^{c,d}(v_k^L) = \sum_{v_k^L \in V_z^L} |W(:,:,k) \cap \overline{W}_{c,d,:}| = |\overline{W}_{:,:,z} \cap \overline{W}_{c,d,:}| = |\overline{W}_{c,d,z}|.$$

So, the (c,d)th partitioning constraint (2) of maintaining balance on the (c,d)th weights of the parts encodes maintaining balance on the voxel counts of the cuboids in the horizontal fiber block $\overline{W}_{c,d,:}$ Recall that $(\Pi(\phi_1), \Pi(\phi_2))$ obtained in the first two phases already produce fiber blocks with roughly equal number of voxels. Hence, the single partitioning constraint in the first phase and p_x partitioning constraints in the second phase together with the $p_x \times p_y$ partitioning constraints in the third phase encode maintaining balance on the voxel counts in the individual cuboids. Since each cuboid is assigned to a distinct processor, the proposed multi-constraint partitioning formulation encodes computational load-balance among processors.

For a cut-net $n_{i,j}^Z$ with $\Lambda(n_{i,j}^Z)$, each part $V_z^L \in \Lambda(n_{i,j}^Z)$ corresponds to a lateral processor-layer $P_{:,:,z}$ which is assigned the lateral layers of W whose intersection with fiber $W(i,j,:)$ has at least one voxel . Hence, each part $V_z^L \in \Lambda(n_{i,j}^Z)$ denotes a processor-layer $P_{:,:,z}$ that produces partial results contributing to nonzero $C(i,j)$.

In Fig. 3, the rightmost subfigure of the bottom part shows $H(\phi_3)$. As seen in the figure, $H(\phi_3)$ contains three vertices and five nets corresponding to the three lateral layers and five C-matrix nonzeros, respectively. Each vertex is associated with four weights since both $H(\phi_1)$ and $H(\phi_2)$ are 2-way partitioned (i.e., $p_x \times p_y = 4$). In the sample 2-way partition of $H(\phi_3)$, nets $n_{1,1}^Z$, $n_{2,1}^Z$ and $n_{2,2}^Z$ are internal to part V_2^L, whereas nets $n_{1,2}^Z$ and $n_{3,2}^Z$ are cut thus incurring a communication volume of two words during the reduce operations on C-matrix nonzero entries.

We define a C-matrix nonzero distribution to be consistent with task partition $\Pi(\phi_3)$, if all partial results for $C(i,j)$ is accumulated and stored in one of the lateral processor-layers in $\Lambda(n_{i,j}^Z)$. Assume that $C(i,j)$ is assigned to one of the processor-layer $P_{:,:,z}$ in $\Lambda(n_{i,j}^Z)$. Also assume that $(\Pi(\phi_1), \Pi(\phi_2))$ induces the assignment of horizontal and frontal layers $W(i,:,:)$ and $W(:,j,:)$ to horizontal and frontal processor-layers $P_{x,:,:}$ and $P_{:,y,:}$, respectively. This induces the assignment of horizontal fiber $W(i,j,:)$ to processor-fiber $P_{x,y,:}$. Then, processor $P_{x,y,z}$ in processor-fiber $P_{x,y,:}$ will receive partial results contributing to $C(i,j)$ from processors in the intersection of the processor-fiber $P_{x,y,:}$ with each lateral processor-layer in $\Lambda(n_{i,j}^Z) - \{P_{:,:,z}\}$. Note that each processor having multiple partial results contributing to the same $C(i,j)$, which is assigned to another processor, will compute a single partial result through local summation and send this result to that processor. Hence, that cut-net $n_{i,j}^Z$ will incur the communication of $|\Lambda(n_{i,j}^Z)|-1$ words. So, the total communication volume associated with C-matrix nonzeros between lateral processor-layers is

$$\text{FoldVol}(C) = \sum_{n_{i,j}^Z \in N^Z} \Big(|\Lambda(n_{i,j}^Z)| - 1 \Big).$$

Therefore, the partitioning objective of minimizing the cut size according to (1) encodes the minimization of the total communication volume during the fold-type communications on C-matrix nonzeros.

Consistency of nonzero-based C-matrix distribution with $(\Pi(\phi 1), \Pi(\phi 2), \Pi(\phi 3))$ is already discussed earlier. The discussion for consistency of distribution of A-matrix columns and B-matrix rows with overall task partitioning $(\Pi(\phi 1), \Pi(\phi 2), \Pi(\phi 3))$ can be extended from the 2D discussion as follows: Column $A(:,k)$ and row $B(k,:)$ are stored by processors in lateral processor-layer $P_{:,:,z}$ if vertex v_k^L is assigned to $V_z^L \in \Pi(\phi_3)$. Assume that row $B(k,:)$ is assigned to a vertex part $V_x^H \in \Lambda(n_k^L)$ and correspondingly to horizontal processor-layer $P_{x,:,:}$ by utilizing $\Pi(\phi_1)$. So, each nonzero $B(k,j)$ of row $B(k,:)$ is assigned to processor $P_{x,y,z}$ if $v_j^F \in V_y^F$ in $\Pi(\phi_2)$. That is, row $B(k,:)$ is stored by processors in processor-fiber $P_{x,:,z}$ and expanding B-matrix nonzero row segment(s) is performed along processor-fibers $P_{:,y',z}$. Similarly, assume that column $A(:,k)$ is assigned to a vertex part $V_y^F \in \Lambda(n_k^L)$ and

correspondingly frontal processor-layer $P_{:,y,:}$ by utilizing $\Pi(\phi_2)$. So, each nonzero $A(i,k)$ of column $A(:,k)$ is assigned to processor $P_{x,y,z}$ if $v_i^H \in V_x^H$ in $\Pi(\phi_1)$. That is, column $A(:,k)$ is stored by processor in processor-fiber $P_{:,y,z}$ and expanding A-matrix nonzero column segment(s) is performed along processor-fiber $P_{x',:,z}$ of the grid.

Fig. 4 depicts the proposed partitioning model. As seen in the figure, in phase ϕ_1, cut-net n_k^L has $\Lambda(n_k^L)=\{P_{2,:,:}, P_{3,:,:}\}$. Hence, processors in one of the processor-layers in $\Lambda(n_k^L)$ can store nonzero row segments of $B(k,:)$ and expand these row segments along their respective processor-fibers. For instance, since lateral layer $W(:,:,k)$ is assigned to processor-layer $P_{:,:,3}$, processors $P_{2,2,3}$, $P_{2,3,3}$ and $P_{2,4,3}$ in processor-layer $P_{2,:,:} \in \Lambda(n_k^L)$ may store nonzero row segments of row $B(k,:)$ and expand these row segments along processor-fibers $P_{:,2,3}$, $P_{:,3,3}$ and $P_{:,4,3}$, respectively. In phase ϕ_2, cut-net n_k^L has $\Lambda(n_k^L)=\{P_{:,2,:}, P_{:,3,:}, P_{:,4,:}\}$. Hence, processors in one of the processor-layers in $\Lambda(n_k^L)$ can store nonzero column segments of $A(:,k)$ and expand these column segments along their respective processor-fibers. For instance, processors $P_{2,2,3}$ and $P_{3,2,3}$ in processor-layer $P_{:,2,:} \in \Lambda(n_k^L)$ may store nonzero column segments of column $A(:,k)$ and expand these column segments along processor-fibers $P_{2,:,3}$ and $P_{3,:,3}$, respectively.

As seen in Fig. 4, in phase ϕ_3, fiber $W(i,j,:)$ and nonzero $C(i,j)$ are represented by the cut-net $n_{i,j}^Z$ with $\Lambda(n_k^L) = \{P_{:,:,2}, P_{:,:,3}\}$, since intersections of fiber $W(i,j,:)$ with lateral blocks $\overline{W}_{:,:,2}$ and $\overline{W}_{:,:,3}$ have voxels (partial results) contributing to $C(i,j)$. The responsibility of accumulating and storing nonzero $C(i,j)$ can be given to a processor in one of the processor-layers in $\Lambda(n_k^L)$. For instance, $P_{2,2,3}$ may be given the responsibility of storing the final $C(i,j)$ and hence, $P_{2,2,2}$ may locally sum its *two* partial results and send a single partial result to $P_{2,2,3}$.

Hypergraph $H(\phi_3)$ contains $nnz(C)$ nets and $|W|$ pins which significantly increase the preprocessing overhead of the partitioning model for some SpGEMM instances. To alleviate this problem, instead of introducing a net $n_{i,j}^Z$ for each nonzero entry $C(i,j)$, we use a single net n_i^Z to denote row $C(i,:)$ of matrix C. Then, we add v_k^L as a pin to net n_i^Z if $k \in cols(A(i,:))$. These modifications lead to a hypergraph with m nets and $nnz(A)$ pins. In this way, the accumulation of C-matrix entries is performed in row-basis instead of nonzero-basis in such a way that the accumulation of entries in the same row is performed by the processors in the same processor-layer without considering individual consistency conditions of nonzero entries. That is, a nonzero $C(i,j)$ can be assigned to a processor-layer $P_{:,:,z} \notin \Lambda(n_{i,j}^Z)$ but $P_{:,:,z} \in \Lambda(n_i^Z)$. This approach drastically reduces the number of nets and the size of the hypergraph; but causes the hypergraph model to overestimate the total communication volume in the fold phase. This is because, even though a processor $P_{x,y,z}$ does not have a partial result contributing to a nonzero entry $C(i,j)$, the computation of final $C(i,j)$ can be assigned to this processor due to the assignment of row $C(i,:)$ to processor-layer $P_{:,:,z}$. We used this modified version of 3D scheme in our experiments.

5 Experiments

5.1 Experimental Setup

We use the following abbreviations: 2D and 3D refer to the sparse SUMMA and split-3D SPGEMM algorithms described in Sections 3.2 and 3.3, respectively. The prefix "H" refers to using the hypergraph models proposed in Section 4, whereas "R" refers to using random partitioning. Random partitioning is generated by randomly permuting horizontal, frontal and lateral layers of the workcube and then performing uniform 2D- or 3D-cartesian partitioning on this permuted workcube. H1D refers to using the hypergraph model described in [21] for 1D row-by-row parallel SpGEMM algorithm. H1D is used as a baseline algorithm for H2D and H3D, since H1D is reported as the best performing 1D algorithm in [21].

All parallel SpGEMM algorithms are implemented in C++ and by using *OpenMPI version 3.0.1*. For a fair comparison of partitioning algorithms, local SpGEMM computations are implemented using row-by-row product formulation [12] for all parallel SpGEMM implementations. The sequential SpGEMM implementation, which is used to obtain the speedups of the parallel algorithms, also uses row-by-row product formulation. We used our own sequential implementation of SpGEMM rather than the sequential implementation of CombBLAS library [9], since we found that our 2D- and 3D-parallel SpGEMM algorithms run faster than the implementation provided in CombBLAS on the SpGEMM instances utilized in the paper. Random partitioning is utilized both for our parallel SpGEMM implementations and the one provided in CombBLAS library.

The hypergraph models are partitioned via PaToH [33], [34] which supports multi-constraint hypergraph partitioning. The allowed imbalance ratio is set to $\epsilon = 0.01$ in each phase of the proposed partitioning models. Since PaToH contains randomized algorithms, the averages of three partitioning runs, each randomly seeded, are reported.

Experiments are performed on UHEMS's Sariyer system [35] in which each node contains an *Intel(R) Xeon(R) CPU E5-2680 v4 @2.40 GHz* processor consisting of 28 cores, and 128 GB main memory. Each MPI job is submitted to the system by allocating the number of cores as required by each job, since the tested algorithms do not utilize shared memory parallelism.

The performance of partitioning algorithms are evaluated for parallel SpGEMM algorithms on $p = 25, 100, 225, 400, 625$ and 900 processors. These processor counts are selected so that 2D virtual processor grids will be perfect squares 5×5, 10×10, 15×15, 20×20, 25×25 and 30×30, respectively. The 3D virtual grid sizes are selected such that lateral layers of processor grids are perfect squares, where the size of the third (i.e., z) dimension is selected accordingly. So, 3D virtual girds of sizes of $5\times5\times4$, $5\times5\times9$, $10\times10\times4$ and $10\times10\times9$, are used for $p = 100, 225, 400$ and 900, whereas the numbers of processors in the remaining two 3D virtual grids $3\times3\times3 = 27$ and $9\times9\times8 = 648$ are slightly larger than the p values 25 and 625, respectively.

Table 1 displays the properties of SpGEMM instances used in the experiments. For 20 $C = AA$ instances, A matrices are collected from UFL [36]. For two $C = AB$ instances, the recursive matrix generator *R-MAT* [37] is used to generate two SSCA matrices (HPCS Scalable Synthetic Compact Applications graph analysis benchmark [38]) to be used as input matrices A and B. We generate matrices for parameters $scale = 20$ and $scale = 21$, where for each value of $scale$, a matrix of size $2^{scale}\times2^{scale}$ is produced. Additionally, we

TABLE 1
Properties of Input Matrices of SpGEMM Instances

Matrix	Number of Rows/Cols	Number of Nonzeros	Max degree of Row	Max degree of Col
$C = AA$				
crankseg_2	63,838	14,148,858	3,423	3,423
net4-1	88,343	2,441,727	4,791	4,791
Ge99H100	112,985	8,451,395	469	469
Ge87H76	112,985	7,892,195	469	469
Ga10As10H30	113,081	6,115,633	698	698
torso1	116,158	8,516,500	3,263	1,224
Ga19As19H42	133,123	8,884,839	697	697
bmwcra_1	148,770	10,644,002	351	351
para-10	155,924	5,416,358	6,931	6,931
mono_500Hz	169,410	5,036,288	719	719
ohne2	181,343	11,063,545	3,441	3,441
Si41Ge41H72	185,639	15,011,265	662	662
Si87H76	240,369	10,661,631	361	361
Ga41As41H72	268,096	18,488,476	702	702
coPapersCiteseer	434,102	32,073,440	1,188	1,188
coPapersDBLP	540,486	30,491,458	3,299	3,299
pre2	659,033	5,959,282	628	745
3Dspectralwave	680,943	33,650,589	117	117
Stanford_Berkeley	683,446	7,583,376	83,448	249
StocF-1465	1,465,137	21,005,389	189	189
$C = AB$				
rmat ($scale = 20$)	1,048,576	8,259,994	1,181	1,158
rmat ($scale = 21$)	2,097,152	16,570,170	1,576	1,555

provide parameters $a = 0.55$, $b = 0.1$, $c = 0.1$ and $d = 0.25$, which were the default settings in the tool.

5.2 Performance Results

Tables 2 and 3 compare the relative performance of parallel SpGEMM algorithms in terms of multiple communication cost metrics as well as actual speedup values attained on the subject parallel system. Communication cost metrics are categorized under communication volume and message counts metrics which respectively relate to bandwidth and latency overheads. For both metrics, we display average and maximum volume/number of messages sent by a processor. For a fixed p, average message volume/count values also refer to total message volume/count values. We preferred to display average values instead of total values in order to better see how much the maximum values deviate from the average values. In Tables 2 and 3, for each p, results are displayed as averages (geometric means) over 20 instances.

Table 2 compares H2D against R2D as well as H3D against R3D to show the merits of utilizing the proposed hypergraph partitioning models instead of random partitioning. For communication cost comparison, H2D and H3D respectively achieve 85–89 and 62–76 percent less average volume than R2D and R3D over all p values. Similarly, H2D and H3D respectively achieve 69–84 and 5–62 percent less maximum send volume, 19–42 and 16–31 percent smaller average message counts, 4–19 and 3–8 percent smaller maximum message counts. Relatively smaller improvements in maximum message volume/count values compared to those in average message volume/count values can be attributed to better message volume and count balancing naturally achieved by random partitioning. For speedup comparison, H2D and H3D respectively achieve 27–63 and 31–45 percent higher speedup values than R2D and R3D.

TABLE 2
Average Performance Comparisons: H2D Over R2D and H3D Over R3D

p		Volume Avg	Volume Max	Messages Avg	Messages Max	S		Volume Avg	Volume Max	Messages Avg	Messages Max	S
25	R2D	3128	3180	8	8	11	R3D	7231	7909	6	6	9
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.11	0.16	0.81	0.96	1.27	H3D	0.24	0.38	0.84	0.97	1.39
100	R2D	1654	1714	18	18	31	R3D	3060	3468	11	11	26
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.12	0.19	0.76	0.93	1.48	H3D	0.29	0.55	0.78	0.95	1.40
225	R2D	1072	1136	28	28	54	R3D	1572	1843	16	16	44
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.13	0.22	0.71	0.90	1.53	H3D	0.34	0.66	0.76	0.95	1.45
400	R2D	769	837	38	38	76	R3D	1154	1456	21	21	72
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.13	0.25	0.65	0.86	1.52	H3D	0.34	0.72	0.72	0.92	1.32
625	R2D	586	645	48	48	85	R3D	731	919	23	23	86
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.14	0.27	0.63	0.83	1.43	H3D	0.37	0.88	0.73	0.93	1.34
900	R2D	466	603	58	58	63	R3D	564	726	26	26	92
		1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	1.00	1.00
	H2D	0.15	0.31	0.58	0.81	1.63	H3D	0.38	0.95	0.69	0.92	1.31

For each p, the first row displays the actual values, whereas the second and third rows display normalized values for the respective algorithm. Actual volume values (in thousands) are given in terms of input matrix nonzeros and output-matrix partial results sent by processors.

Table 3 compares hypergraph partitioning models on 1D-, 2D- and 3D-parallel SpGEMM algorithms. In the table, the third column shows the average imbalance ratios computed as the ratio of the computational load (voxel count) of the maximally loaded processor to the average processor load ($|W|/p$). In terms of computational load balance, H1D and H2D display comparable performance, whereas H3D displays considerably worse performance. The relative performance of H3D against H1D/H2D degrades with increasing p. This is because, the number of constraints used in the third partitioning phase of H3D increases with increasing p, where larger number of constraints may adversely affect the load-balancing quality of PaToH [39]. Experimental results on sensitivity of partitioning quality on the number of constrains are given in Appendix A, available in the online supplemental material.

As seen in Table 3, in terms of both communication volume metrics, H3D performs worse than both H1D and H2D. Two factors that explain this experimental finding are: First, the fold phase necessitates many partial results to be communicated among processors. Second, the increased number of constraints may adversely affect the cut-size quality of PaToH [39]. On the other hand, this performance gap between H1D and H3D decreases with increasing p: H3D incurs 5.21x and 2.81x more volume than H1D on $p = 25$ and $p = 900$ processors, respectively.

In terms of average message volume, H1D performs better than H2D for small processor counts ($p = 25, 100$ and 225), whereas H2D performs better for larger processor counts ($p = 400, 625$ and 900). In terms of maximum message volume, H2D performs significantly better than H1D on all processor

TABLE 3
Average Performance Comparison of H1D, H2D and H3D Algorithms

		Message (Actual)					Message (Norm.)				
			Volume		Count			Volume		Count		
p		imb	Avg	Max	Avg	Max	S	Avg	Max	Avg	Max	S
25	H1D	1.01	330	601	11	18	14	1.00	1.00	1.00	1.00	1.00
	H2D	1.01	357	519	7	8	13	1.08	0.86	0.60	0.43	0.97
	H3D	1.05	1719	2987	5	6	12	5.21	4.97	0.46	0.33	0.90
100	H1D	1.01	183	463	24	49	45	1.00	1.00	1.00	1.00	1.00
	H2D	1.01	193	331	14	17	46	1.06	0.71	0.57	0.34	1.02
	H3D	1.12	889	1897	9	10	36	4.87	4.09	0.36	0.21	0.80
225	H1D	1.04	131	430	35	84	65	1.00	1.00	1.00	1.00	1.00
	H2D	1.03	134	249	20	25	83	1.03	0.58	0.56	0.30	1.29
	H3D	1.24	529	1212	12	15	64	4.03	2.82	0.34	0.18	0.98
400	H1D	1.06	102	417	45	123	70	1.00	1.00	1.00	1.00	1.00
	H2D	1.04	100	210	25	33	115	0.98	0.50	0.56	0.27	1.65
	H3D	1.37	388	1052	15	19	96	3.81	2.52	0.34	0.16	1.38
625	H1D	1.10	86	394	52	161	62	1.00	1.00	1.00	1.00	1.00
	H2D	1.10	81	171	30	40	122	0.94	0.43	0.59	0.25	1.96
	H3D	1.54	272	807	17	21	116	3.16	2.05	0.33	0.13	1.86
900	H1D	1.13	75	415	55	196	56	1.00	1.00	1.00	1.00	1.00
	H2D	1.10	68	190	34	47	103	0.91	0.46	0.61	0.24	1.85
	H3D	1.61	214	688	18	24	121	2.84	1.66	0.33	0.12	2.17

Actual volume values (in thousands) are given in terms of input matrix nonzeros and output-matrix partial results sent by processors.

counts and this performance gap widens with increasing p in general. It is interesting to observe that the performance improvement of H2D over H1D is much higher in maximum message volume than its improvement in average message volume. For example, for $p=900$, on average, H2D incurs 54 percent less maximum message volume than H1D, whereas H2D incurs only 9 percent less average message volume than H1D. This is because, dense rows/columns of input matrices incur large communication volume for processors storing these rows/columns. This issue is largely resolved by H2D, since the nonzeros of individual dense rows/columns are partitioned among multiple processors.

In terms of both message count metrics, H3D is the clear winner, whereas H2D is the second best. The relative performance of H2D over H1D as well as the performance of H3D over both H2D and H1D increase with increasing p. For example, for $p=900$, H2D performs 39 and 76 percent better than H1D in average and maximum message count metrics, respectively. For $p=900$, H3D performs approximately 2x better than H2D in both average and maximum message count metrics. These experimental findings are expected, since 2D and 3D algorithms naturally establish the upper bounds of $O(\sqrt[2]{p})$ and $O(\sqrt[3]{p})$ on the number of messages handled by a processor, respectively, whereas this upper bound is $O(p)$ in the 1D algorithm.

As seen in Table 3, H1D and H2D display comparable average speedup values on small processor counts ($p=25$ and 100), whereas H3D displays worse performance. On the larger processors counts ($p=225, 400$ and 625), H2D becomes the clear winner, so that with increasing p, the performance gap between H2D and H1D widens, whereas the performance gap between H2D and H3D closes. H3D becomes the clear winner for the largest processor count $p=900$. These experimental results on speedup values conform with the relative performance variation of H1D, H2D and H3D in different communication cost metrics as discussed earlier. Parallel SpGEMM algorithms are bandwidth bound on smaller number of processors, whereas they become latency bound on larger number of processor so that H3D becomes the clear winner on $p=900$ processor even it incurs much more volume than both H1D and H2D.

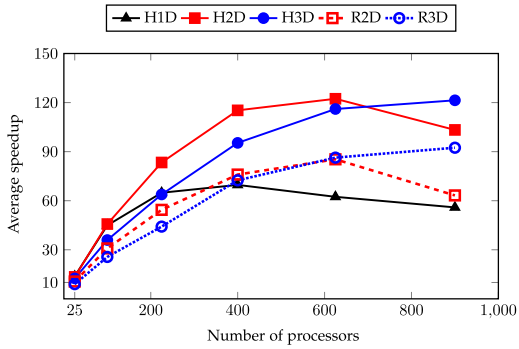

Fig. 5. Average speedup curves for 20 $C=AA$ instances.

Fig. 5 displays the average speedup curves (averaged over all 20 $C=AA$ instances) for all of the five algorithms, whereas speedup curves for each instance is given in Appendix C, available in the online supplemental material. As seen in Fig. 5, H2D achieves the best average speedup performance up to $p=625$ whereas it scales down on $p=900$. On the other hand, H3D continues to scale up to $p=900$ so that it achieves considerably higher average speedup than H2D on $p=900$. Observe that both R2D and R3D achieve higher speedup than H1D for $p\geq 400$.

Fig. 6 displays the speedup curves for two $C=AB$ instances. These instances constitute hard instances for intelligent partitioning algorithms because of skewed nonzero distributions along rows and columns. As seen in the figure, H1D scales only up to 100 processors, H2D scales up to 600 processors, whereas H3D scales up to 900 processors. Even on these hard instances, H2D and H3D achieve 6–15 and 5–10 percent higher speedup than R2D and R3D, respectively.

Fig. 7 displays the performance profiles [40] for parallel SpGEMM times for all *five* partitioning methods for a more comprehensive comparison. A test instance is defined as the parallel running time obtained by an algorithm for a matrix on a given number of processors. A point (x, y) for an algorithm in a profile denotes that the algorithm's performance is

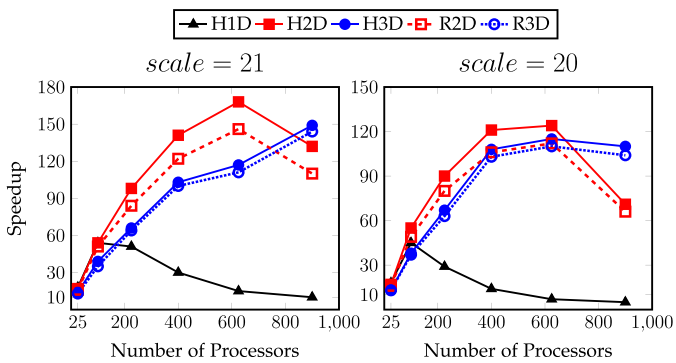

Fig. 6. Speedup curves for *R-MAT* $C=AB$ instances.

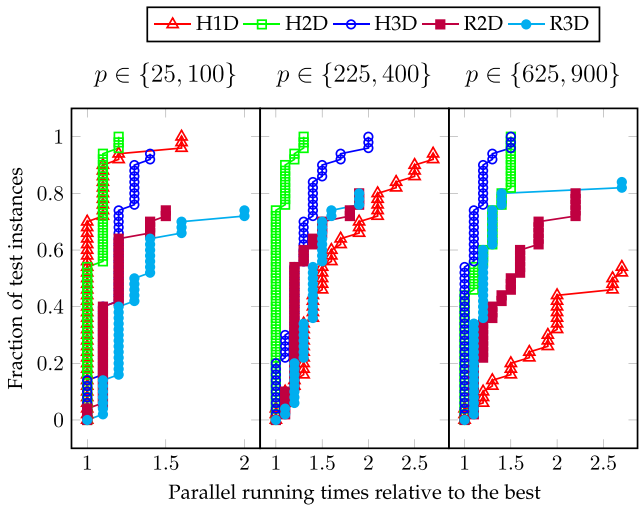

Fig. 7. Performance profiles for H1D, H2D, H3D, R2D, and R3D.

within x factor of the best result obtained in y fraction of the test instances. We compare the performances of algorithms under three different processor count groups: small ($p \in \{25, 100\}$), medium ($p \in \{225, 400\}$) and large ($p \in \{625, 900\}$). If the profile of an algorithm is closer to the y-axis, its performance is considered to be better.

Fig. 7 shows that for $p \in \{25, 100\}$, H1D achieves the best results on approximately 70 percent of the instances and its performance is within a factor of 1.6 of the best results, whereas H2D performs the best in 54 percent of the instances and its performance is within a factor of 1.2 of the best results in all instances. For $p \in \{225, 400\}$, H2D is the clear winner where it achieves the best in approximately 75 percent of the instances and its performance is within a factor of 1.2 of the best results in all instances. The second best performance is achieved by H3D and the performance of H1D significantly degrades in this processor group. For $p \in \{625, 900\}$, H3D and H2D respectively perform the best in 54 and 44 percent of the instances, whereas performance of both algorithms are within a factor of 1.5 of the best results in all instances.

As mentioned earlier, both 2D [18] and 3D [19] parallel SpGEMM algorithms perform communication operations in multiple stages through utilizing blocking factors to reduce processors' local memory requirements. In the proposed hypergraph models, the partitioning objective which corresponds to minimizing total communication volume also corresponds to minimizing the total sizes of the local communication buffers used for send and receive operations. In other words, the proposed hypergrah models already address minimizing the increase in the processors' local memory requirements due to communication buffers. For example, for H2D on $p = 400$, a single-stage communication scheme will incur an average increase of only 84 percent (between 25–200 percent) in processors' local memory requirements. This justifies the use of single-stage communications in our H2D and H3D implementations since multi-stage communication will increase the latency overhead significantly.

6 Conclusion

We proposed two novel hypergraph models that minimize total communication volume requirements of successful 2D- and 3D-parallel SpGEMM algorithms. Different from the previously proposed hypergraph models for 1D-parallel SpGEMM algorithms, our methods regard the multidimensional arrangement of processors and hence exploit the nice upper bounds of 2D and 3D algorithms on latency costs. In this way, our partitioning models provide much better optimizations in terms of both bandwidth and latency costs.

The proposed cartesian workcube partitioning models provide significant improvements on the scalability of 2D- and 3D-parallel algorithms. As the number of processors increase, the proposed partitioning models provide significant improvements over 1D counterparts, since the latency costs become more pronounced in the overall cost of communication at higher scales. Furthermore, improvements of the proposed models become significantly higher for SpGEMM instances that contain input matrices with dense rows/columns. Dense rows/columns incur high communication costs in terms of maximum communication volume handled by a processor in 1D algorithms, whereas communication load associated with such rows/columns are shared among multiple processors in 2D and 3D algorithms.

Acknowledgments

This work was supported in part by Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-115E512. Computing resources used were provided by the National Center for High Performance Computing of Turkey (UHeM) under Grant 4005992019.

References

[1] V. Hapla, D. Horák, and M. Merta, "Use of direct solvers in TFETI massively parallel implementation," in *Proc. Int. Workshop Appl. Parallel Comput.*, 2012, pp. 192–205.
[2] H. B. Schlegel *et al.*, "*Ab initio* molecular dynamics: Propagating the density matrix with Gaussian orbitals," *J. Chem. Phys.*, vol. 114, no. 22, pp. 9758–9763, 2001.
[3] A. D. Daniels, J. M. Millam, and G. E. Scuseria, "Semiempirical methods with conjugate gradient density matrix search to replace diagonalization for molecular systems containing thousands of atoms," *J. Chem. Phys.*, vol. 107, no. 2, pp. 425–431, 1997.
[4] R. H. Bisseling, T. Doup, and L. D. J. Loyens, "A parallel interior point algorithm for linear programming on a network of transputers," *Ann. Operations Res.*, vol. 43, no. 2, pp. 49–86, 1993.
[5] G. Karypis, A. Gupta, and V. Kumar, "A parallel formulation of interior point algorithms," in *Proc. ACM/IEEE Conf. Supercomput.*, 1994, pp. 204–213.
[6] M. Brezina and P. S. Vassilevski, "Smoothed aggregation spectral element agglomeration AMG: SA-ρAMGe," in *Proc. Int. Conf. Large-Scale Sci. Comput.*, 2011, pp. 3–15.
[7] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.*, 2010, pp. 421–434.
[8] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Comput. Sci. Eng.*, vol. 10, no. 2, pp. 20–25, 2008.
[9] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, 2011.
[10] S. Van Dongen, "Graph clustering by flow simulation," University of Utrecht, Utrecht Netherlands, Sep. 2000. [Online]. Available: https://dspace.library.uu.nl/bitstream/handle/1874/848/full.pdf?sequence=1
[11] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 804–811.
[12] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, 2011.

[13] K. Akbudak and C. Aykanat, "Exploiting locality in sparse matrix-matrix multiplication on many-core architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2258–2271, Aug. 2017.
[14] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C123–C152, 2012.
[15] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrixmatrix multiplication for the GPU," *ACM Trans. Math. Softw.*, vol. 41, no. 4, 2015, Art. no. 25.
[16] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM J. Sci. Comput.*, vol. 37, no. 1, pp. C54–C71, 2015.
[17] M. Deveci *et al.*, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures," *Parallel Comput.*, Elsevier, vol. 78, pp. 33–46, 2018.
[18] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C170–C191, 2012.
[19] A. Azad *et al.*, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM J. Sci. Comput.*, vol. 38, no. 6, pp. C624–C651, 2016.
[20] G. V. Demirci and C. Aykanat, "Scaling sparse matrix-matrix multiplication in the accumulo database," *Distrib. Parallel Databases*, vol. 38, pp. 31–62, 2020.
[21] K. Akbudak, O. Selvitopi, and C. Aykanat, "Partitioning models for scaling parallel sparse matrix-matrix multiplication," *ACM Trans. Parallel Comput.*, vol. 4, no. 3, 2018, Art. no. 13.
[22] K. Akbudak and C. Aykanat, "Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C568–C590, 2014.
[23] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Trans. Parallel Comput.*, vol. 3, no. 3, 2016, Art. no. 18.
[24] C. Ordonez, "Optimization of linear recursive queries in SQL," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 2, pp. 264–277, Feb. 2010.
[25] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, Jan./Feb. 2003.
[26] E. Boman, C. Phillips, and O. Parekh, "LDRD final report on massively-parallel linear programming: The parPCx system," Sandia National Laboratories, NM, USA, SAND2004–6440, Aug. 2005.
[27] Intel MKL, "Math kernel library (MKL)," 2007, Accessed: 2019. [Online]. Available: https://software.intel.com/en-us/mkl
[28] M. M. A. Patwary *et al.*, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *Proc. Int. Conf. High Perform. Comput.*, 2015, pp. 48–57.
[29] M. A. Heroux *et al.*, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.
[30] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Anal. Appl.*, vol. 32, no. 3, pp. 866–901, 2011.
[31] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2012, pp. 77–79.
[32] G. Ballard *et al.*, "Communication optimal parallel multiplication of sparse random matrices," in *Proc. 25th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2013, pp. 222–231.
[33] 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.
[34] U. V. Catalyürek and C. Aykanat, "PaToH: A multilevel hypergraph partitioning tool, version 3.0," Bilkent University, Dept. Comput. Eng., Ankara, 1999.
[35] UHEM Website, 2019. [Online]. Available: http://www.uhem.itu.edu.tr/
[36] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[37] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.
[38] SSCA Benchmark, 2019, [Online]. Available: http://www.graphanalysis.org/benchmark/
[39] C. Aykanat, B. B. Cambazoglu, and B. Uçar, "Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices," *J. Parallel Distrib. Comput.*, vol. 68, no. 5, pp. 609–625, 2008.
[40] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Math. Program.*, vol. 91, no. 2, pp. 201–213, 2002.

Gunduz Vehbi Demirci received the BS degree from the Computer Science and Engineering Department, Hacettepe University, Istanbul, Turkey, in 2011, and the MS and PhD degrees from the Computer Engineering Department, Bilkent University, Ankara, Turkey, in 2013 and 2019, respectively. His research interests include data mining, parallel computing, and distributed graph computations.

Cevdet Aykanat received the BS and MS degrees from Middle East Technical University, Ankara, Turkey, both in electrical engineering, and the PhD degree in electrical and computer engineering from Ohio State University, Columbus, Ohio. He worked with 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. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He has served as an associate editor of the *IEEE Transactions of Parallel and Distributed Systems* between 2008 and 2012.