Parallel Computing 59 (2016) 71–96

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems

Seher Acer, Oguz Selvitopi, Cevdet Aykanat*

Computer Engineering Department, Bilkent University, 06800 Ankara, Turkey

ARTICLE INFO

Article history:
Received 30 March 2015
Revised 25 August 2016
Accepted 5 October 2016
Available online 6 October 2016

Keywords:
Irregular applications
Sparse matrices
Sparse matrix dense matrix multiplication
Load balancing
Communication volume balancing
Matrix partitioning
Graph partitioning
Hypergraph partitioning
Recursive bipartitioning
Combinatorial scientific computing

ABSTRACT

We propose a comprehensive and generic framework to minimize multiple and different volume-based communication cost metrics for sparse matrix dense matrix multiplication (SpMM). SpMM is an important kernel that finds application in computational linear algebra and big data analytics. On distributed memory systems, this kernel is usually characterized with its high communication volume requirements. Our approach targets irregularly sparse matrices and is based on both graph and hypergraph partitioning models that rely on the widely adopted recursive bipartitioning paradigm. The proposed models are lightweight, portable (can be realized using any graph and hypergraph partitioning tool) and can simultaneously optimize different cost metrics besides total volume, such as maximum send/receive volume, maximum sum of send and receive volumes, etc., in a single partitioning phase. They allow one to define and optimize as many custom volume-based metrics as desired through a flexible formulation. The experiments on a wide range of about thousand matrices show that the proposed models drastically reduce the maximum communication volume compared to the standard partitioning models that only address the minimization of total volume. The improvements obtained on volume-based partition quality metrics using our models are validated with parallel SpMM as well as parallel multi-source BFS experiments on two large-scale systems. For parallel SpMM, compared to the standard partitioning models, our graph and hypergraph partitioning models respectively achieve reductions of 14% and 22% in runtime, on average. Compared to the state-of-the-art partitioner UMPa, our graph model is overall 14.5× faster and achieves an average improvement of 19% in the partition quality on instances that are bounded by maximum volume. For parallel BFS, we show on graphs with more than a billion edges that the scalability can significantly be improved with our models compared to a recently proposed two-dimensional partitioning model.

1. Introduction

Sparse matrix kernels form the computational basis of many scientific and engineering applications. An important kernel is the sparse matrix dense matrix multiplication (SpMM) of the form $Y = AX$, where A is a sparse matrix, and X and Y are dense matrices.

* Corresponding author. Fax: +90 312 266 4047.
E-mail addresses: acer@cs.bilkent.edu.tr (S. Acer), reha@cs.bilkent.edu.tr (O. Selvitopi), aykanat@cs.bilkent.edu.tr (C. Aykanat).

http://dx.doi.org/10.1016/j.parco.2016.10.001
0167-8191/

SpMM is already a common operation in computational linear algebra, usually utilized repeatedly within the context of block iterative methods. The practical benefits of block methods have been emphasized in several studies. These studies either focus on the block versions of certain solvers (i.e., conjugate gradient variants) which address multiple linear systems [1–4], or the block methods for eigenvalue problems, such as block Lanczos [5] and block Arnoldi [6]. The column dimension of X and Y in block methods is usually very small compared to that of A [7].

Along with other sparse matrix kernels, SpMM is also used in the emerging field of big data analytics. Graph algorithms are ubiquitous in big data analytics. Many graph analysis approaches such as centrality measures [8] rely on shortest path computations and use breadth-first search (BFS) as a building block. As indicated in several recent studies [9–14], processing each level in BFS is actually equivalent to a sparse matrix vector "multiplication". Graph algorithms often necessitate BFS from multiple sources. In this case, processing each level becomes equivalent to multiplication of a sparse matrix with another sparse (the SpGEMM kernel [15]) or dense matrix. For a typical small world network [16], matrix X is sparse at the beginning of BFS, however it usually gets denser as BFS proceeds. Even in cases when it remains sparse, the changing pattern of this matrix throughout the BFS levels and the related sparse bookkeeping overhead make it plausible to store it as a dense matrix if there is memory available.

SpMM is provided in Intel MKL [17] and Nvidia cuSPARSE [18] libraries for multi-/many-core and GPU architectures. To optimize SpMM on distributed memory architectures for sparse matrices with irregular sparsity patterns, one needs to take communication bottlenecks into account. Communication bottlenecks are usually summarized by latency (message start-up) and bandwidth (message transfer) costs. The latency cost is proportional to the number of messages while the bandwidth cost is proportional to the number of words communicated, i.e., communication volume. These costs are usually addressed in the literature with intelligent graph and hypergraph partitioning models that can exploit irregular patterns quite well [19–24]. Most of these models focus on improving the performance of parallel sparse matrix vector multiplication. Although one can utilize them for SpMM as well, SpMM necessitates the use of new models tailored to this kernel since it is specifically characterized with its high communication volume requirements because of the increased column dimensions of dense X and Y matrices. In this regard, the bandwidth cost becomes critical for overall performance, while the latency cost becomes negligible with increased average message size. Therefore, to get the best performance out of SpMM, it is vital to address communication cost metrics that are centered around volume such as maximum send volume, maximum receive volume, etc.

1.1. Related work on multiple communication cost metrics

Total communication volume is the most widely optimized communication cost metric for improving the performance of sparse matrix operations on distributed memory systems [21,22,25–27]. There are a few works that consider communication cost metrics other than total volume [28–33]. In an early work, Uçar and Aykanat [29] proposed hypergraph partitioning models to optimize two different cost metrics simultaneously. This work is a two-phase approach, where the partitioning in the first phase is followed by a latter phase in which they minimize total number of messages and achieve a balance on communication volumes of processors. In a related work, Uçar and Aykanat [28] adapted the mentioned model for two-dimensional fine-grain partitioning. A very recent work by Selvitopi and Aykanat aims to reduce the latency overhead in two-dimensional jagged and checkerboard partitioning [34].

Bisseling and Meesen [30] proposed a greedy heuristic for balancing communication loads of processors. This method is also a two-phase approach, in which the partitioning in the first phase is followed by a redistribution of communication tasks in the second phase. While doing so, they try to minimize the maximum send and receive volumes of processors while respecting the total volume obtained in the first phase.

The two-phase approaches have the flexibility of working with already existing partitions. However, since the first phase is oblivious to the cost metrics addressed in the second phase, they can get stuck in local optima. To remedy this issue, Deveci et al. [32] recently proposed a hypergraph partitioner called UMPa, which is capable of handling multiple cost metrics in a single partitioning phase. They consider various metrics such as maximum send volume, total number of messages, maximum number of messages, etc., and propose a different gain computation algorithm specific to each of these metrics. In the center of their approach are the move-based iterative improvement heuristics which make use of directed hypergraphs. These heuristics consist of a number of refinement passes. To each pass, their approach is reported to introduce an $O(VK^2)$-time overhead, where V is the number of vertices in the hypergraph (number of rows/columns in A) and K is the number of parts/processors. They also report that the slowdown of UMPa increases with increasing K with respect to the native hypergraph partitioner PaToH due to this quadratic complexity.

1.2. Contributions

In this study, we propose a comprehensive and generic one-phase framework to minimize multiple volume-based communication cost metrics for improving the performance of SpMM on distributed memory systems. Our framework relies on the widely adopted recursive bipartitioning paradigm utilized in the context of graph and hypergraph partitioning. Total volume can already be effectively minimized with existing partitioners [21,22,25]. We focus on the other important volume-based metrics besides total volume, such as maximum send/receive volume, maximum sum of send and receive volumes, etc. The proposed model associates additional weights with boundary vertices to keep track of volume loads of processors

during recursive bipartitioning. The minimization objectives associated with these loads are treated as constraints in order to make use of a readily available partitioner. Achieving a balance on these weights of boundary vertices through these constraints enables the minimization of target volume-based metrics. We also extend our model by proposing two practical enhancements to handle these constraints in partitioners more efficiently.

Our framework is unique and flexible in the sense that it handles multiple volume-based metrics through the same formulation in a generic manner. This framework also allows the optimization of any *custom* metric defined on send/receive volumes. Our algorithms are computationally lightweight: they only introduce an extra $O(nnz(A))$ time to each recursive bipartitioning level, where $nnz(A)$ is the number of nonzeros in matrix A. To the best of our knowledge, it is the first portable one-phase method that can easily be integrated into any state-of-the-art graph and hypergraph partitioner. Our work is also the first work that addresses multiple volume-based metrics in the graph partitioning context.

Another important aspect is the *simultaneous* handling of multiple cost metrics. This feature is crucial as overall communication cost is simultaneously determined by multiple factors and the target parallel application may demand optimization of different cost metrics simultaneously for good performance (SpMM and multi-source BFS in our case). In this regard, Uçar and Aykanat [28,29] accommodate this feature for two metrics, whereas Deveci et al. [32], although address multiple metrics, do not handle them in a completely simultaneous manner since some of the metrics may not be minimized in certain cases. Our models in contrast can optimize all target metrics simultaneously by assigning equal importance to each of them in the feasible search space. In addition, the proposed framework allows one to define and optimize as many volume-based metrics as desired.

For experiments, the proposed partitioning models for graphs and hypergraphs are realized using the widely-adopted partitioners Metis [22] and PaToH [21], respectively. We have tested the proposed models for 128, 256, 512 and 1024 processors on a dataset of 964 matrices containing instances from different domains. We achieve average improvements of up to 61% and 78% in maximum communication volume for graph and hypergraph models, respectively, in the categories of matrices for which maximum volume is most critical. Compared to the state-of-the-art partitioner UMPa, our graph model achieves an overall improvement of 5% in the partition quality 14.5× faster and our hypergraph model achieves an overall improvement of 11% in the partition quality 3.4× faster. Our average improvements for the instances that are bounded by maximum volume are even higher: 19% for the proposed graph model and 24% for the proposed hypergraph model.

We test the validity of the proposed models for both parallel SpMM and multi-source BFS kernels on large-scale HPC systems Cray XC40 and Lenovo NeXtScale, respectively. For parallel SpMM, compared to the standard partitioning models, our graph and hypergraph partitioning models respectively lead to reductions of 14% and 22% in runtime, on average. For parallel BFS, we show on graphs with more than a billion edges that the scalability can significantly be improved with our models compared to a recently proposed two-dimensional partitioning model [12] for the parallelization of this kernel on distributed systems.

The rest of the paper is organized as follows. Section 2 gives background for partitioning sparse matrices via graph and hypergraph models. Section 3 defines the problems regarding minimization of volume-based cost metrics. The proposed graph and hypergraph partitioning models to address these problems are described in Section 4. Section 5 proposes two practical extensions to these models. Section 6 gives experimental results for investigated partitioning schemes and parallel runtimes. Section 7 concludes.

2. Background

2.1. One-dimensional sparse matrix partitioning

Consider the parallelization of sparse matrix dense matrix multiplication (SpMM) of the form $Y = AX$, where A is an $n \times n$ sparse matrix, and X and Y are $n \times s$ dense matrices. Assume that A is permuted into a K-way block structure of the form

$$A_{BL} = \begin{bmatrix} C_1 & \cdots & C_K \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_K \end{bmatrix} = \begin{bmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & \ddots & \vdots \\ A_{K1} & \cdots & A_{KK} \end{bmatrix}, \tag{1}$$

for rowwise or columnwise partitioning, where K is the number of processors in the parallel system. Processor P_k owns row stripe $R_k = [A_{k1} \cdots A_{kK}]$ for rowwise partitioning, whereas it owns column stripe $C_k = [A_{1k}^T \cdots A_{Kk}^T]^T$ for columnwise partitioning. We focus on rowwise partitioning in this work, however, all described models apply to columnwise partitioning as well. We use R_k and A_k interchangeably throughout the paper as we only consider rowwise partitioning.

In both block iterative methods and BFS-like computations, SpMM is performed repeatedly with the same input matrix A and changing X-matrix elements. The input matrix X of the next iteration is obtained from the output matrix Y of the current iteration via element-wise linear matrix operations. We focus on the case where the rowwise partitions of the input and output dense matrices are conformable to avoid redundant communication during these linear operations. Hence, a partition of A naturally induces partition $[Y_1^T \ldots Y_K^T]^T$ on the rows of Y, which is in turn used to induce a conformable partition $[X_1^T \ldots X_K^T]^T$ on the rows of X. In this regard, the row and column permutation mentioned in (1) should be conformable.

A nonzero column segment is defined as the nonzeros of a column in a specific submatrix block. For example in Fig. 1, there are two nonzero column segments in A_{14} which belong to columns 13 and 15. In row-parallel $Y = AX$, P_k owns row

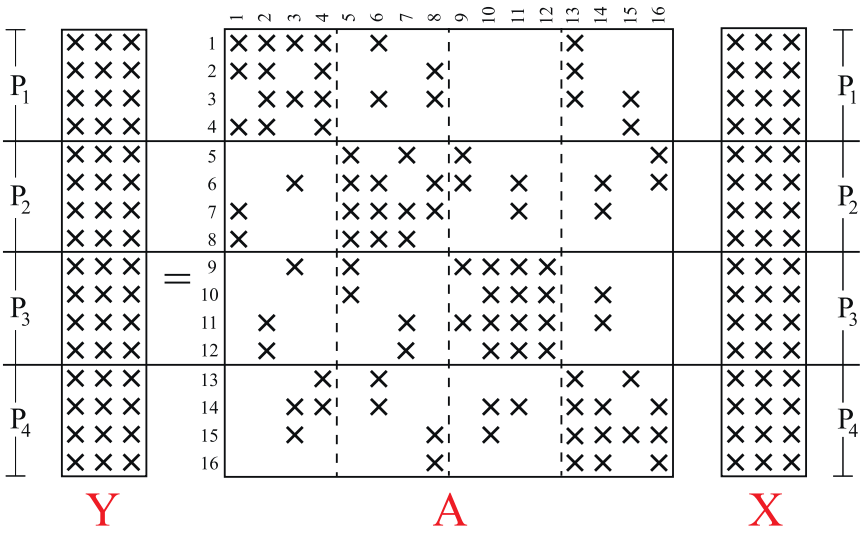

Fig. 1. Row-parallel $Y = AX$ with $K = 4$ processors, $n = 16$ and $s = 3$.

stripes A_k and X_k of the input matrices, and is responsible for computing respective row stripe $Y_k = A_k X$ of the output matrix. P_k can perform computations regarding diagonal block A_{kk} locally using its own portion X_k without requiring any communication, where A_{kl} is called a diagonal block if $k = l$, and an off-diagonal block otherwise. Since P_k owns only X_k, it needs the remaining X-matrix rows that correspond to nonzero column segments in off-diagonal blocks of A_k. Hence, the respective rows must be sent to P_k by their owners in a pre-communication phase prior to SpMM computations. Specifically, to perform the multiplication regarding off-diagonal block A_{kl}, P_k needs to receive the respective X-matrix rows from P_l. For example, in Fig. 1 for P_3, since there exists a nonzero column segment in A_{34}, P_3 needs to receive the corresponding three elements in row 14 of X from P_4. In a similar manner, it needs to receive the elements of X-matrix rows 2, 3 from P_1 and 5, 7 from P_2.

2.2. Graph and hypergraph partitioning problems

A graph $G = (\mathcal{V}, \mathcal{E})$ consists of a set $\mathcal{V}$ of vertices and a set $\mathcal{E}$ of edges. Each edge e_{ij} connects a pair of distinct vertices v_i and v_j. A cost c_{ij} is associated with each edge e_{ij}. $Adj(v_i)$ denotes the neighbors of v_i, i.e., $Adj(v_i) = \{v_j : e_{ij} \in \mathcal{E}\}$.

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set $\mathcal{V}$ of vertices and a set $\mathcal{N}$ of nets. Each net n_j connects a subset of vertices denoted as $Pins(n_j)$. A cost c_j is associated with each net n_j. $Nets(v_i)$ denotes the set of nets that connect v_i. In both graph and hypergraph, multiple weights $w^1(v_i), \ldots, w^C(v_i)$ are associated with each vertex v_i, where $w^c(v_i)$ denotes the cth weight associated with v_i.

$\Pi(G) = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ and $\Pi(\mathcal{H}) = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ are called K-way partitions of G and $\mathcal{H}$ if parts are mutually disjoint and mutually exhaustive. In $\Pi(G)$, an edge e_{ij} is said to be cut if vertices v_i and v_j are in different parts, and uncut otherwise. The cutsize of $\Pi(G)$ is defined as $\sum_{e_{ij} \in \mathcal{E}_E} c_{ij}$, where $\mathcal{E}_E \subseteq \mathcal{E}$ denotes the set of cut edges. In $\Pi(\mathcal{H})$, the connectivity set $\Lambda(n_j)$ of net n_j consists of the parts that are connected by that net, i.e., $\Lambda(n_j) = \{\mathcal{V}_k : Pins(n_j) \cap \mathcal{V}_k \neq \emptyset\}$. The number of parts connected by n_j is denoted by $\lambda(n_j) = |\Lambda(n_j)|$. A net n_j is said to be cut if it connects more than one part, i.e., $\lambda(n_j) > 1$, and uncut otherwise. The cutsize of $\Pi(\mathcal{H})$ is defined as $\sum_{n_j \in \mathcal{N}} c_j(\lambda(n_j) - 1)$. A vertex v_i in $\Pi(G)$ or $\Pi(\mathcal{H})$ is said to be a boundary vertex if it is connected by at least one cut edge or cut net.

The weight $W^c(\mathcal{V}_k)$ of part $\mathcal{V}_k$ is defined as the sum of the cth weights of the vertices in $\mathcal{V}_k$. A partition $\Pi(G)$ or $\Pi(\mathcal{H})$ is said to be balanced if

$$W^c(\mathcal{V}_k) \leq W^c_{avg}(1 + \epsilon^c), \quad k \in \{1, \ldots, K\} \quad \text{and} \quad c \in \{1, \ldots, C\}, \tag{2}$$

where $W^c_{avg} = \sum_k W^c(\mathcal{V}_k)/K$, and ϵ^c is the predetermined imbalance value for the cth weight.

The K-way multi-constraint graph/hypergraph partitioning problem [35,36] is then defined as finding a K-way partition such that the cutsize is minimized while the balance constraint (2) is maintained. Note that for $C = 1$, this reduces to the well-studied standard partitioning problem. Both graph and hypergraph partitioning problems are NP-hard [37,38].

2.3. Sparse matrix partitioning models

In this section, we describe how to obtain a one-dimensional rowwise partitioning of matrix A for row-parallel $Y = AX$ using graph and hypergraph partitioning models. These models are the extensions of standard models used for sparse matrix vector multiplication [21,22,39–41].

In the graph and hypergraph partitioning models, matrix A is represented as an undirected graph $G = (\mathcal{V}, \mathcal{E})$ and a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$. In both, there exists a vertex $v_i \in \mathcal{V}$ for each row i of A, where v_i signifies the computational task of multiplying row i of A with X to obtain row i of Y. So, in both models, a single ($C = 1$) weight of s times the number of nonzeros in row i of A is associated with v_i to encode the load of this computational task. For example, in Fig. 1, $w^1(v_5) = 4 \times 3 = 12$.

In G, each nonzero a_{ij} or a_{ji} (or both) of A is represented by an edge $e_{ij} \in \mathcal{E}$. The cost of edge e_{ij} is assigned as $c_{ij} = 2s$ for each edge e_{ij} with $a_{ij} \neq 0$ and $a_{ji} \neq 0$, whereas it is assigned as $c_{ij} = s$ for each edge e_{ij} with either $a_{ij} \neq 0$ or $a_{ji} \neq 0$, but not both. In $\mathcal{H}$, each column j of A is represented by a net $n_j \in \mathcal{N}$, which connects the vertices that correspond to the rows that contain a nonzero in column j, i.e., $Pins(n_j) = \{v_i : a_{ij} \neq 0\}$. The cost of net n_j is assigned as $c_j = s$ for each net in $\mathcal{N}$.

In a K-way partition $\Pi(G)$ or $\Pi(\mathcal{H})$, without loss of generality, we assume that the rows corresponding to the vertices in part $\mathcal{V}_k$ are assigned to processor P_k. In $\Pi(G)$, each cut edge e_{ij}, where $v_i \in \mathcal{V}_k$ and $v_j \in \mathcal{V}_\ell$, necessitates c_{ij} units of communication between processors P_k and P_ℓ. Here, P_ℓ sends row j of X to P_k if $a_{ij} \neq 0$ and P_k sends row i of X to P_ℓ if $a_{ji} \neq 0$. In $\Pi(\mathcal{H})$, each cut net n_j necessitates $c_j(\lambda(n_j) - 1)$ units of communication between processors that correspond to the parts in $\Lambda(n_j)$, where the owner of row j of X sends it to the remaining processors in $\Lambda(n_j)$. Hereinafter, $\Lambda(n_j)$ is interchangeably used to refer to parts and processors because of the identical vertex part to processor assignment.

Through these formulations, the problem of obtaining a good row partitioning of A becomes equivalent to the graph and hypergraph partitioning problems in which the objective of minimizing cutsize relates to minimizing total communication volume, while the constraint of maintaining balance on part weights ((2) with $C = 1$) corresponds to balancing computational loads of processors. The objective of hypergraph partitioning problem is an exact measure of total volume, whereas the objective of graph partitioning problem is an approximation [21].

3. Problem definition

Assume that matrix A is distributed among K processors for parallel SpMM operation as described in Section 2.1. Let $\sigma(P_k, P_\ell)$ be the amount of data sent from processor P_k to P_ℓ in terms of X-matrix elements. This is equal to s times the number of X-matrix rows that are owned by P_k and needed by P_ℓ, which is also equal to s times the number of nonzero column segments in off-diagonal block $A_{\ell k}$. Since X_k is owned by P_k and computations on A_{kk} require no communication, $\sigma(P_k, P_k) = 0$. We use the function $ncs(.)$ to denote the number of nonzero column segments in a given block of matrix. $ncs(A_{k\ell})$ is defined to be the number of nonzero column segments in $A_{k\ell}$ if $k \neq \ell$, and 0 otherwise. This is extended to a row stripe R_k and a column stripe C_k, where $ncs(R_k) = \sum_\ell ncs(A_{k\ell})$ and $ncs(C_k) = \sum_\ell ncs(A_{\ell k})$. Finally, for the whole matrix, $ncs(A_{BL}) = \sum_k ncs(R_k) = \sum_k ncs(C_k)$. For example, in Fig. 1, $ncs(A_{42}) = 2$, $ncs(R_3) = 5$, $ncs(C_3) = 4$ and $ncs(A_{BL}) = 21$.

The send and receive volumes of P_k are defined as follows:

- $SV(P_k)$, *send volume of* P_k: The total number of X-matrix elements sent from P_k to other processors. That is, $SV(P_k) = \sum_\ell \sigma(P_k, P_\ell)$. This is equal to $s \times ncs(C_k)$.
- $RV(P_k)$, *receive volume of*P_k: The total number of X-matrix elements received by P_k from other processors. That is, $RV(P_k) = \sum_\ell \sigma(P_\ell, P_k)$. This is equal to $s \times ncs(R_k)$.

Note that the total volume of communication is equal to $\sum_k SV(P_k) = \sum_k RV(P_k)$. This is also equal to s times the total number of nonzero column segments in all off-diagonal blocks, i.e., $s \times ncs(A_{BL})$.

In this study, we extend the sparse matrix partitioning problem in which the only objective is to minimize the total communication volume, by introducing four more minimization objectives which are defined on the following metrics:

1. $\max_k SV(P_k)$: maximum send volume of processors (equivalent to maximum $s \times ncs(C_k)$),
2. $\max_k RV(P_k)$: maximum receive volume of processors (equivalent to maximum $s \times ncs(R_k)$),
3. $\max_k (SV(P_k) + RV(P_k))$: maximum sum of send and receive volumes of processors (equivalent to maximum $s \times (ncs(C_k) + ncs(R_k))$),
4. $\max_k \max\{SV(P_k), RV(P_k)\}$: maximum of maximum of send and receive volumes of processors (equivalent to maximum $s \times \max\{ncs(C_k), ncs(R_k)\}$).

Under the objective of minimizing the total communication volume, minimizing one of these volume-based metrics (e.g., $\max_k SV(P_k)$) relates to *minimizing imbalance* on the respective quantity (e.g., imbalance on $SV(P_k)$ values). For instance, the imbalance on $SV(P_k)$ values is defined as

$$\frac{\max_k SV(P_k)}{\sum_k SV(P_k)/K}.$$

Here, the expression in the denominator denotes the average send volume of processors.

A parallel application may necessitate one or more of these metrics to be minimized. These metrics are considered besides total volume since minimization of them is plausible only when total volume is also minimized as mentioned above. Hereinafter, these metrics except total volume are referred to as volume-based metrics.

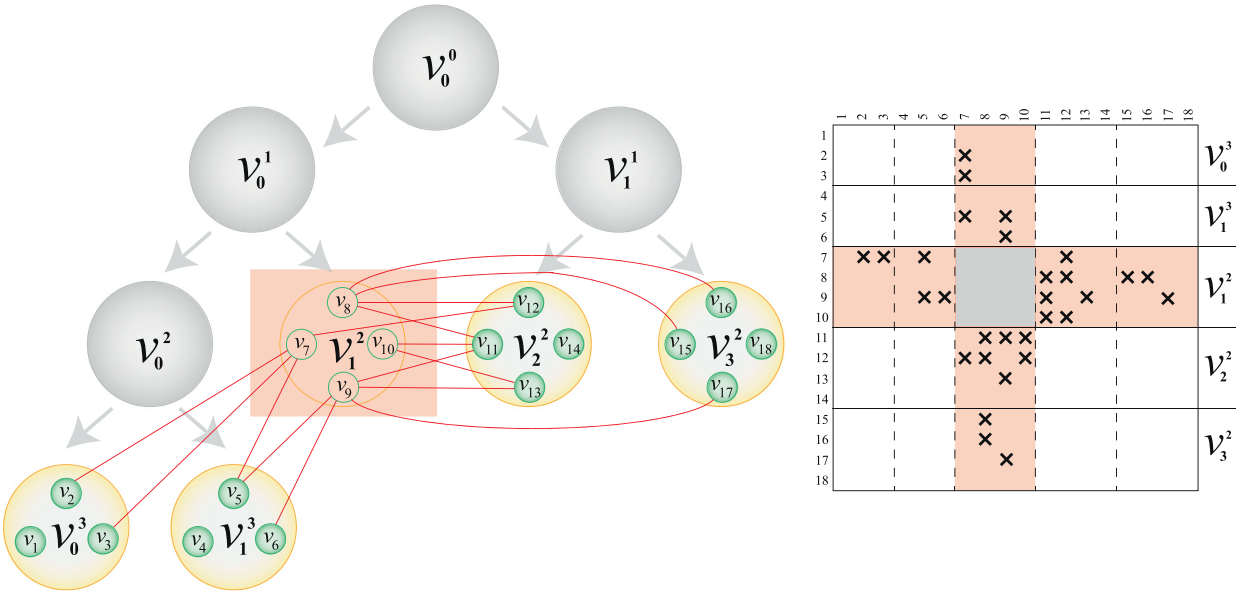

Fig. 2. The state of the RB tree prior to bipartitioning G_1^2 and the corresponding sparse matrix. Among the edges and nonzeros, only the external (cut) edges of $\mathcal{V}_1^2$ and their corresponding nonzeros are shown.

4. Models for minimizing multiple volume-based metrics

This section describes the proposed graph and hypergraph partitioning models for addressing volume-based cost metrics defined in the previous section. Our models have the capability of addressing a single, a combination or all of these metrics *simultaneously* in a single phase. Moreover, they have the flexibility of handling custom metrics based on volume other than the already defined four metrics. Our approach relies on the widely adopted recursive bipartitioning (RB) framework utilized in a breadth-first manner and can be realized by any graph and hypergraph partitioning tool.

4.1. Recursive bipartitioning

In the RB paradigm, the initial graph/hypergraph is partitioned into two subgraphs/subhypergraphs. These two subgraphs/subhypergraphs are further bipartitioned recursively until K parts are obtained. This process forms a full binary tree, which we refer to as an RB tree, with $\lg_2 K$ levels, where K is a power of 2. Without loss of generality, graphs and hypergraphs at level r of the RB tree are numbered from left to right and denoted as $G_0^r, \ldots, G_{2^r-1}^r$ and $\mathcal{H}_0^r, \ldots, \mathcal{H}_{2^r-1}^r$, respectively. From bipartition $\Pi(G_k^r) = \{\mathcal{V}_{2k}^{r+1}, \mathcal{V}_{2k+1}^{r+1}\}$ of graph $G_k^r = (\mathcal{V}_k^r, \mathcal{E}_k^r)$, two vertex-induced subgraphs $G_{2k}^{r+1} = (\mathcal{V}_{2k}^{r+1}, \mathcal{E}_{2k}^{r+1})$ and $G_{2k+1}^{r+1} = (\mathcal{V}_{2k+1}^{r+1}, \mathcal{E}_{2k+1}^{r+1})$ are formed. All cut edges in $\Pi(G_k^r)$ are excluded from the newly formed subgraphs. From bipartition $\Pi(\mathcal{H}_k^r) = \{\mathcal{V}_{2k}^{r+1}, \mathcal{V}_{2k+1}^{r+1}\}$ of hypergraph $\mathcal{H}_k^r = (\mathcal{V}_k^r, \mathcal{N}_k^r)$, two vertex-induces subhypergraphs are formed similarly. All cut nets in $\Pi(\mathcal{H}_k^r)$ are split to correctly encode the cutsize metric [21].

4.2. Graph model

Consider the use of the RB paradigm for partitioning the standard graph representation $G = (\mathcal{V}, \mathcal{E})$ of A for row-parallel $Y = AX$ to obtain a K-way partition. We assume that the RB proceeds in a breadth-first manner and RB process is at level r prior to bipartitioning kth graph G_k^r. Observe that the RB process up to this bipartitioning already induces a K'-way partition $\Pi(G) = \{\mathcal{V}_0^{r+1}, \ldots, \mathcal{V}_{2k-1}^{r+1}, \mathcal{V}_k^r, \ldots, \mathcal{V}_{2^r-1}^r\}$. $\Pi(G)$ contains $2k$ vertex parts from level $r+1$ and $2^r - k$ vertex parts from level r, making $K' = 2^r + k$. After bipartitioning G_k^r, a $(K'+1)$-way partition $\Pi'(G)$ is obtained which contains $\mathcal{V}_{2k}^{r+1}$ and $\mathcal{V}_{2k+1}^{r+1}$ instead of $\mathcal{V}_k^r$. For example, in Fig. 2, the RB process is at level $r = 2$ prior to bipartitioning $G_1^2 = (\mathcal{V}_1^2, \mathcal{E}_1^2)$, so, the current state of the RB induces a five-way partition $\Pi(G) = \{\mathcal{V}_0^3, \mathcal{V}_1^3, \mathcal{V}_1^2, \mathcal{V}_2^2, \mathcal{V}_3^2\}$. Bipartitioning G_1^2 induces a six-way partition $\Pi'(G) = \{\mathcal{V}_0^3, \mathcal{V}_1^3, \mathcal{V}_2^3, \mathcal{V}_3^3, \mathcal{V}_2^2, \mathcal{V}_3^2\}$. P_k^r denotes the group of processors which are responsible for performing the tasks represented by the vertices in $\mathcal{V}_k^r$. The send and receive volume definitions $SV(P_k)$ and $RV(P_k)$ of individual processor P_k are easily extended to $SV(P_k^r)$ and $RV(P_k^r)$ for processor group P_k^r.

We first formulate the send volume of the processor group P_k^r to all other processor groups corresponding to vertex parts in $\Pi(G)$. Let *connectivity set* of vertex $v_i \in \mathcal{V}_k^r$, $Con(v_i)$, denote the subset of parts in $\Pi(G) - \{\mathcal{V}_k^r\}$ in which v_i has at least one

neighbor. That is,

$$Con(v_i) = \{\mathcal{V}_\ell^t \in \Pi(G) : Adj(v_i) \cap \mathcal{V}_\ell^t \neq \emptyset\} - \{\mathcal{V}_k^r\},$$

where t is either r or $r+1$. Vertex v_i is boundary if $Con(v_i) \neq \emptyset$, and once v_i becomes boundary, it remains boundary in all further bipartitionings. For example, in Fig. 2, $Con(v_9) = \{\mathcal{V}_1^3, \mathcal{V}_2^2, \mathcal{V}_3^2\}$. $Con(v_i)$ signifies the communication operations due to v_i, where P_k^r sends row i of X to processor groups that correspond to the parts in $Con(v_i)$. The send load associated with v_i is denoted by $sl(v_i)$ and is equal to

$$sl(v_i) = s \times |Con(v_i)|$$

The total send volume of P_k^r is then equal to the sum of the send loads of all vertices in $\mathcal{V}_k^r$, i.e., $SV(P_k^r) = \sum_{v_i \in \mathcal{V}_k^r} sl(v_i)$. In Fig. 2, the total send volume of P_1^2 is equal to $sl(v_7) + sl(v_8) + sl(v_9) + sl(v_{10}) = 3s + 2s + 3s + s = 9s$. Therefore, during bipartitioning G_k^r, minimizing

$$\max\left\{ \sum_{v_i \in \mathcal{V}_{2k}^{r+1}} sl(v_i), \sum_{v_i \in \mathcal{V}_{2k+1}^{r+1}} sl(v_i) \right\}$$

is equivalent to minimizing the maximum send volume of the two processor groups P_{2k}^{r+1} and P_{2k+1}^{r+1} to the other processor groups that correspond to the vertex parts in $\Pi(G)$.

In a similar manner, we formulate the receive volume of the processor group P_k^r from all other processor groups corresponding to the vertex parts in $\Pi(G)$. Observe that for each boundary $v_j \in \mathcal{V}_\ell^t$ that has at least one neighbor in $\mathcal{V}_k^r$, P_k^r needs to receive the corresponding row j of X from P_ℓ^t. For instance, in Fig. 2, since $v_5 \in \mathcal{V}_1^3$ has two neighbors in $\mathcal{V}_1^2$, P_1^2 needs to receive the corresponding fifth row of X from P_1^3. Hence, P_k^r receives a subset of X-matrix rows whose cardinality is equal to the number of vertices in $\mathcal{V} - \mathcal{V}_k^r$ that have at least one neighbor in $\mathcal{V}_k^r$, i.e., $|\{v_j \in \{\mathcal{V} - \mathcal{V}_k^r\} : v_i \in \mathcal{V}_k^r \text{ and } e_{ji} \in \mathcal{E}\}|$. The size of this set for $\mathcal{V}_1^2$ in Fig. 2 is equal to 10. Note that each such v_j contributes s words to the receive volume of P_k^r. This quantity can be captured by evenly distributing it among v_j's neighbors in $\mathcal{V}_k^r$. In other words, a vertex $v_j \in \mathcal{V}_l^t$ that has at least one neighbor in $\mathcal{V}_k^r$ contributes $s/|Adj(v_j) \cap \mathcal{V}_k^r|$ to the receive load of each vertex $v_i \in \{Adj(v_j) \cap \mathcal{V}_k^r\}$. The receive load of v_i, denoted by $rl(v_i)$, is given by considering all neighbors of v_i that are not in $\mathcal{V}_k^r$, that is,

$$rl(v_i) = \sum_{e_{ji} \in \mathcal{E} \text{ and } v_j \in \mathcal{V}_\ell^t} \frac{s}{|Adj(v_j) \cap \mathcal{V}_k^r|}.$$

The total receive volume of P_k^r is then equal to the sum of the receive loads of all vertices in $\mathcal{V}_k^r$, i.e., $RV(P_k^r) = \sum_{v_i \in \mathcal{V}_k^r} rl(v_i)$. In Fig. 2, the vertices v_{11}, v_{12}, v_{15} and v_{16} respectively contribute $s/3$, $s/2$, s and s to the receive load of v_8, which makes $rl(v_8) = 17s/6$. The total receive volume of P_1^2 is equal to $rl(v_7) + rl(v_8) + rl(v_9) + rl(v_{10}) = 3s + 17s/6 + 10s/3 + 5s/6 = 10s$. Note that this is also equal to the s times the number of neighboring vertices of $\mathcal{V}_1^2$ in $\mathcal{V} - \mathcal{V}_1^2$. Therefore, during bipartitioning G_k^r, minimizing

$$\max\left\{ \sum_{v_i \in \mathcal{V}_{2k}^{r+1}} rl(v_i), \sum_{v_i \in \mathcal{V}_{2k+1}^{r+1}} rl(v_i) \right\}$$

is equivalent to minimizing maximum receive volume of the two processor groups P_{2k}^{r+1} and P_{2k+1}^{r+1} from the other processor groups that correspond to the vertex parts in $\Pi(G)$.

Although these two formulations correctly encapsulate the send/receive volume loads of P_{2k}^{r+1} and P_{2k+1}^{r+1} to/from all other processor groups in $\Pi(G)$, they overlook the send/receive volume loads between these two processor groups. Our approach tries to refrain from this small deviation by immediately utilizing the newly generated partition information while computing volume loads in the upcoming bipartitionings. That is, the computation of send/receive loads for bipartitioning G_k^r utilizes the most recent K'-way partition information, i.e., $\Pi(G)$. This deviation becomes negligible with increasing number of subgraphs in the latter levels of the RB tree. The encapsulation of send/receive volumes between P_{2k}^{r+1} and P_{2k+1}^{r+1} during bipartitioning G_k^r necessitates implementing a new partitioning tool.

Algorithm 1 presents the computation of send and receive loads of vertices in G_k^r prior to its bipartitioning. As its inputs, the algorithm needs the original graph $G = (\mathcal{V}, \mathcal{E})$, graph $G_k^r = (\mathcal{V}_k^r, \mathcal{E}_k^r)$, and the up-to-date partition information of vertices, which is stored in *part* array of size $V = |\mathcal{V}|$. To compute the send load of a vertex $v_i \in \mathcal{V}_k^r$, it is necessary to find the set of parts in which v_i has at least one neighbor. For this purpose, for each $v_j \notin \mathcal{V}_k^r$ in $Adj(v_i)$, $Con(v_i)$ is updated with the part that v_j is currently in (lines 2–4). $Adj(\cdot)$ lists are the adjacency lists of the vertices in the original graph G. Next, the send load of v_i, $sl(v_i)$, is simply set to s times the size of $Con(v_i)$ (line 5). To compute the receive load of $v_i \in \mathcal{V}_k^r$, it is necessary to visit the neighbors of v_i that are not in $\mathcal{V}_k^r$. For each such neighbor v_j, the receive load of v_i, $rl(v_i)$, is updated by adding v_i's share of receive load due to v_j, which is equal to $s/|Adj(v_j) \cap \mathcal{V}_k^r|$ (lines 6–8). Observe that only the boundary vertices in $\mathcal{V}_k^r$ will have nonzero volume loads at the end of this process.

Algorithm 2 presents the overall partitioning process to obtain a K-way partition utilizing breadth-first RB. For each level r of the RB tree, the graphs in this level are bipartitioned from left to right, G_0^r to $G_{2^r-1}^r$ (lines 3–4). Prior to bipartitioning of G_k^r, the send load and the receive load of each vertex in G_k^r are computed with `GRAPH-COMPUTE-VOLUME-LOADS`

Algorithm 1 GRAPH-COMPUTE-VOLUME-LOADS.

Input: $G = (\mathcal{V}, \mathcal{E})$, $G_k^r = (\mathcal{V}_k^r, \mathcal{E}_k^r)$, $part$, s

1 **foreach** boundary vertex $v_i \in \mathcal{V}_k^r$ **do**

▷ Compute the send load
2 $Con(v_i) \leftarrow \emptyset$
3 **foreach** boundary vertex $v_j \in Adj(v_i)$ **and** $v_j \notin \mathcal{V}_k^r$ **do**
4 $Con(v_i) \leftarrow Con(v_i) \cup \{part[v_j]\}$
5 $sl(v_i) \leftarrow s \times |Con(v_i)|$

▷ Compute the receive load
6 $rl(v_i) \leftarrow 0$
7 **foreach** boundary vertex $v_j \in Adj(v_i)$ **and** $v_j \notin \mathcal{V}_k^r$ **do**
8 $rl(v_i) \leftarrow rl(v_i) + s/|Adj(v_j) \cap \mathcal{V}_k^r|$

Algorithm 2 GRAPH-PARTITION.

Input: $G = (\mathcal{V}, \mathcal{E})$, K, s
1 Let $part$ be an array of size $|\mathcal{V}|$
2 $G_0^0 \leftarrow G$
3 **for** $r \leftarrow 0$ **to** $\lg_2 K - 1$ **do**
4 **for** $k \leftarrow 0$ **to** $2^r - 1$ **do**
5 GRAPH-COMPUTE-VOLUME-LOADS$(G, G_k^r, part, s)$
6 Set volume-based vertex weights using $sl(v_i)$ and/or $rl(v_i)$
7 Bipartition $G_k^r = (\mathcal{V}_k^r, \mathcal{E}_k^r)$ to obtain $\Pi(G_k^r) = (\mathcal{V}_{2k}^{r+1}, \mathcal{V}_{2k+1}^{r+1})$
8 Form new graphs G_{2k}^{r+1} and G_{2k+1}^{r+1} using $\Pi(G_k^r)$
9 Update $part$ using $\Pi(G_k^r)$
10 **return** $part$

(line 5). Recall that in the original sparse matrix partitioning with graph model, each vertex v_i has a single weight $w^1(v_i)$, which represents the computational load associated with it. To address the minimization of maximum send/receive volume, we associate an extra weight with each vertex. Specifically, to minimize the maximum send volume, the send load of v_i is assigned as its second weight, i.e., $w^2(v_i) = sl(v_i)$. In a similar manner, to minimize the maximum receive volume, the receive load of v_i is assigned as its second weight, i.e., $w^2(v_i) = rl(v_i)$. Observe that only the boundary vertices have nonzero second weights. Next, G_k^r is bipartitioned to obtain $\Pi(G_k^r) = \{\mathcal{V}_{2k}^{r+1}, \mathcal{V}_{2k+1}^{r+1}\}$ using *multi-constraint* partitioning to handle multiple vertex weights (line 7). Then, two new subgraphs G_{2k}^{r+1} and G_{2k+1}^{r+1} are formed from G_k^r using $\Pi(G_k^r)$ (line 8). In partitioning, minimizing imbalance on the second part weights corresponds to minimizing imbalance on send (receive) volume if these weights are set to send (receive) loads. In other words, under the objective of minimizing total volume in this bipartitioning, minimizing

$$\frac{\max\{W^2(\mathcal{V}_{2k}^{r+1}), W^2(\mathcal{V}_{2k+1}^{r+1})\}}{(W^2(\mathcal{V}_{2k}^{r+1}) + W^2(\mathcal{V}_{2k+1}^{r+1}))/2}$$

relates to minimizing $\max\{SV(P_{2k}^{r+1}), SV(P_{2k+1}^{r+1})\}$ ($\max\{RV(P_{2k}^{r+1}), RV(P_{2k+1}^{r+1})\}$) if the second weights are set to send (receive) loads. Then *part* array is updated after each bipartitioning to keep track of the most up-to-date partition information of all vertices (line 9). Finally, the resulting K-way partition information is returned in *part* array (line 10). Note that in the final K-way partition, processor group $P_k^{\lg_2 K}$ denotes the individual processor P_k, for $0 \leq k \leq K-1$.

In order to efficiently maintain the send and receive loads of vertices, we make use of the RB paradigm in a breadth-first order. Since these loads are not known in advance and depend on the current state of the partitioning, it is crucial to act proactively by avoiding high imbalances on them. Compare this to computational loads of vertices, which is known in advance and remains the same for each vertex throughout the partitioning. Hence, utilizing a breadth-first or a depth-first RB does not affect the quality of the obtained partition in terms of computational load. We prefer a breadth-first RB to a depth-first RB for minimizing volume-based metrics since operating on the parts that are at the same level of the RB tree (in order to compute send/receive loads) prevents the possible deviations from the target objective(s) by quickly adapting the current available partition to the changes that occur in send/receive volume loads of vertices.

The described methodology addresses the minimization of $\max_k SV(P_k)$ or $\max_k RV(P_k)$ separately. After computing the send and receive loads, we can also easily minimize $\max_k(SV(P_k) + RV(P_k))$ by associating the second weight of each vertex with the sum of send and receive loads, i.e., $w^2(v_i) = sl(v_i) + rl(v_i)$. For the minimization of $\max_k \max\{SV(P_k), RV(P_k)\}$, either

the send loads or the receive loads are targeted at each bipartitioning. For this objective, the decision of minimizing which measure in a particular bipartitioning can be given according to the imbalance values on these measures for the current overall partition. If the imbalance on send loads is larger, then the second weights of vertices are set to the send loads, whereas if the imbalance on receive loads is larger, then the second weights of vertices are set to the receive loads. In this way, we try to control the high imbalance in $\max_k RV(P_k)$ that is likely to occur when minimizing solely $\max_k SV(P_k)$, and vice versa.

Apart from minimizing a single volume-based metric, our approach is very flexible in the sense that it can address *any* combination of volume-based metrics simultaneously. This is achieved by simply associating even more weights with vertices. For instance, if one wishes to minimize $\max_k SV(P_k)$ and $\max_k RV(P_k)$ at the *same* time, it is enough to use two more weights in addition to the computational weight by setting $w^2(v_i) = sl(v_i)$ and $w^3(v_i) = rl(v_i)$ accordingly. Observe that one can utilize as many weights as desired with vertices. However, associating several weights with vertices does not come for free and has practical implications, which we address in the next section. Another important useful feature of our model is that, once the send and the receive loads are in hand, it is possible to define *custom* metrics regarding volume to best suit the needs of the target parallel application. For instance, although not sensible and just for demonstration purposes, one can address objectives like $\max_k \min\{SV(P_k), RV(P_k)\}$, $\max_k (SV(P_k)^2 + RV(P_k))$, etc. For our work, we have chosen the metrics which we believe to be the most crucial and definitive for a general application realized in message passing paradigm.

The arguments made so far are valid for the graph representation of symmetric matrices. To handle nonsymmetric matrices, it is necessary to modify the adjacency list definition by defining two adjacency lists for each vertex. This is because, the nonzeros a_{ij} and a_{ji} have different communication requirements in nonsymmetric matrices. Specifically, a nonzero a_{ji} signifies a send operation from P_k to P_ℓ no matter whether a_{ij} is nonzero or not, where v_i and v_j are respectively mapped to processors P_k and P_ℓ. Hence, the adjacency list definition regarding the send operations for v_i becomes $Adj_S(v_i) = \{v_j : a_{ji} \neq 0\}$. In a dual manner, a nonzero a_{ij} signifies a receive operation from P_ℓ to P_k no matter whether a_{ji} is nonzero or not. Thus, the adjacency list definition regarding the receive operations for v_i becomes $Adj_R(v_i) = \{v_j : a_{ij} \neq 0\}$. Accordingly, in Algorithm 1, the adjacency lists in lines 4, 7, and 8 need to be replaced with $Adj_S(v_i)$, $Adj_R(v_i)$, and $Adj_S(v_j)$, respectively, to handle nonsymmetric matrices. Note that for all $v_i \in \mathcal{V}$, if the matrix is symmetric, then $Adj_S(v_i) = Adj_R(v_i) = Adj(v_i)$.

Complexity analysis. Compared to the original RB-based graph partitioning model, our approach additionally requires computing and setting volume loads (lines 5–6). Hence, we only focus on the runtime of these operations to analyze the additional cost introduced by our method. When we consider `GRAPH-COMPUTE-VOLUME-LOADS` for a single bipartitioning of graph G^r_k, the adjacency list of each boundary vertex ($Adj(v_i)$) in this graph is visited once. Note that although the lines 4 and 8 in this algorithm could be realized in a single for-loop, the computation of loads are illustrated with two distinct for-loops for the ease of presentation. In a single level of the RB tree (lines 4–9 of `GRAPH-PARTITION`), each edge e_{ij} of G is considered at most twice, once for computing loads of v_i, and once for computing loads of v_j. The efficient computation of $|Con(v_i)|$ in line 4 and $|Adj(v_j) \cap \mathcal{V}^r_k|$ in line 8 requires special attention. By maintaining an array of size $O(K)$ for each boundary vertex, we can retrieve these values in $O(1)$ time. In the computation of the send loads, the ℓth element of this array is one if v_i has neighbor(s) in $\mathcal{V}^r_\ell$, and zero otherwise. In the computation of the receive loads, it stands for the number of neighbors of v_i in $\mathcal{V}^r_\ell$. Since both of these operations can be performed in $O(1)$ time with the help of these arrays, the computation of volume loads in a single level takes $O(E)$ time in `GRAPH-PARTITION` (line 5). For lines 6 and 9, each vertex in a single level is visited only once, which takes $O(V)$ time. Hence, our method introduces an additional $O(V + E) = O(E)$ cost to each level of the RB tree. Note that $O(E) = O(nnz(A))$, where $nnz(A)$ is the number of nonzeros in A. The total runtime due to handling of volume-based loads thus becomes $O(E \lg_2 K)$. The space complexity of our algorithm is $O(V_B K)$ due to the arrays used to handle connectivity information of boundary vertices, where $V_B \subseteq \mathcal{V}$ denotes the set of boundary vertices in the final K-way partition. In practice $|\mathcal{V}_B|$ and K are much smaller than $|\mathcal{V}|$. In addition, for the send loads, these arrays contain only binary information which can be stored as bit vectors. Also note that the multi-constraint partitioning is expected to be costlier than its single-constraint counterpart.

4.3. Hypergraph model

Consider the use of the RB paradigm for partitioning the hypergraph representation $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ of A for row-parallel $Y = AX$ to obtain a K-way partition (Section 2.3) . Without loss of generality, we assume that the communication task represented by net n_i is performed by the processor that v_i is assigned to.

We assume that the assumptions made for the graph model also applies here so that we are at the stage of bipartitioning $\mathcal{H}^r_k$ for a given K'-way partition $\Pi(\mathcal{H})$. The hypergraph model for minimizing volume-based metrics resembles to the graph model. The only differences are the definitions regarding the send and receive loads of vertices. Recall that in the hypergraph model, n_i represents the communication task in which the processor that owns $v_i \in \mathcal{V}^r_k$ sends row i of X to the processors that correspond to the parts in $\Lambda(n_i) - \{\mathcal{V}^r_k\}$. So, in the hypergraph model, the connectivity set of vertex v_i is defined as the number of parts that n_i connects other than V^r_k, that is,

$$Con(v_i) = \{\mathcal{V}^t_\ell \in \Pi(\mathcal{H}) : Pins(n_i) \cap \mathcal{V}^t_\ell \neq \emptyset\} - \mathcal{V}^r_k.$$

Hence, in the hypergraph model, the send load $sl(v_i)$ of vertex v_i is given by

$$sl(v_i) = s \times |Con(v_i)| = s \times (\lambda(n_i) - 1).$$

Algorithm 3 HYPERGRAPH-COMPUTE-VOLUME-LOADS.

Input: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\mathcal{H}_k^r = (\mathcal{V}_k^r, \mathcal{N}_k^r)$, $part$, s

1 **foreach** boundary vertex $v_i \in \mathcal{V}_k^r$ **do**

▷ Compute the send load
2 $Con(v_i) \leftarrow \emptyset$
3 **foreach** boundary vertex $v_j \in Pins(n_i)$ **and** $v_j \notin \mathcal{V}_k^r$ **do**
4 $Con(v_i) \leftarrow Con(v_i) \cup \{part[v_j]\}$
5 $sl(v_i) \leftarrow s \times |Con(v_i)|$

▷ Compute the receive load
6 $rl(v_i) \leftarrow 0$
7 **foreach** $n_j \in Nets(v_i) - \{n_i\}$ **and** $v_j \notin \mathcal{V}_k^r$ **do**
8 $rl(v_i) \leftarrow rl(v_i) + s/|Pins(n_j) \cap \mathcal{V}_k^r|$

Consider the communication task represented by a net n_j that connects $v_i \in \mathcal{V}_k^r$, where the vertex v_j associated with n_j is in $\mathcal{V}_\ell^t$. Recall that $\mathcal{V}_\ell^t$ is a part in $\Pi(\mathcal{H})$ other than V_k^r, where t is either r or $r+1$. For this task, the processor groups that correspond to the parts in $\Lambda(n_j) - \{\mathcal{V}_\ell^t\}$ receive row j of X from P_ℓ^t. This receive load of s words from P_ℓ^t to P_k^r is evenly distributed among the vertices in $Pins(n_j) \cap \mathcal{V}_k^r$. That is, n_j contributes $s/|Pins(n_j) \cap \mathcal{V}_k^r|$ amount to the receive load of v_i. Hence, the receive load $rl(v_i)$ of v_i is given by

$$rl(v_i) = \sum_{n_j \in Nets(v_i) - \{n_i\}} \frac{s}{|Pins(n_j) \cap \mathcal{V}_k^r|}.$$

The remaining definitions regarding $SV(P_k^r)$, $RV(P_k^r)$ and the equivalence of minimization of the above-mentioned quantities with the defined metrics for the graph model hold as is for the hypergraph model. The algorithm HYPERGRAPH-COMPUTE-VOLUME-LOADS (Algorithm 3) computes the send and receive loads of vertices in the hypergraph model and resembles to that of graph model (Algorithm 1). In line 3 of this algorithm where we compute the send load of v_i, we traverse pin list of n_i instead of adjacency list of v_i. In line 7 where we compute the receive load of v_i, we traverse the nets that connect v_i instead of its adjacency list and in line 8, the receive load of v_i is updated by taking intersection of $\mathcal{V}_k^r$ with $Pins(n_j)$ instead of with $Adj(v_j)$. To compute a K-way partition of $\mathcal{H}$, Algorithm 2 can be used as is by replacing its graph terminology with the hypergraph terminology.

Complexity analysis. The computation of volume loads in the hypergraph model differs from the graph model only in the sense that instead of visiting the adjacency lists of boundary vertices, the vertices connected by cut nets and the nets connecting boundary vertices are visited. Again, by associating an $O(K)$-size array with each boundary vertex, lines 4 and 8 in HYPERGRAPH-COMPUTE-VOLUME-LOADS can be performed in $O(1)$ time. In the computation of the send loads, each vertex and the vertices connected by the net associated with that vertex are visited at most once in a single level of the RB tree. This requires visiting all vertices and pins of the hypergraph once in a single level in the worst case, which takes $O(V+P)$ time, where $P = \sum_{n \in \mathcal{N}} |Pins(n)|$. In the computation of the receive loads, each vertex and its net list are visited once. This also requires visiting all vertices and pins of the hypergraph once in a single level, which takes $O(V+P)$ time. Hence, our method introduces an additional $O(V+P) = O(P)$ cost to each level of the RB tree. Note that $O(P) = O(nnz(A))$. The total runtime due to handling of volume-based loads thus becomes $O(P \lg_2 K)$. The space complexity is $O(V_B K)$, where $V_B \subseteq \mathcal{V}$ denotes the set of boundary vertices in the final K-way partition. Observe that we introduce the same overhead both in graph and hypergraph models.

4.4. Partitioning tools

The multi-constraint graph and hypergraph partitioning tools associate multiple weights with vertices. These tools allow users to define different maximum allowed imbalance ratios $\epsilon^1, \ldots, \epsilon^C$ for each constraint, where ϵ^c denotes the maximum allowed imbalance ratio on the cth constraint. Recall that in our approach, minimizing the imbalance on a specific weight relates to minimizing the respective volume-based metric. Hence, by using the existing tools within our approach, it is possible to minimize the target volume-based metric(s).

The partitioning tools do not try to minimize the imbalance on a specific constraint. Rather, they aim to stay within the given threshold for any given ϵ^c. For this reason, the imbalance values provided to the tools should be as low as to the degree how much these metrics are important for optimization. Enforcing a very small value on ϵ^c can put a lot of strain on the partitioning tool, which in turn may cause the tool to intolerably loosen its objective. This may increase total volume drastically and make the minimization of target volume-based metrics pointless as they are defined on the amount of volume communicated. For this reason, it is not sensible to use a very small value for ϵ^c.

5. Efficient handling of multiple constraints

In this section, we describe the two drawbacks of using multiple constraints within the context of our model and propose two practical schemes which enhance this model to overcome them.

Our approach introduces as many constraints as needed in order to address the desired volume-based cost metrics. Recall that the volume related weights are nonzero only for the boundary vertices because only these vertices incur communication. Since the objective of minimizing cutsize with partitioners also relates to minimizing the number of boundary vertices, only a small portion of all vertices will have nonzero volume related weights throughout the partitioning process. So, balancing the volume related weights of parts will have much less degree of freedom compared to balancing the computational weights of parts. That is, the partitioner will have difficulty in maintaining balance on volume-related weights of parts because of small number of vertices with nonzero volume-related weights.

Each introduced constraint puts an extra burden on the partitioning tool by restricting the solution space, where the more restricted the solution space, the worse the quality of the solutions generated by the partitioning tool. Hence, the additional constraint(s) used for minimizing volume-based metrics may lead to higher total volume (i.e., cutsize). This also has the side effect on the other factors that determine the overall communication cost, such as increasing contention on the network or increasing the latency overhead.

To address these shortcomings, in Section 5.1, we propose a scheme which selectively utilizes volume-related weights, and in Section 5.2, we propose another scheme which unifies multiple weights.

5.1. Delayed formation of volume loads

In this scheme, we utilize level information in the RB tree to form and make use of the volume related loads in a delayed manner. Specifically, in bipartitionings of the first ρ levels of the RB tree, we allow only a single constraint, i.e., regarding the computational load. In the remaining bipartitionings which belong to the latter $\lg_2 K - \rho$ levels, we consider volume-based metrics by introducing as many constraints as needed. This results in a level-based hybrid scheme in which either a single constraint or multiple constraints are utilized.

Our motivations for adopting this scheme are three-fold. First, we aim to improve the quality of the obtained solutions in terms of total volume by sacrificing from the quality of the volume-based metrics. Recall that the minimization of volume-based metrics is pointless unless the total volume is properly addressed. Next, the total volume changes as the partitioning progresses, and the volume-based metrics are defined over this changing quantity. As the ratio of boundary vertices increases in latter levels of the RB tree, addressing volume-based loads in bipartitionings of these levels leads to more efficient utilization of partitioners. Finally, utilization of volume-based loads in the latter levels rather than the earlier levels of the RB tree prevents the deviations on these loads which are likely to occur in the final solution if these constraints were utilized in the earlier levels rather than the latter levels.

This can be seen as an effort to achieve a tradeoff between minimizing total volume and minimizing target volume-based metrics. If we use multiple constraints in all bipartitionings, the target volume-based metrics will be optimized but the total obtained volume will be relatively high. On the other hand, if we use a single constraint (i.e., computational load), the total volume will be relatively low but the target metrics will not be addressed properly.

5.2. Unified weighting

In this scheme, we utilize only a *single* constraint by unifying multiple loads into a single load through a linear formula. Note that this scheme also refrains from the issue related with boundary vertices since the unified single weight for each vertex becomes almost always nonzero.

In order to use a single weight for vertices, it is required to establish a relation between distinct loads those are of interest. For SpMM, determining the relationship between the computational and communication loads is necessary to accurately estimate a single load for each vertex. In large-scale parallel architectures, per unit communication time is usually greater than per unit computation time. To unify the respective loads, we define a coefficient α that represents the per unit communication time in terms of per unit computation time. This coefficient depends on various factors such as clock rate, properties of the interconnect network, the requirements of the underlying parallel application, etc. The following code snippet constitutes the basic skeleton of the SpMM operations from processor P_k's point of view:

```
...
MPI_Irecv()
MPI_Send()
Perform local computations using A_kk
MPI_Waitall() // Wait all receives to complete
Perform non-local computations using A_kℓ, ℓ ≠ k
...
```

In this implementation, non-blocking receive operation is preferred to enable overlapping local SpMM computations $A_{kk}X_k$ and incoming messages. Blocking send operation is used since the performance gain from overlapping local computations and outgoing messages is very limited. The total load of a vertex v_i in this example can be captured with two

distinct weights, where the first weight $w^1(v_i)$ and the second weight $w^2(v_i)$ respectively represent the computational load and the send load associated with v_i. The receive loads of vertices are neglected for this implementation because of the non-blocking receive operations under the assumption that each processor has enough amount of local computation to overlap with the incoming messages. Then, with α in hand, we can easily unify these weights into a single weight as $w(v_i) = w^1(v_i) + \alpha w^2(v_i)$. Note that for non-boundary vertices $w(v_i) = w^1(v_i)$.

6. Experiments

6.1. Experimental setting

6.1.1. Datasets

We perform our experiments on three datasets. The first dataset is used to compare the proposed graph and hypergraph partitioning models (Sections 4 and 5) against the standard partitioning models (Section 2.3). Note that the standard models address only total volume. The second dataset is used to compare our models against the state-of-the-art partitioner UMPa, which addresses maximum send volume of processors. The third dataset is used to assess the strong and weak scaling performance of our models on multi-source breadth-first search (BFS) by comparing them against a recent two-dimensional partitioning model [12]. Table 1 describes the basic properties of these three datasets.

The first dataset is abbreviated as `ds-general` and contains all the square matrices from UFL Sparse Matrix Collection [42] with at least 5000 rows/columns and between 50,000 and 50,000,000 nonzeros. At the time of the experiment, UFL had 964 such matrices, so `ds-general` contains 964 matrices. We categorized these matrices according to the maximum send volume of processors obtained by the standard partitioning models when they are partitioned among 128, 256, 512 and 1024 processors. The naming for the categories is in the format of `m-Kk-Vv`. Here, $\mathtt{m} \in \{\mathtt{G}, \mathtt{H}\}$ denotes the model, where $\mathtt{m} = \mathtt{G}$ if the partitions are obtained by the standard graph model and $\mathtt{m} = \mathtt{H}$ if the partitions are obtained by the standard hypergraph model. `k` denotes the number of processors, where $\mathtt{k} \in \{128, 256, 512, 1024\}$. `v` denotes the lower bound for the maximum send volume obtained, where $\mathtt{v} \in \{8000, 4000, 2000, 1000, 500, 250, 125, 0\}$. For example, `G-K512-V1000` denotes the set of matrices for which the standard graph model obtains a maximum send volume of at least 1000 units for $K = 512$ processors. Here and hereafter one unit of communication refers to an X-matrix row that contains s words. Our motivations for this categorization are two-fold:

(i) to categorize the matrices according to their likelihood for which the maximum send volume is a bottleneck for parallel performance and
(ii) to facilitate a better performance analysis of the proposed models. The top section of Table 1 displays various important properties of the matrices in those categories as the geometric averages. Note that the categories for the standard graph and hypergraph models are different as the maximum send volume values obtained by them are typically different.

Also note that `m-Kk-V8000` $\subseteq$ `m-Kk-V4000` $\subseteq \cdots \subseteq$ `m-Kk-V0` and `m-Kk-V0` contains all the matrices in `ds-general`, for any `m` and `k`. "Avg max row/col sparsity" column in the top section of Table 1 denotes the maximum number of nonzeros in a row or column divided by the number of rows/columns in the respective matrix, averaged over all the matrices in the respective category.

For the comparison against UMPa, we run our models on the matrices in Table 1 of [32], the work that propose UMPa. We use these matrices for our experiments since UMPa is not publicly available. The matrices in this dataset are obtained from 10th DIMACS Implementation Challenge [43] and contains 38 matrices from eight different classes. We exclude two synthetic matrices, namely `1,280,000` and `320,000`, in our experiments as the communicated items in these matrices have different sizes and the communication load formulation utilized in our models does not support varying sizes of communicated items. We abbreviate the resulting dataset of 36 matrices as `ds-dimacs`. For the matrices in this dataset, the number of rows/columns is between 100,000 and 1,585,478 and the number of nonzeros is between 119,666 and 38,354,076. The properties of the matrices in this dataset are given in the middle section of Table 1.

The third dataset is abbreviated as `ds-large` and contains five of the six largest matrices in UFL Sparse Matrix Collection [42]. The properties of the matrices in this dataset are given in the bottom section of Table 1. These are larger than the matrices in `ds-general` and `ds-dimacs`, with the number of rows/columns between 22.7 million and 118.1 million and the number of nonzeros between 640 million and 1.15 billion. We experiment with larger number of processors up to 2048 on this dataset.

6.1.2. Implementation and parallel systems

The parallel SpMM and multi-source BFS kernels are implemented within a parallel library [44] in C that uses MPI for interprocess communication. We use two systems in our experiments. The first system is a Cray XC40 machine. A node on this machine consists of 24 cores (two 12-core Intel Haswell Xeon processors) with 2.5 GHz clock frequency and 128 GB memory. The nodes are connected with a high speed Dragonfly network topology called CRAY Aries. The second system is used for scalability analysis and is a Lenovo NeXtScale machine. A node on this machine consists of 28 cores (two 14-core Intel Haswell Xeon processors) with 2.6 GHz clock frequency and 64 GB memory. The nodes are connected with an Infiniband non-blocking tree network topology.

Table 1
The properties of three datasets used in experiments. The values for `ds-general` are the averages of the matrices in the respective category, while the values for the other two datasets are the individual values for each matrix.

`ds-general` (for comparison against the standard models)

	Matrix category	Number of matrices	Avg number of rows/cols	Avg number of nonzeros	Avg max row/col sparsity	Matrix category	Number of matrices	Avg number of rows/cols	Avg number of nonzeros	Avg max row/col sparsity
categories obtained by G-TV	G-K128-V8000	50	303K	8,903K	0.0443	G-K512-V8000	32	350K	10,279K	0.0746
	G-K128-V4000	103	191K	5,513K	0.0233	G-K512-V4000	54	240K	8,296K	0.0598
	G-K128-V2000	167	175K	4,328K	0.0122	G-K512-V2000	98	179K	6,083K	0.0302
	G-K128-V1000	287	129K	3,067K	0.0099	G-K512-V1000	175	159K	4,304K	0.0146
	G-K128-V500	462	113K	2,487K	0.0056	G-K512-V500	329	108K	2,534K	0.0134
	G-K128-V250	646	83K	1,566K	0.0049	G-K512-V250	553	84K	1,769K	0.0084
	G-K128-V125	808	73K	1,227K	0.0040	G-K512-V125	709	73K	1,394K	0.0055
	G-K128-V0	964	60K	907K	0.0032	G-K512-V0	964	60K	907K	0.0032
	G-K256-V8000	40	303K	9,832K	0.0915	G-K1024-V8000	23	312K	9,907K	0.1171
	G-K256-V4000	73	195K	7,089K	0.0362	G-K1024-V4000	47	271K	8,053K	0.0867
	G-K256-V2000	135	182K	5,136K	0.0154	G-K1024-V2000	78	168K	5,885K	0.0628
	G-K256-V1000	200	149K	3,943K	0.0138	G-K1024-V1000	174	120K	3,216K	0.0360
	G-K256-V500	404	123K	2,851K	0.0063	G-K1024-V500	313	78K	1,520K	0.0330
	G-K256-V250	574	91K	1,874K	0.0057	G-K1024-V250	500	85K	1,823K	0.0105
	G-K256-V125	770	74K	1,279K	0.0045	G-K1024-V125	650	75K	1,502K	0.0068
	G-K256-V0	964	60K	907K	0.0032	G-K1024-V0	964	60K	907K	0.0032
categories obtained by H-TV	H-K128-V8000	50	185K	8,869K	0.0686	H-K512-V8000	39	155K	8,791K	0.0852
	H-K128-V4000	91	170K	6,253K	0.0250	H-K512-V4000	51	132K	7,726K	0.0848
	H-K128-V2000	167	166K	4,323K	0.0142	H-K512-V2000	89	165K	6,262K	0.0351
	H-K128-V1000	267	123K	3,122K	0.0133	H-K512-V1000	170	154K	4,383K	0.0135
	H-K128-V500	453	112K	2,478K	0.0059	H-K512-V500	303	111K	2,895K	0.0115
	H-K128-V250	624	86K	1,686K	0.0049	H-K512-V250	554	84K	1,700K	0.0078
	H-K128-V125	801	72K	1,246K	0.0041	H-K512-V125	696	74K	1,412K	0.0058
	H-K128-V0	964	60K	907K	0.0032	H-K512-V0	964	60K	907K	0.0032
	H-K256-V8000	41	152K	9,447K	0.0893	H-K1024-V8000	38	134K	6,529K	0.1061
	H-K256-V4000	69	155K	7,494K	0.0519	H-K1024-V4000	50	124K	6,564K	0.1042
	H-K256-V2000	128	164K	5,191K	0.0177	H-K1024-V2000	75	151K	6,240K	0.0567
	H-K256-V1000	214	135K	3,607K	0.0163	H-K1024-V1000	126	151K	5,090K	0.0267
	H-K256-V500	380	117K	2,808K	0.0071	H-K1024-V500	322	83K	1,654K	0.0261
	H-K256-V250	556	93K	1,965K	0.0053	H-K1024-V250	508	81K	1,644K	0.0103
	H-K256-V125	753	72K	1,300K	0.0048	H-K1024-V125	653	75K	1,470K	0.0067
	H-K256-V0	964	60K	907K	0.0032	H-K1024-V0	967	60K	907K	0.0032

`ds-dimacs` (for comparison against `UMPa` [32])

Matrix name	Number of rows/cols	Number of nonzeros	class	Matrix name	Number of rows/cols	Number of nonzeros	Class
citationCiteseer	268K	2,313K	Citation	rgg_n_2_19_s0	524K	6,540K	RandomGeometric
coAuthorsCiteseer	227K	1,628K	Citation	rgg_n_2_20_s0	1,049K	13,783K	RandomGeometric
coAuthorsDBLP	299K	1,955K	Citation	af_shell10	1,508K	52,672K	Sparse
coPapersCiteseer	434K	32,073K	Citation	af_shell9	505K	17,589K	Sparse
coPapersDBLP	540K	30,491K	Citation	audikw_1	944K	77,652K	Sparse
caidaRouterLevel	192K	1,218K	Clustering	ecology1	1,000K	4,996K	Sparse
cnr-2000	326K	3,216K	Clustering	ecology2	1,000K	4,996K	Sparse
eu-2005	863K	19,235K	Clustering	G3_circuit	1,585K	7,661K	Sparse
G_n_pin_pout	100K	1,002K	Clustering	ldoor	952K	46,522K	Sparse
in-2004	1,383K	16,917K	Clustering	thermal2	1,228K	8,580K	Sparse
pref.Att.	100K	1,000K	Clustering	belgium_osm	1,441K	3,100K	Street
smallworld	100K	1,000K	Clustering	luxembourg_osm	115K	239K	Street
delaunay_n17	131K	786K	Delaunay	144	145K	2,149K	Walshaw
delaunay_n18	262K	1,573K	Delaunay	598a	111K	1,484K	Walshaw
delaunay_n19	524K	3,146K	Delaunay	auto	449K	6,629K	Walshaw
delaunay_n20	1,049K	6,291K	Delaunay	fe_ocean	143K	819K	Walshaw
rgg_n_2_17_s0	131K	1,458K	RandomGeometric	m14b	215K	3,358K	Walshaw
rgg_n_2_18_s0	262K	3,095K	RandomGeometric	wave	156K	2,119K	Walshaw

`ds-large` (for comparison against `2D` [12])

Matrix name	Number of rows/cols	Number of nonzeros	Problem kind
arabic-2005	22,744K	640,000K	directed graph
nlpkkt240	27,994K	760,648K	optimization problem
uk-2005	39,460K	936,364K	directed graph
webbase-2001	118,142K	1,019,903K	directed graph
it-2004	41,292K	1,150,725K	directed graph

The proposed models can be realized with any graph and hypergraph partitioning tool. For models that rely on using multiple constraints (i.e., the proposed model and its delayed version), the tool should support multiple weights on vertices. In our experiments, we used Metis [22] for partitioning graphs and PaToH [21] for partitioning hypergraphs, both in default settings. Metis and PaToH are respectively used to bipartition the graphs and hypergraphs in line 7 of Algorithm 2.

6.1.3. Compared schemes and models

We evaluate six proposed schemes that address the total volume and the maximum send volume simultaneously. Each of these schemes considers the minimization of the maximum send volume of processors (i.e., $\max_k SV(P_k)$) in accordance with the discussions given in Section 5.2. These schemes are as follows:

- `G-TMV`: The proposed graph partitioning model which addresses both Total and Maximum Volume metrics (Section 4.2). This scheme utilizes two weights, one for the computational loads and one for the send volume loads.
- `H-TMV`: The hypergraph counterpart of `G-TMV` (Section 4.3).
- `G-TMVd`: The variant of `G-TMV` with the send loads formed in a *delayed* manner (Section 5.1). ρ is set to $\lceil \lg_2 K/2 \rceil$.
- `H-TMVd`: The hypergraph counterpart of `G-TMVd`.
- `G-TMVu`: The variant of `G-TMV` with *unified* loads (Section 5.2). The coefficient that determines the relation between computational and communication loads is set to $\alpha = 10$.
- `H-TMVu`: The hypergraph counterpart of `G-TMVu`.

The baseline models that we compare our proposed models against are as follows:

- `G-TV`: The standard graph partitioning model which only addresses Total Volume (Section 2.3). Recall that this is the most widely adopted model in the literature for sparse matrix/graph partitioning and there exists a single weight, which is on the computational loads. This scheme refers to the use of Metis as is for K-way partitioning.
- `H-TV`: The standard hypergraph partitioning model which only addresses Total Volume (Section 2.3). This scheme refers to the use of PaToH as is for K-way partitioning.
- `UMPa`: The state-of-the-art partitioner that can minimize multiple communication cost metrics [32]. In our case, we consider $\texttt{UMPa}_{\texttt{MSV}}$ in which the single objective is to minimize the maximum send volume handled by a processor.
- 2D: The two-dimensional partitioning model proposed for parallel level-synchronized BFS [12]. Although this method considers the single-source BFS, it is trivially extended to the multi-source BFS.

For the proposed schemes and the baseline schemes `G-TV` and `H-TV`, we use 10% as the maximum allowed imbalance for each of the constraints. Hereinafter, we will refer to the maximum send volume of processors simply as maximum volume.

In Section 6.2, we compare the six proposed schemes among themselves and against standard models `G-TV` and `H-TV`. In Section 6.3, we compare the best performing proposed scheme for graph and hypergraph models against `UMPa`. In Section 6.4, we compare these best schemes against 2D to analyze their scalability on parallel multi-source BFS.

6.2. Comparison against standard partitioning models

6.2.1. Partitioning results

In this section, we provide the results of the proposed partitioning schemes for both graph and hypergraph models in terms of maximum volume, total volume, maximum number of messages and total number of messages, on dataset `ds-general` following the categorization described in Section 6.1.1. Although bandwidth-related metrics are expected to be more important for parallel SpMM performance, latency-related metrics such as maximum and total number of messages can still affect the performance for certain matrices, hence we include them in our analysis. The results obtained by `G-TMV`, `G-TMVd` and `G-TMVu` are normalized with respect to those of `G-TV` and the results obtained by `H-TMV`, `H-TMVd` and `H-TMVu` are normalized with respect to those of `H-TV`. The obtained normalized values for $K = 1024$ processors are displayed as plots for each of the four metrics separately for graph and hypergraph models, respectively in Figs. 3 and 4. The detailed results for each $K \in \{128, 256, 512, 1024\}$ are given in Tables 2 and 3. In the plots in Figs. 3 and 4, the x-axis denotes the eight respective matrix categories in order of increasing maximum volume whereas the y-axis denotes the normalized values. Each value in the plots and the tables is the geometric average of the normalized values obtained for the matrices in the respective category. For example, the three values reported for category `G-K1024-V2000` in the plot for maximum volume for graph model (the top left one in Fig. 3) denote the geometric averages of the normalized values obtained by `G-TMV`, `G-TMVd` and `G-TMVu` with respect to those obtained by `G-TV` on 78 matrices in this category.

As seen in Fig. 3, `G-TMV`, `G-TMVd` and `G-TMVu` perform better than `G-TV` in terms of maximum volume. These three schemes perform drastically better in the matrices for which maximum volume is a bottleneck for performance. The improvements obtained by all three schemes increase with increasing significance of maximum volume. For example, for category `G-K1024-V500`, `G-TMV`, `G-TMVd` and `G-TMVu` respectively achieve improvements of 21%, 21% and 14%, whereas for category `G-K1024-V8000`, these improvements increase to respectively, 61%, 59% and 39%. In addition, for category `G-K1024-Vv` with varying `v` values of 0, 125, 250, 500, 1000, 2000, 4000 and 8000, the improvement of `G-TMV` over `G-TV` gradually increases as 5%, 10%, 14%, 21%, 33%, 48%, 56% and 61%, respectively. The reason for this increase in the improvement is that there exists more room for improvement in partitioning the matrices for which the standard graph model yields high maximum volume.

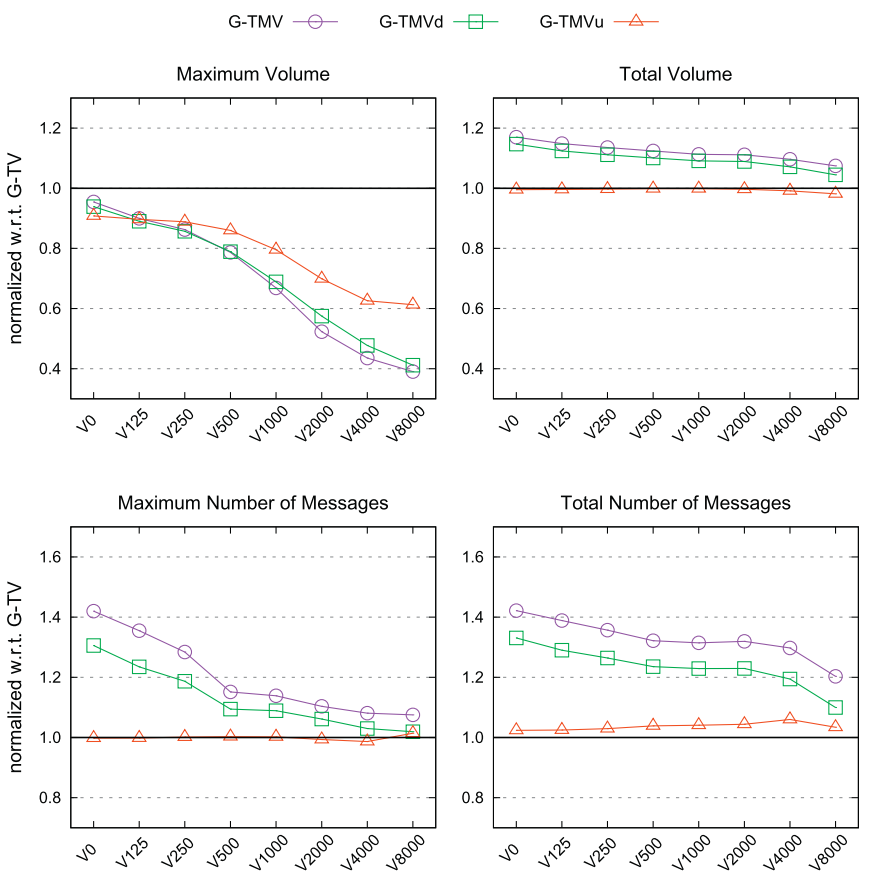

Fig. 3. Maximum volume, total volume, maximum number of messages and total number of messages of the proposed graph schemes G-TMV, G-TMVd and G-TMVu normalized with respect to those of G-TV for $K = 1024$, averaged on matrices in each category G-K1024-Vv.

When we compare G-TMV, G-TMVd and G-TMVu among themselves, G-TMV usually obtains the best improvements in maximum volume since it addresses this metric as a stand-alone objective during the entire partitioning process. In this sense, it differs from G-TMVd and G-TMVu, where G-TMVd addresses maximum volume only in latter bipartitionings and G-TMVu, in order to address maximum volume, uses a single unified constraint that also involves the computational load. Compared to G-TV, G-TMV causes an increase of 17% and 7% in total volume for G-K1024-V0 and G-K1024-V8000, respectively, and G-TMVd causes an increase of 15% and 4%. This is due to the additional constraint utilized in G-TMV and G-TMVd. Recall that utilizing multiple constraints degrades the quality of the solutions obtained by the partitioner in terms of total volume. The reason for smaller degradation rates in total volume for categories with high maximum volume can be attributed to the matrices in these categories having a high total volume, which leaves less room for degradation as a significant fraction of the edges were already in the cut in the partitions obtained by the standard model. G-TMVu, which is proposed to remedy this problem, does not increase the total volume in contrast to G-TMV and G-TMVd, and attains comparable results with G-TV in this metric.

As seen in Fig. 3, G-TMV obtains the worst results in terms of maximum number of messages, followed by G-TMVd. These two schemes in this metric respectively cause increases of 42% and 33% in category G-K1024-V0, and 20% and 10% in category G-K1024-V8000 over G-TV. G-TMVu on the other hand causes a very slight increase over G-TV. The same observations hold for the total number of messages as well. Observe that while maximum volume is substantially improved by G-TMVu, other important factors that determine the communication time such as maximum and total number of messages and total volume are kept almost intact.

As seen in Fig. 4, most of the above observations and discussions made for the graph model hold for the hypergraph model as well. In the hypergraph model for maximum and total volume, the improvement and deterioration rates of the proposed schemes over H-TV are magnified compared to those of the graph models over G-TV. For example, for category H-K1024-V500, H-TMV, H-TMVd and H-TMVu respectively achieve improvements of 28%, 25% and 26%, whereas for category H-K1024-V8000, these improvements become 78%, 66% and 63%. H-TMV causes increases of 28% and 14% in total volume for H-K1024-V0 and H-K1024-V8000, respectively, and H-TMVd causes increases of 25% and 11%. H-TMVu on the other hand obtains slightly smaller total volume than H-TV. In terms of maximum number of messages, H-TMV and

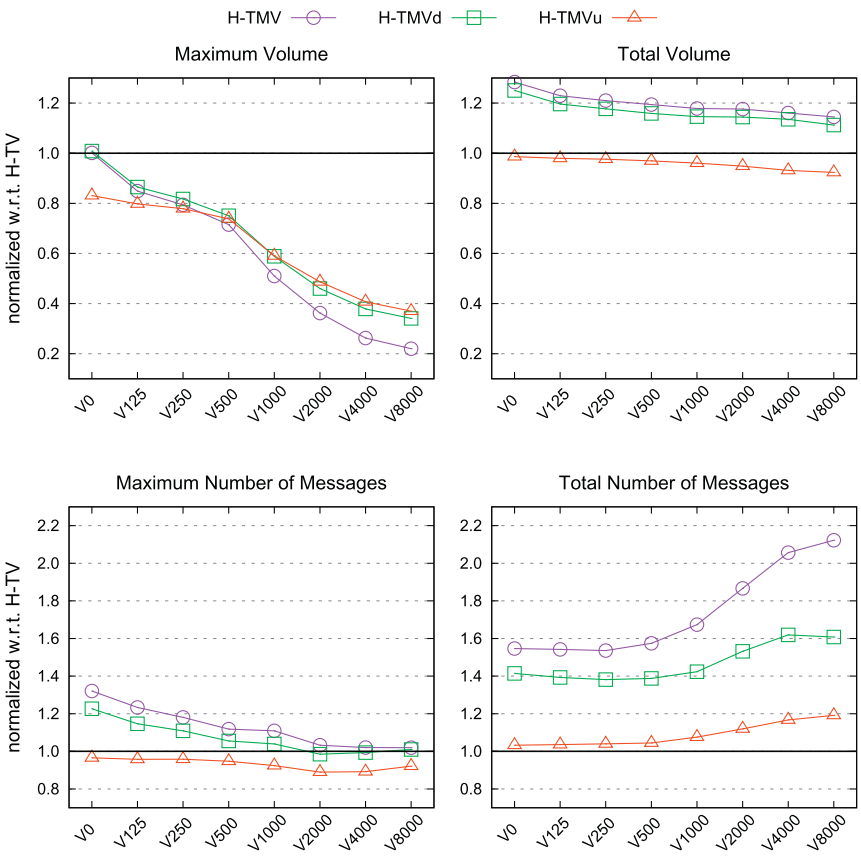

Fig. 4. Maximum volume, total volume, maximum number of messages and total number of messages of the proposed hypergraph schemes `H-TMV`, `H-TMVd` and `H-TMVu` normalized with respect to those of `H-TV` for $K = 1024$, averaged on matrices in each category `H-K1024-Vv`.

`H-TMVd` perform worse than `H-TV` while `H-TMVu` performs slightly better. In terms of total number of messages, all three schemes perform worse than `H-TV`.

As seen in Tables 2 and 3, the improvements of the proposed models in maximum volume increase as the number of processors increases. For example, for category `G-Kk-V2000` with increasing number of `k` processors 128, 256, 512 and 1024, the improvements of `G-TMVu` over `G-TV` respectively increase as 19%, 25%, 27% and 30%. The improvements of `H-TMVu` over `H-TV` for the same setting respectively increase as 18%, 25%, 38% and 51%. The reason for the better performance of the proposed models in larger number of processors is the increased number of bipartitions in which our model is applied throughout the recursive bipartitioning process.

Table 4 displays the partitioning times of the compared schemes for the graph and hypergraph models. For each $K \in \{128, 256, 512, 1024\}$, we present the actual time and the normalized time (with respect to `G-TV` for graph schemes `G-TMV`, `G-TMVd` and `G-TMVu`, and `H-TV` for hypergraph schemes `H-TMV`, `H-TMVd` and `H-TMVu`), which are both geometric averages of 964 matrices. Recall that the proposed schemes introduce the same partitioning overhead of $O(nnz(A))$ in each RB level for both models. This overhead can be extracted from the normalized values of `G-TMVu` over `G-TV` and `H-TMVu` over `H-TV`, and is only 6%–10% for the graph model and 2%–8% for the hypergraph model. For the graph model, among the proposed schemes, `G-TMVu` introduces the lowest partitioning overhead compared to `G-TV`. The worse performances of `G-TMV` and `G-TMVd` compared to `G-TMVu` are expected since multi-constraint partitioning is more expensive than single-constraint partitioning because of the additional feasibility conditions. Although one expects the same for the hypergraph model, the multi-constraint partitioning times of PaToH are surprisingly better than those of single-constraint partitioning.

Judging from the partitioning results, `G-TMVu` among the graph schemes and `H-TMVu` among the hypergraph schemes always achieve significant reductions in maximum volume while keeping the change in other three metrics as small as possible compared to `G-TV` and `H-TV`, respectively. For this reason, we only consider `G-TMVu` and `H-TMVu` among the proposed schemes in the rest of the experimentation.

6.2.2. Parallel SpMM runtime results

We have run parallel SpMM [44] on a Cray XC40 machine for 128, 256 and 512 processors, with $s = 10$. Due to the quota limitations on our core-hours on this system, we have tested the performance of `G-TMVu` over `G-TV` and `H-TMVu` over

Table 2
Normalized values of maximum volume, total volume, maximum number of messages and total number of messages of the proposed graph models G–TMV, G–TMVd and G–TMVu with respect to those of G–TV for $K \in \{128, 256, 512, 1024\}$.

		Normalized values w.r.t those of G–TV											
		Max volume			Total volume			Max message			Total message		
K	Category	G–TMV	G–TMVd	G–TMVu	G–TMV	G–TMVd	G–TMVu	G–TMV	G–TMVd	G–TMVu	G–TMV	G–TMVd	G–TMVu
128	G-K128-V8000	0.69	0.74	0.73	1.10	1.07	1.00	1.05	1.03	0.99	1.22	1.14	1.04
	G-K128-V4000	0.73	0.82	0.76	1.12	1.09	0.99	1.09	1.04	1.00	1.27	1.17	1.03
	G-K128-V2000	0.78	0.85	0.81	1.10	1.08	0.99	1.13	1.06	1.02	1.29	1.16	1.05
	G-K128-V1000	0.84	0.88	0.85	1.10	1.07	0.99	1.16	1.09	1.01	1.28	1.16	1.03
	G-K128-V500	0.86	0.89	0.87	1.11	1.08	0.99	1.18	1.10	1.01	1.31	1.19	1.03
	G-K128-V250	0.89	0.90	0.88	1.12	1.09	0.99	1.24	1.12	1.01	1.35	1.21	1.03
	G-K128-V125	0.91	0.92	0.89	1.13	1.10	1.00	1.30	1.16	1.01	1.40	1.24	1.03
	G-K128-V0	0.95	0.95	0.90	1.16	1.12	1.00	1.43	1.24	1.00	1.48	1.29	1.02
256	G-K256-V8000	0.54	0.69	0.65	1.09	1.07	0.99	1.07	1.02	1.01	1.26	1.16	1.08
	G-K256-V4000	0.63	0.73	0.68	1.12	1.08	0.99	1.10	1.05	1.00	1.31	1.18	1.04
	G-K256-V2000	0.70	0.81	0.75	1.10	1.07	0.99	1.12	1.07	1.00	1.31	1.17	1.04
	G-K256-V1000	0.76	0.84	0.79	1.10	1.07	0.99	1.13	1.05	0.99	1.30	1.16	1.03
	G-K256-V500	0.82	0.86	0.85	1.11	1.08	0.99	1.21	1.10	1.00	1.32	1.18	1.03
	G-K256-V250	0.86	0.88	0.87	1.12	1.08	0.99	1.24	1.10	0.99	1.35	1.20	1.03
	G-K256-V125	0.90	0.90	0.89	1.13	1.09	0.99	1.30	1.13	0.99	1.39	1.22	1.03
	G-K256-V0	0.95	0.94	0.90	1.16	1.11	1.00	1.43	1.21	0.99	1.47	1.27	1.02
512	G-K512-8000	0.49	0.59	0.68	1.09	1.06	0.99	1.09	1.03	1.00	1.27	1.13	1.06
	G-K512-V4000	0.52	0.60	0.67	1.11	1.07	0.99	1.10	1.04	0.99	1.30	1.17	1.08
	G-K512-V2000	0.60	0.70	0.73	1.11	1.08	0.99	1.13	1.07	0.99	1.35	1.21	1.06
	G-K512-V1000	0.68	0.74	0.78	1.11	1.08	0.99	1.15	1.08	1.00	1.32	1.21	1.04
	G-K512-V500	0.77	0.80	0.84	1.12	1.09	0.99	1.23	1.12	1.01	1.33	1.21	1.04
	G-K512-V250	0.85	0.86	0.88	1.13	1.10	0.99	1.28	1.15	1.00	1.38	1.24	1.03
	G-K512-V125	0.89	0.89	0.89	1.14	1.11	0.99	1.34	1.19	0.99	1.40	1.26	1.03
	G-K512-V0	0.94	0.93	0.90	1.17	1.13	0.99	1.43	1.27	0.99	1.46	1.31	1.02
1024	G-K1024-V8000	0.39	0.41	0.61	1.07	1.04	0.98	1.08	1.02	1.01	1.20	1.10	1.03
	G-K1024-V4000	0.44	0.48	0.63	1.10	1.07	0.99	1.08	1.03	0.99	1.30	1.19	1.06
	G-K1024-V2000	0.52	0.57	0.70	1.11	1.09	1.00	1.10	1.06	0.99	1.32	1.23	1.04
	G-K1024-V1000	0.67	0.69	0.80	1.11	1.09	1.00	1.14	1.09	1.00	1.31	1.23	1.04
	G-K1024-V500	0.79	0.79	0.86	1.12	1.10	1.00	1.15	1.09	1.00	1.32	1.24	1.04
	G-K1024-V250	0.86	0.86	0.89	1.14	1.11	1.00	1.28	1.19	1.00	1.36	1.26	1.03
	G-K1024-V125	0.90	0.89	0.90	1.15	1.12	1.00	1.36	1.23	1.00	1.39	1.29	1.02
	G-K1024-V0	0.95	0.94	0.91	1.17	1.15	1.00	1.42	1.31	1.00	1.42	1.33	1.02

H–TV for 30 test matrices. Whenever we use the phrase "parallel SpMM with G–TMVu/H–TMVu/G–TV/H–TV", we refer to the parallel SpMM execution when the matrices in SpMM are partitioned with G–TMVu/H–TMVu/G–TV/H–TV.

Table 5 presents the parallel SpMM runtime results with G–TMVu and H–TMVu normalized with respect to those of G–TV and H–TV, respectively. The 30 test matrices are a subset of dataset ds–general with 964 matrices whose partitioning results are given in the previous section.

Observe that the improvements obtained by G–TMVu and H–TMVu in maximum volume (Tables 2 and 3) are reflected upon the parallel SpMM runtimes. In most instances, these two schemes lead to a lower runtime compared to the standard models. On the average, parallel SpMM with G–TMVu runs 4%, 11% and 14% faster than parallel SpMM with G–TV, whereas parallel SpMM with H–TMVu runs 14%, 13% and 22% faster than parallel SpMM with H–TV for 128, 256 and 512 processors, respectively. There are two important observations that can be inferred from these results. (i) The improvements in parallel SpMM runtimes attained both by G–TMVu and H–TMVu increase with increasing number of processors. This can be attributed to the increased importance of communication costs with increasing number of processors. (ii) H–TMVu attains higher improvements in parallel SpMM runtime compared to G–TMVu. This conforms with the experimental finding that H–TMVu attains higher improvements in maximum volume compared to G–TMVu as also seen in Tables 2 and 3.

We analyze the scalability of parallel SpMM with G–TMVu and H–TMVu in Fig. 5 on 10 matrices by respectively comparing them against those with G–TV and H–TV. The results of each matrix are grouped for graph and hypergraph model. Each bar chart in the figure belongs to a different matrix and indicates the parallel SpMM runtime obtained with the respective scheme and the number of processors. The three consecutive bars for each scheme and each matrix denote the respective parallel SpMM runtimes obtained on 128, 256 and 512 processors. These matrices are chosen in such a way that they illustrate and summarize different scalability characteristics of both schemes. For matrices that already scale well with G–TV and H–TV such as 144, 598a, bauru5727 and m14b, G–TMVu and H–TMVu almost always lead to lower SpMM runtimes for all K values and improve the scalability. For matrices such as bcsstk25, juba40k and pattern1, which have an elbow while moving from 256 to 512 processors, communication costs become a bottleneck and hinder scalability. Addressing the right bottleneck for these matrices via G–TMVu and H–TMVu pays off with improved scalability by the decreased

Table 3
Normalized values of maximum volume, total volume, maximum number of messages and total number of messages of the proposed hypergraph models H−TMV, H−TMVd and H−TMVu with respect to those of H−TV for $K \in \{128, 256, 512, 1024\}$.

		Normalized values w.r.t those of H−TV											
		Max volume			Total volume			Max message			Total message		
K	Category	H−TMV	H−TMVd	H−TMVu	H−TMV	H−TMVd	H−TMVu	H−TMV	H−TMVd	H−TMVu	H−TMV	H−TMVd	H−TMVu
128	H−K128−V8000	0.50	0.66	0.72	1.15	1.10	0.97	1.07	1.03	0.99	1.58	1.24	1.02
	H−K128−V4000	0.62	0.73	0.79	1.16	1.10	0.98	1.09	1.06	1.00	1.48	1.22	1.02
	H−K128−V2000	0.71	0.80	0.82	1.15	1.11	0.98	1.13	1.07	1.00	1.49	1.23	1.03
	H−K128−V1000	0.78	0.85	0.85	1.17	1.12	0.99	1.15	1.09	0.99	1.46	1.24	1.02
	H−K128−V500	0.85	0.89	0.87	1.19	1.15	0.99	1.21	1.11	1.00	1.48	1.26	1.02
	H−K128−V250	0.89	0.91	0.87	1.21	1.16	0.99	1.25	1.13	0.99	1.49	1.27	1.02
	H−K128−V125	0.93	0.95	0.89	1.24	1.19	0.99	1.30	1.16	1.00	1.51	1.29	1.03
	H−K128−V0	1.01	1.01	0.90	1.28	1.23	0.99	1.35	1.20	1.00	1.54	1.32	1.02
256	H−K256−V8000	0.38	0.53	0.65	1.16	1.12	0.96	1.06	1.04	0.98	1.84	1.41	1.04
	H−K256−V4000	0.50	0.62	0.72	1.18	1.13	0.97	1.10	1.07	0.98	1.76	1.43	1.04
	H−K256−V2000	0.61	0.69	0.75	1.18	1.15	0.98	1.12	1.08	0.99	1.61	1.38	1.04
	H−K256−V1000	0.71	0.77	0.79	1.17	1.13	0.98	1.16	1.09	0.99	1.54	1.36	1.04
	H−K256−V500	0.79	0.83	0.82	1.19	1.16	0.98	1.19	1.11	0.99	1.52	1.36	1.03
	H−K256−V250	0.84	0.86	0.84	1.21	1.17	0.98	1.24	1.14	0.98	1.51	1.36	1.03
	H−K256−V125	0.90	0.91	0.86	1.23	1.20	0.99	1.27	1.16	0.98	1.52	1.37	1.03
	H−K256−V0	1.00	1.00	0.88	1.28	1.24	0.99	1.35	1.22	0.98	1.55	1.40	1.03
512	H−K512−V8000	0.32	0.50	0.52	1.15	1.11	0.95	1.05	1.04	0.95	1.95	1.43	1.09
	H−K512−V4000	0.34	0.50	0.53	1.19	1.14	0.96	1.05	1.03	0.94	2.00	1.51	1.11
	H−K512−V2000	0.47	0.60	0.62	1.19	1.14	0.97	1.08	1.05	0.96	1.78	1.44	1.09
	H−K512−V1000	0.64	0.74	0.73	1.18	1.14	0.98	1.14	1.08	0.96	1.60	1.35	1.05
	H−K512−V500	0.73	0.79	0.76	1.19	1.15	0.98	1.17	1.09	0.97	1.56	1.34	1.04
	H−K512−V250	0.83	0.86	0.82	1.21	1.17	0.98	1.20	1.11	0.97	1.52	1.33	1.03
	H−K512−V125	0.87	0.89	0.83	1.23	1.19	0.98	1.25	1.13	0.98	1.53	1.34	1.03
	H−K512−V0	1.00	1.01	0.85	1.28	1.23	0.99	1.33	1.20	0.98	1.55	1.36	1.03
1024	H−K1024−V8000	0.22	0.34	0.37	1.14	1.11	0.92	1.02	1.01	0.92	2.12	1.61	1.19
	H−K1024−V4000	0.26	0.38	0.41	1.16	1.14	0.93	1.02	0.99	0.89	2.06	1.62	1.17
	H−K1024−V2000	0.36	0.46	0.49	1.18	1.14	0.95	1.03	0.98	0.89	1.87	1.53	1.12
	H−K1024−V1000	0.51	0.59	0.59	1.18	1.15	0.96	1.11	1.04	0.92	1.67	1.42	1.08
	H−K1024−V500	0.72	0.75	0.74	1.19	1.16	0.97	1.12	1.06	0.95	1.57	1.39	1.04
	H−K1024−V250	0.79	0.82	0.78	1.21	1.18	0.98	1.18	1.11	0.96	1.54	1.38	1.04
	H−K1024−V125	0.85	0.86	0.80	1.23	1.20	0.98	1.23	1.15	0.96	1.54	1.39	1.04
	H−K1024−V0	1.00	1.01	0.83	1.28	1.25	0.99	1.32	1.23	0.97	1.55	1.41	1.03

Table 4
Comparison of partitioning times averaged over 964 matrices.

		Actual (ms)				Normalized w.r.t. G−TV		
model	K	G−TV	G−TMV	G−TMVd	G−TMVu	G−TMV	G−TMVd	G−TMVu
graph	128	597	1307	1192	636	2.19	2.00	1.06
	256	801	2136	1897	865	2.67	2.37	1.08
	512	1143	3148	2933	1251	2.75	2.57	1.09
	1024	1073	3662	3502	1184	3.41	3.26	1.10
		Actual (ms)				Normalized w.r.t. H−TV		
model	K	H−TV	H−TMV	H−TMVd	H−TMVu	H−TMV	H−TMVd	H−TMVu
hypergraph	128	5601	5225	5124	5939	0.93	0.91	1.06
	256	6602	5973	6786	6758	0.90	1.03	1.02
	512	7720	7012	8249	7933	0.91	1.07	1.03
	1024	8932	7956	9161	9669	0.89	1.03	1.08

runtimes with increasing number of processors. For harder instances such as lhr11, although none of the two schemes scales, G−TMVu and H−TMVu are still able to reduce the parallel SpMM runtime drastically. For example, for lhr11 on 512 processors, G−TMVu and H−TMVu respectively lead to 48% and 50% better SpMM runtimes compared to G−TV and H−TV.

6.3. Comparison against UMPa

In this section, we compare our models against UMPa [32] and present the results in Table 6. Each instance reported in the table is the geometric average of the results of five partitioning runs. The comparison is performed in terms of the

Table 5
Parallel SpMM runtimes attained by G−TMVu and H−TMVu normalized with respect to parallel SpMM runtimes attained by G−TV and H−TV, respectively, for 128, 256 and 512 processors.

Matrix name	Number of rows/cols	Number of nonzeros	Problem kind	G−TMVu			H−TMVu		
				128	256	512	128	256	512
144	145K	2,149K	Undirected graph	1.08	0.78	0.78	0.81	0.89	1.01
598a	111K	1,484K	Undirected graph	1.02	0.85	0.88	0.81	0.89	0.84
bauru5727	40K	145K	Eigenvalue/model reduction	0.97	0.70	0.73	0.75	0.73	0.83
bcsstk25	40K	145K	Structural	1.16	1.16	0.84	0.97	0.91	0.63
big	13K	92K	Directed weighted graph	0.76	0.87	0.78	1.07	1.00	0.90
bips07_3078	21K	76K	Eigenvalue/model reduction	0.87	0.69	0.93	0.94	0.88	0.77
bodyy4	18K	122K	Structural	1.03	0.92	1.00	0.74	1.00	0.90
chipcool0	20K	281K	Model reduction	1.03	0.76	0.96	1.18	0.92	0.95
copter1	17K	211K	Computational fluid dynamics	0.81	0.88	1.09	1.30	1.04	0.96
ford1	19K	102K	Structural	0.97	1.00	0.71	0.70	0.76	1.05
fv1	10K	85K	2D/3D	1.57	0.90	1.00	1.00	1.11	1.00
hcircuit	106K	513K	Circuit simulation	0.96	0.93	0.80	0.64	1.01	1.02
hvdc1	25K	158K	Power network	0.86	0.82	0.73	1.00	1.03	1.03
jan99jac040	14K	73K	Economic	0.56	0.91	0.90	0.97	0.64	0.68
juba40k	40K	145K	Eigenvalue/model reduction	0.98	1.13	0.55	1.18	1.08	0.50
lhr11	11K	232K	Chemical process simulation	0.83	0.82	0.52	0.95	0.80	0.50
m14b	215K	3,358K	Undirected graph	1.00	0.85	0.92	0.95	0.90	0.70
offshore	260K	4,243K	Electromagnetics	0.98	0.93	0.80	0.92	0.87	0.91
OPF_3754	15K	142K	Power network	1.00	0.88	1.14	1.08	0.80	0.67
pattern1	19K	9,323K	Optimization	0.77	0.81	0.65	0.62	0.67	0.60
pds10	17K	150K	Optimization	0.93	1.09	0.94	1.21	0.83	0.75
pesa	12K	80K	Directed weighted graph	0.97	0.74	0.90	0.76	1.00	0.86
rail_20209	20K	139K	Model reduction	1.17	1.00	1.18	1.03	0.90	0.94
ri2010	25K	126K	Undirected weighted graph	0.95	0.83	0.85	0.83	0.96	0.83
skirt	13K	197K	Structural	1.42	0.78	1.10	0.80	1.04	1.00
std1_Jac2	22K	1,248K	Chemical process simulation	0.82	0.97	0.62	0.47	0.63	0.54
tandem_vtx	19K	253K	Structural	1.31	0.98	1.09	0.93	0.82	1.14
TSOPF_RS_b2383	38K	16,171K	Power network	0.92	1.01	1.01	0.62	0.82	0.37
xingo3012	21K	74K	Eigenvalue/model reduction	0.93	0.86	0.79	0.96	1.00	0.63
Zd_Jac3	23K	1,916K	Chemical process simulation	0.85	0.92	1.00	0.45	0.54	0.69
			Geomean	**0.96**	**0.89**	**0.86**	**0.86**	**0.87**	**0.78**
			Best	0.56	0.69	0.52	0.45	0.54	0.37
			Worst	1.57	1.16	1.18	1.30	1.11	1.14

partition quality and the partitioning time. The partition quality is measured in terms of the maximum volume in number of words and reported as the actual values for UMPa in the second column (as reported in [32]). The third and fourth columns display the maximum volume values obtained by G−TMVu and H−TMVu as normalized values with respect to those of UMPa, respectively. The last three columns display the partitioning times of the compared models as normalized values with respect to the runtime of the partitioner PaToH [21] in default setting. The rows of the table are sorted with respect to the maximum volume obtained by UMPa and are divided into two according to the matrices for which UMPa obtains a maximum volume higher/lower than 500 words.

Both G−TMVu and H−TMVu are able to obtain better quality partitions than UMPa. On the average, G−TMVu and H−TMVu obtain improvements of 5% and 11% in maximum volume compared to UMPa, respectively. For the matrices for which UMPa obtains higher maximum volume, i.e., at least 500 words, the improvements attained by G−TMVu and H−TMVu are more apparent: 19% for G−TMVu and 24% for H−TMVu. Recall that for such matrices, maximum volume is more likely to be a critical factor in determining the overall performance. The examples for such matrices are seen in classes such as “Citation” and “Clustering” (see the classes of the matrices in Table 1), for which both G−TMVu and H−TMVu perform significantly better than UMPa in terms of partition quality. These are usually hard instances that are scale-free. In the remaining classes, G−TMVu produces slightly worse quality partitions, while H−TMVu produces comparable quality partitions.

Both G−TMVu and H−TMVu are significantly faster than UMPa. The average partitioning time of UMPa is 4.40× that of PaToH, whereas the average partitioning times of G−TMVu and H−TMVu are respectively 0.30× and 1.31× that of PaToH. As a result, remarkably, G−TMVu is on the average 14.5× faster than UMPa. H−TMVu is 3.4× faster than UMPa.

6.4. Scalability analysis

We thoroughly evaluate the scalability of the multi-source level-synchronized BFS kernel executed on the five graphs in dataset ds−large. We compare our models G−TMVu and H−TMVu against 2D [12] in terms of parallel runtime of multi-source BFS and communication statistics. The runtimes reported in this section are the execution times of parallel multi-source BFS on these graphs and not the partitioning times. Whenever we use the phrase “parallel BFS with G−TMVu/H−TMVu/2D”, we refer to the parallel BFS execution when the vertices/edges of the input graph are partitioned using G−TMVu/H−TMVu/2D. We investigate the scalability performance on the multi-source BFS and not on SpMM since

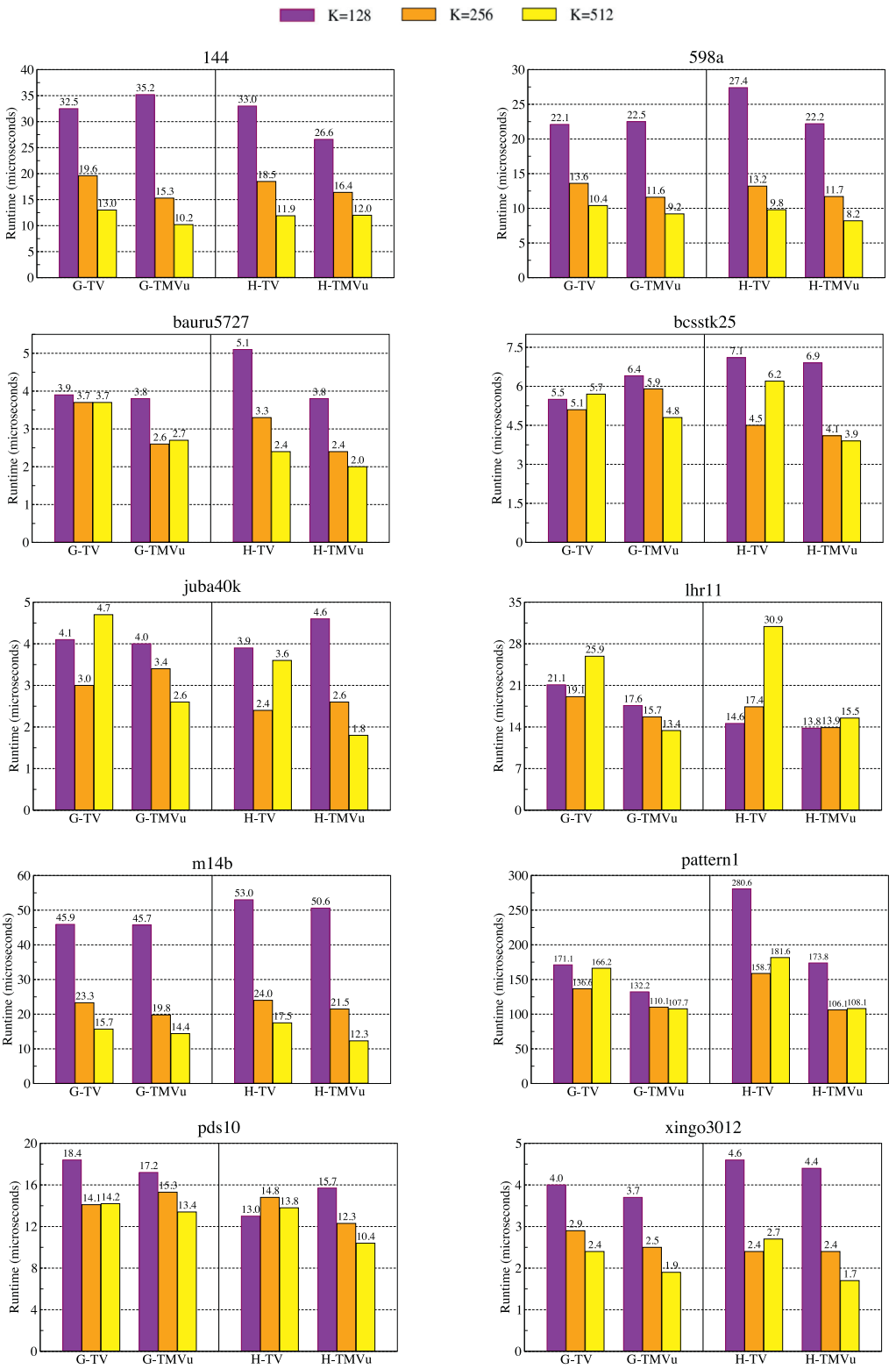

Fig. 5. Strong scaling analysis of parallel SpMM with `G-TMVu` and `H-TMVu` schemes compared to those with `G-TV` and `H-TV`, respectively.

Table 6
Comparison of G-TMVu and H-TMVu against UMPa in terms of maximum volume (number of words communicated) and partitioning time for $K = 512$ processors. The matrices are sorted according to the maximum volume values obtained by UMPa.

		Max volume			Partitioning time w.r.t. PaToH		
		Actual	Norm. w.r.t. UMPa				
name		UMPa	G-TMVu	H-TMVu	UMPa	G-TMVu	H-TMVu
eu-2005		8544	1.18	0.27	6.68	0.18	1.05
coPapersDBLP		7229	1.03	0.96	3.40	0.10	0.91
in-2004		4962	0.74	0.60	2.83	0.17	0.87
preferentialAttachment		3938	0.45	1.62	8.48	0.33	1.27
coPapersCiteseer		3927	0.83	0.78	1.56	0.07	1.06
cnr-2000		3096	0.41	0.32	6.93	0.22	1.05
caidaRouterLevel		2932	0.29	0.29	6.93	0.38	1.32
audikw_1		2655	1.04	1.04	1.57	0.09	1.03
citationCiteseer		2003	1.06	1.08	6.64	0.26	1.24
coAuthorsDBLP		1489	0.71	0.61	8.72	0.44	1.35
G_n_pin_pout		1283	0.96	1.12	7.01	0.40	1.46
auto		717	1.22	1.17	6.96	0.21	1.34
coAuthorsCiteseer		688	0.85	0.89	7.17	0.44	1.33
af_shell10		621	1.03	0.97	1.13	0.20	1.10
ldoor		582	0.98	0.98	1.61	0.14	1.07
m14b		501	1.01	0.99	6.94	0.23	1.42
Max volume of UMPa > 500	**Avg:**	–	**0.81**	**0.76**	**4.35**	**0.21**	**1.17**
	Impr:	–	**19%**	**24%**	–	**20.7x**	**3.7x**
smallworld		497	0.93	0.93	7.84	0.50	1.35
G3_circuit		406	1.02	0.99	5.40	0.45	1.41
144		402	0.89	0.86	6.64	0.25	1.48
wave		375	1.09	1.10	6.87	0.27	1.51
af_shell9		352	1.06	1.04	1.61	0.20	1.16
598a		327	1.04	1.02	6.93	0.27	1.48
rgg_n_2_20_s0		253	1.17	1.02	2.31	0.30	1.25
thermal2		248	0.98	0.96	3.12	0.39	1.31
fe_ocean		232	1.29	1.01	7.36	0.45	1.95
delaunay_n20		209	1.02	1.01	3.49	0.43	1.36
ecology1		203	1.16	0.97	4.58	0.51	1.45
ecology2		201	1.18	1.01	4.58	0.50	1.45
rgg_n_2_19_s0		172	1.17	1.01	3.51	0.31	1.26
delaunay_n19		147	1.02	1.00	4.38	0.43	1.41
rgg_n_2_18_s0		123	1.13	0.99	4.06	0.33	1.31
delaunay_n18		108	0.99	0.97	5.15	0.42	1.46
rgg_n_2_17_s0		85	1.13	1.01	5.70	0.35	1.34
delaunay_n17		77	1.01	0.97	6.05	0.42	1.47
belgium_osm		65	1.29	1.29	2.06	1.03	1.72
luxembourg_osm		17	1.21	1.10	4.83	1.14	1.90
Overall	**Avg:**	–	**0.95**	**0.89**	**4.40**	**0.30**	**1.31**
	Impr:	–	**5%**	**11%**	–	**14.5x**	**3.4x**

2D was originally proposed for the former and all three models exhibit similar behaviors in both. There are four different number of processors, $K \in \{256, 512, 1024, 2048\}$ and four different number of source vertices, $s \in \{5, 10, 20, 40\}$ in our experiments. Recall that the number of source vertices in the multi-source BFS is equivalent to the number of columns of input dense matrix X in SpMM. We investigate both strong and weak scaling performances of multi-source BFS with these three models. The experiments were performed on the Lenovo NeXtScale system.

We present the strong scaling results in Fig. 6. Each row in the figure belongs to a different graph and each column belongs to a different s value. Each plot contains three lines comparing G-TMVu, H-TMVu and 2D for a specific graph and s. The x-axis and y-axis respectively denote the number of processors and the runtime of the operations in a single level of parallel multi-source BFS in miliseconds. Both axes are in logarithmic scale.

As seen from the plots in Fig. 6, for all instances, both parallel BFS with G-TMVu and parallel BFS with H-TMVu run much faster than parallel BFS with 2D. For example, for 256, 512, 1024 and 2048 processors, parallel BFS with G-TMVu respectively runs 5.3× , 6.9× , 8.0× and 10.8× faster than 2D, for $s = 20$, on the average. Again for $s = 20$ and for the same numbers of processors, parallel BFS with H-TMVu respectively runs 6.6× , 8.4× , 10.3× and 10.3× faster than 2D. This is simply because the communication cost of parallel multi-source BFS is largely dominated by the bandwidth costs and our models aim at reducing bandwidth-related metrics total and maximum volume, whereas 2D only aims to provide an upper bound on latency-related metrics. This results in our models to achieve lower communication overhead and hence better performance. The scalability of our models becomes more apparent with increasing s. The performance gap between 2D and our models in terms of scalability turns into favor of our models with increasing s. For example, for it-2004, compared

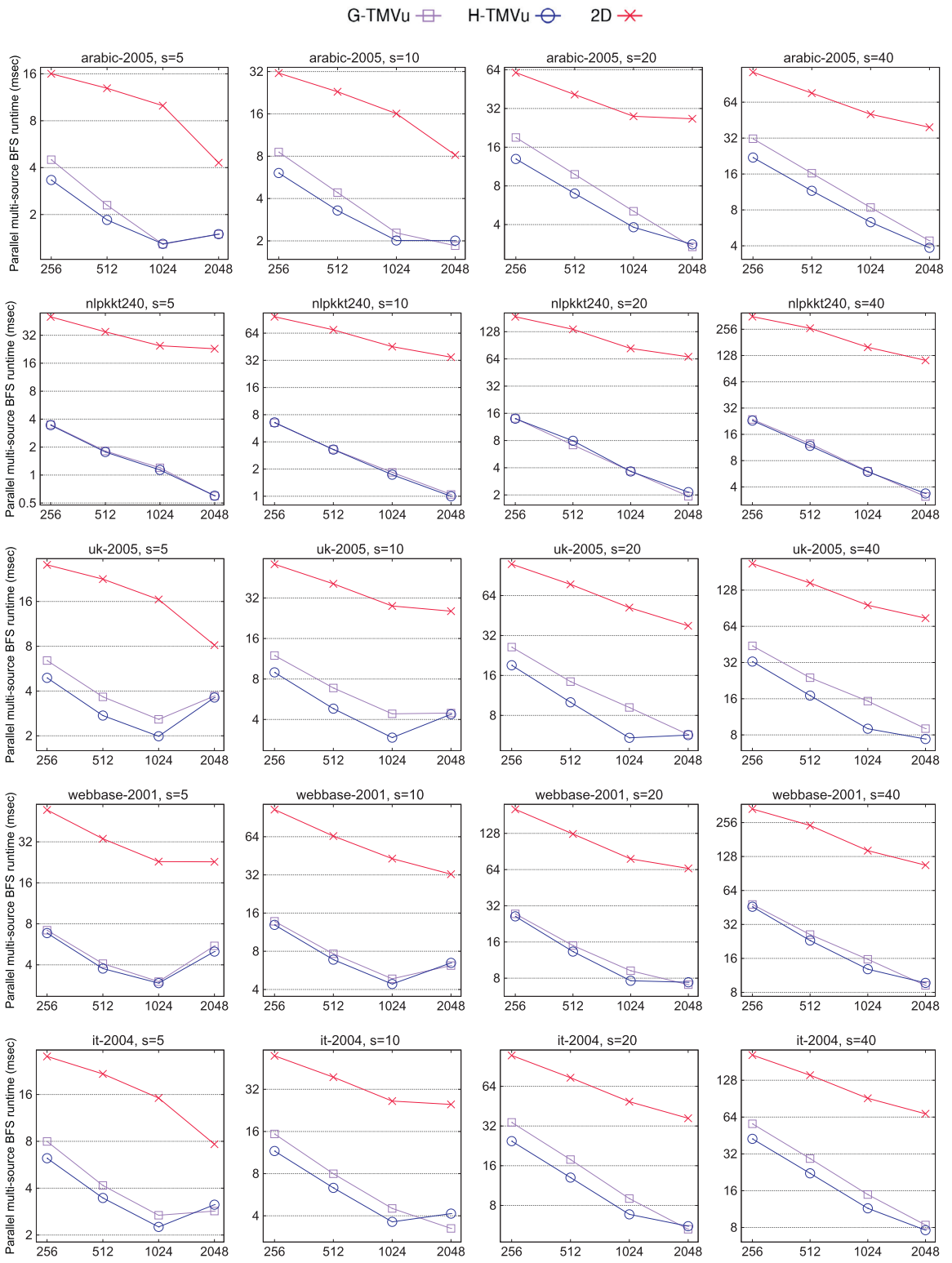

Fig. 6. Strong scaling analysis of parallel multi-source BFS with `G-TMVu`, `H-TMVu` and 2D. The *x*-axis denotes the number of processors.

Table 7
Volume and message count statistics of `it-2004` and `nlpkkt240` for $s = 5$ and $s = 40$. Note that message count statistics are the same for $s = 5$ and $s = 40$.

Volume statistics (in megabytes)

		$s = 5$						$s = 40$					
		Maximum			Average			Maximum			Average		
Graph	K	2D	G-TMVu	H-TMVu	2D	G-TMVu	H-TMVu	2D	G-TMVu	H-TMVu	2D	G-TMVu	H-TMVu
it-2004	256	37.71	0.83	0.75	37.45	0.16	0.11	301.70	6.63	6.00	299.61	1.31	0.84
	512	21.11	0.71	0.57	20.86	0.09	0.06	168.85	5.64	4.59	166.89	0.75	0.49
	1024	13.93	0.64	0.32	13.70	0.06	0.04	111.45	5.14	2.59	109.63	0.50	0.30
	2048	7.70	0.50	0.52	7.48	0.05	0.03	61.58	3.98	4.15	59.81	0.38	0.27
nlpkkt240	256	54.53	0.64	0.48	54.44	0.47	0.35	436.26	5.15	3.86	435.52	3.72	2.80
	512	33.45	0.39	0.31	33.38	0.29	0.23	267.63	3.10	2.50	267.07	2.32	1.83
	1024	19.84	0.25	0.21	19.77	0.18	0.15	158.69	2.03	1.69	158.19	1.48	1.21
	2048	11.08	0.16	0.15	11.02	0.12	0.10	88.62	1.31	1.21	88.15	0.95	0.79

Message count statistics

		Maximum			Average		
Graph	K	2D	G-TMVu	H-TMVu	2D	G-TMVu	H-TMVu
it-2004	256	30	245	240	30	172	139
	512	46	463	442	46	267	184
	1024	62	724	731	62	347	192
	2048	94	1171	1488	94	324	231
nlpkkt240	256	30	20	22	30	12	12
	512	46	23	21	46	13	13
	1024	62	22	23	62	14	14
	2048	94	21	25	94	14	14

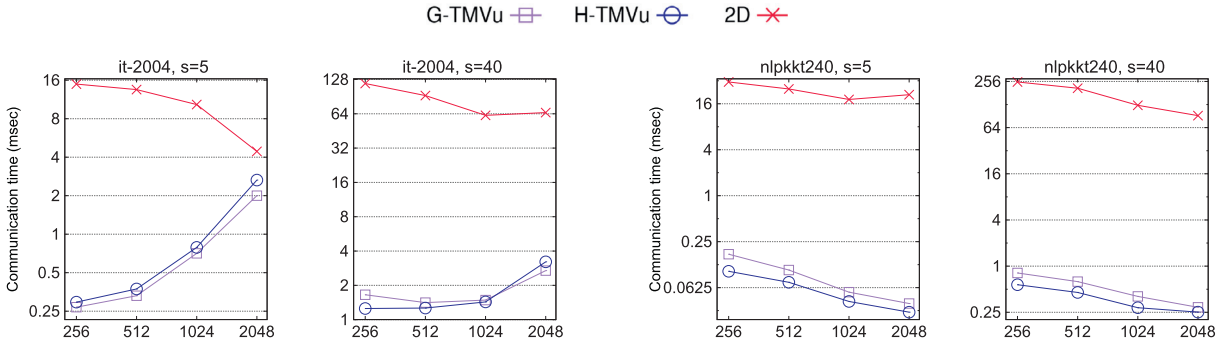

Fig. 7. Communication times in parallel multi-source BFS with `G-TMVu`, `H-TMVu` and 2D for `it-2004` and `nlpkkt240` both with $s \in \{5, 40\}$. The x-axis denotes the number of processors.

to 2D, parallel BFS with `G-TMVu` runs 3.5× faster on 256 processors and 2.7× faster on 2048 processors for $s = 5$, whereas for $s = 40$ it runs 3.6× faster on 256 processors and 8.1× faster on 2048 processors. Similar improvements are observed for parallel BFS with `H-TMVu` as well, where these two values are 4.5× and 2.4× for $s = 5$ and they are 4.8× and 9.0× for $s = 40$. The close performances of `G-TMVu` and `H-TMVu` for `nlpkkt240` are due to the regular sparsity pattern of this matrix which hides the flaw of the graph model [21,39] to a large extent.

We investigate the communication performance of parallel BFS with `G-TMVu`, `H-TMVu` and 2D. The volume and message count statistics obtained by these models are given in Table 7 for $s = 5$ and $s = 40$, and for 256, 512, 1024 and 2048 processors. Both the volume and message count statistics include maximum and average values. We focus on two graphs `it-2004` and `nlpkkt240` since `it-2004` is the largest graph in `ds-large` and for the other graphs in `ds-large` except `nlpkkt240`, we observe similar findings with those for `it-2004`. Fig. 7 illustrates variation of the communication times with varying number of processors for parallel BFS.

In all instances, both of our models obtain lower communication times than 2D. The significantly better performance of our models can be explained with the significant reductions obtained in both maximum and average volume, as seen in Table 7. For example, for 512 processors and $s = 40$, the maximum volume values obtained by 2D for 512 processors and $s = 40$ are 168.85 MB and 267.63 MB for `it-2004` and `nlpkkt240`, respectively, whereas these two values are 5.64 MB and 3.10 MB for `G-TMVu`, respectively, and 4.59 MB and 2.50 MB for `H-TMVu`, respectively. There are similar significant reductions in average volume.

For `it-2004`, it is seen from the left two plots in Fig. 7 that the performance gap between `G-TMVu`/`H-TMVu` and 2D is much higher for $s = 40$ than the gap for $s = 5$, especially with larger number processors. This is mainly because the volume-based metrics for $s = 40$ are more determinant in communication times compared to $s = 5$, as message count statistics do

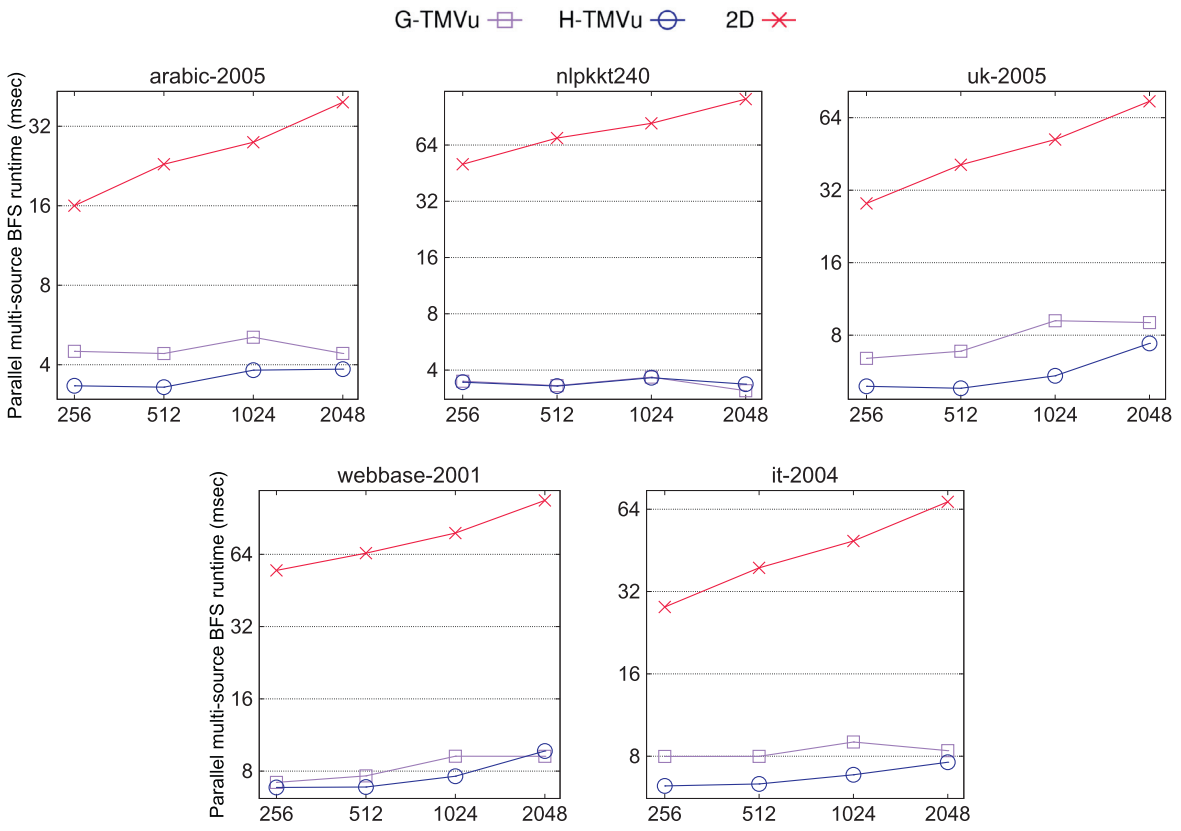

Fig. 8. Weak scaling analysis of parallel multi-source BFS with G−TMVu, H−TMVu and 2D. The *x*-axis denotes the number of processors.

not change while volume statistics increase with increasing s. For example, the average volume obtained by 2D for 1024 processors is 13.70 MB for $s = 5$ while it is 109.63 MB for $s = 40$ and the average message count is 62 regardless of s. For $s = 5$, the decrease in the performance gap between our models and 2D for larger number of processors can be explained by the increased importance of latency-related metrics in communication time and since 2D model provides an upper bound on the maximum and total message counts, it achieves a lower latency overhead compared to G−TMVu and H−TMVu.

Compared to it−2004, nlpkkt240 exhibits a more regular structure as it is obtained by PDE discretization while it−2004 is a web graph. This can be seen in Table 7 by comparing the maximum or average message counts obtained by our models. For example, for 1024 processors, the maximum message count obtained by G−TMVu for it−2004 is 724 while it is only 22 for nlpkkt240. The regular structure of nlpkkt240 is successfully exploited by our models as G−TMVu and H−TMVu always obtain lower maximum and average message counts than 2D. As opposed to it−2004, there always exists a big performance gap between our models and 2D regardless of s and number of processors since both G−TMVu and H−TMVu perform much better in terms of both bandwidth and latency costs.

The different behavior of G−TMVu and H−TMVu for it−2004 and nlpkkt240 can be explained by the varying importance of latency costs in communication times. For both of these graphs, with increasing number of processors, maximum and average volume values tend to decrease, however maximum and average message count values for it−2004 increase while they remain the same for nlpkkt240.

We present the weak scaling results in Fig. 8. To keep the computational load of each processor fixed when we double the number of processors, we double the number of source vertices while using the same input graph. In this way, when we double the number of processors, we double the total amount of computation while keeping the structure of the input graph same. Ideally, the number of edges assigned to each processor is halved when the number of processors is doubled. In other words, we use $s = 5$ at 256 processors, $s = 10$ at 512 processors, etc. Using different number of source vertices for the BFS enables us to seamlessly perform weak scaling analysis. The five plots in Fig. 8 show that our models exhibit superior weak scaling performance compared to 2D. This is mainly because the communication costs incurred by our models tend to increase less than those incurred by 2D when the number of processors is doubled. The lines that belong to our models in these plots sometimes have a negative slope when the number of processors is doubled. This behavior can be attributed to the following two reasons: (i) unstable computational load imbalances in the partitions obtained with the partitioners, leading to number of edges owned by the processors to not always halve when the number of processors is doubled, and (ii) the increased cache utilization in local computations due to the good reorderings generated by the partitioners. Nonetheless, it can be said that our models more than often exhibit consistently good weak scaling performance overall.

7. Conclusion

This work aimed to improve the performance of sparse matrix dense matrix multiplication on distributed memory systems. We addressed the high communication volume requirements of this kernel by proposing graph and hypergraph partitioning models which can minimize multiple volume-based communication cost metrics simultaneously in a single partitioning phase. Relying on a formulation that makes use of multiple constraints in recursive bipartitioning framework, we additionally proposed two practical schemes to efficiently utilize the existing partitioning tools. The experiments performed with this kernel and a level-synchronized multi-source parallel breadth-first search kernel on a large-scale high performance computing system up to 2048 processors validate the benefits of optimizing multiple volume-based metrics via our models by improving scalability.

As future work, we plan to try out different orders for bipartitionings in recursive bipartitioning. Moreover, we also consider using other partitioners to realize our models. Among them, Scotch [25] is the first to consider due to its high quality partitions in terms of load balance.

Acknowledgments

We acknowledge PRACE (Partnership for Advanced Computing In Europe) for awarding us access to resources Hazelhen (Cray XC40) based in Germany at High Performance Computing Center Stuttgart (HLRS) and SuperMUC (Lenovo NeXtScale) based in Germany at Liebniz Supercomputing Center (LRZ). Seher Acer acknowledges the support through the Scientific and Technological Research Council of Turkey (TUBITAK), under the program BIDEB-2211.

References

[1] D.P. O'Leary, The block conjugate gradient algorithm and related methods, Linear Algebra Appl. 29 (1980) 293–322. Special Volume Dedicated to Alson S. Householder.
[2] D.P. O'Leary, Parallel implementation of the block conjugate gradient algorithm, Parallel Comput. 5 (1987) 127–139. Proceedings of the International Conference on Vector and Parallel Computing-Issues in Applied Research and Development.
[3] Y. Feng, D. Owen, D. Perić, A block conjugate gradient method applied to linear systems with multiple right-hand sides, Comput. Methods Appl. Mech. Eng. 127 (1995) 203–215.
[4] A. Murli, L. D'Amore, G. Laccetti, F. Gregoretti, G. Oliva, A multi-grained distributed implementation of the parallel block conjugate gradient algorithm, Concurrency Comput. Pract. Exper. 22 (2010) 2053–2072.
[5] R.G. Grimes, J.G. Lewis, H.D. Simon, A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems, SIAM J. Matrix Anal. Appl. 15 (1994) 228–272.
[6] M. Sadkane, A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices, Numer. Math. 64 (1993) 181–193.
[7] E.J. Im, K. Yelick, Optimization of sparse matrix kernels for data mining, in: SIAM Conference on Data Mining, 2000.
[8] S. Wasserman, Social Network Analysis: Methods and Applications, 8, Cambridge University Press, 1994.
[9] U. Kang, C.E. Tsourakakis, C. Faloutsos, Pegasus: a peta-scale graph mining system implementation and observations, in: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 229–238, doi:10.1109/ICDM.2009.14.
[10] V. Agarwal, F. Petrini, D. Pasetto, D.A. Bader, Scalable graph exploration on multicore processors, in: Proceedings of the 2010ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, in: SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11, doi:10.1109/SC.2010.46.
[11] Z. Shi, B. Zhang, Fast network centrality analysis using GPUs, BMC Bioinf. 12 (2011) 149.
[12] A. Buluç, J.R. Gilbert, The combinatorial BLAS: design, implementation, and applications, Int. J. High Perform. Comput. Appl. 25 (2011) 496–509.
[13] A. Buluç, K. Madduri, Parallel breadth-first search on distributed memory systems, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 65:1–65:12, doi:10.1145/2063384.2063471.
[14] A.E. Sarıyüce, E. Saule, K. Kaya, U.V. Çatalyürek, Regularizing graph centrality computations, J. Parallel Distrib. Comput. 76 (2015) 106–119. Special Issue on Architecture and Algorithms for Irregular Applications.
[15] A. Buluç, J.R. Gilbert, Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments, SIAM J. Sci. Comput. 34 (2012) C170–C191.
[16] L.A.N. Amaral, A. Scala, M. Barthélémy, H.E. Stanley, Classes of small-world networks, Proc. Natl. Acad. Sci. 97 (2000) 11149–11152.
[17] Intel Math Kernel Library, 2015, https://software.intel.com/en-us/intel-mkl.
[18] Nvidia cuSPARSE Library, 2015, https://developer.nvidia.com/cusparse.
[19] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing '95, ACM, New York, NY, USA, 1995, doi:10.1145/224170.224228.
[20] B. Hendrickson, T.G. Kolda, Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing, SIAM J. Sci. Comput. 21 (1999) 2048–2072.
[21] U.V. Çatalyürek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (1999) 673–693.
[22] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1998) 359–392.
[23] B. Uçar, C. Aykanat, Revisiting hypergraph models for sparse matrix partitioning, SIAM Rev. 49 (2007) 595–603.
[24] B. Uçar, C. Aykanat, Partitioning sparse matrices for parallel preconditioned iterative methods, SIAM J. Sci. Comput. 29 (2007) 1683–1709.
[25] F. Pellegrini, J. Roman, Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: High-Performance Computing and Networking, in: Lecture Notes in Computer Science, 1067, Springer, Berlin, Heidelberg, 1996, pp. 493–498, doi:10.1007/3-540-61142-8_588.
[26] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar, Multilevel hypergraph partitioning: applications in VLSI domain, IEEE Trans. Very Large Scale Integr. Syst. 7 (1999) 69–79.
[27] K. Devine, E. Boman, R. Heapby, B. Hendrickson, C. Vaughan, Zoltan data management service for parallel dynamic applications, Comput. Sci. Eng. 4 (2002) 90–97.
[28] B. Uçar, C. Aykanat, Minimizing communication cost in fine-grain partitioning of sparse matrices, in: Computer and Information Sciences - ISCIS 2003, in: Lecture Notes in Computer Science, 2869, Springer, Berlin, Heidelberg, 2003, pp. 926–933, doi:10.1007/978-3-540-39737-3_115.
[29] B. Uçar, C. Aykanat, Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies, SIAM J. Sci. Comput. 25 (2004) 1837–1859.

[30] R.H. Bisseling, W. Meesen, Communication balancing in parallel sparse matrix-vector multiply, Electron. Trans. Numer. Anal. 21 (2005) 47–65.
[31] U.V. Çatalyürek, M. Deveci, K. Kaya, B. Uçar, UMPa: a multi-objective, multi-level partitioner for communication minimization, in: Graph Partitioning and Graph Clustering 2012, in: Contemporary Mathematics, 588, AMS, 2013, pp. 53–66, doi:10.1090/conm/588/11704.
[32] M. Deveci, K. Kaya, B. Uçar, U.V. Çatalyürek, Hypergraph partitioning for multiple communication cost metrics: model and methods, J. Parallel Distrib. Comput. 77 (2015) 69–83.
[33] O. Selvitopi, S. Acer, C. Aykanat, A recursive hypergraph bipartitioning framework for reducing bandwidth and latency costs simultaneously, IEEE Trans. Parallel Distrib. Syst. (2016), doi:10.1109/TPDS.2016.2577024.
[34] O. Selvitopi, C. Aykanat, Reducing latency cost in 2D sparse matrix partitioning models, Parallel Comput. 57 (2016) 1–24, doi:10.1016/j.parco.2016.04.004.
[35] U.V. Çatalyürek, C. Aykanat, A hypergraph-partitioning approach for coarse-grain decomposition, in: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, SC '01, ACM, New York, NY, USA, 2001, p. 28, doi:10.1145/582034.582062.
[36] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint hypergraph partitioning, Technical Report, 99-034, University of Minnesota, Dept. Computer Science/Army HPC Research Center, Minneapolis, MN, 1998.
[37] M.R. Garey, D.S. Johnson, L. Stockmeyer, Some simplified NP-complete problems, in: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, STOC '74, ACM, New York, NY, USA, 1974, pp. 47–63, doi:10.1145/800119.803884.
[38] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, John Wiley & Sons, Inc., New York, NY, USA, 1990.
[39] B. Hendrickson, Graph partitioning and parallel solvers: has the emperor no clothes? (extended abstract), in: Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel, IRREGULAR '98, Springer-Verlag, London, UK, 1998, pp. 218–225. http://dl.acm.org/citation.cfm?id=646012.677019.
[40] B. Hendrickson, T.G. Kolda, Graph partitioning models for parallel computing, Parallel Comput. 26 (2000) 1519–1534.
[41] V. Kumar, Introduction to Parallel Computing, 2nd, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[42] T.A. Davis, University of Florida Sparse Matrix Collection, NA Digest 92 (1994).
[43] 10th DIMACS Implementation Challenge: Graph Partitioning and Graph Clustering, 2011, http://www.cc.gatech.edu/dimacs10/.
[44] B. Uçar, C. Aykanat, A library for parallel sparse matrix-vector multiplies, Technical Report, BU-CE-0506, Bilkent University, Computer Engineering Department., 2005. Also available at http://www.cs.bilkent.edu.tr/tech-reports/2005.