Partitioning, Ordering, and Load Balancing in a Hierarchically Parallel Hybrid Linear Solver

Ichitaro Yamazaki* Xiaoye S. Li* François-Henry Rouet† Bora Uçar‡

Abstract

PDSLin is a general-purpose algebraic parallel hybrid (direct/iterative) linear solver based on the Schur complement method. The most challenging step of the solver is the computation of a preconditioner based on an approximate global Schur complement. We investigate two combinatorial problems to enhance PDSLin's performance at this step. The first is a multi-constraint partitioning problem to balance the workload while computing the preconditioner in parallel. For this, we describe and evaluate a number of graph and hypergraph partitioning algorithms to satisfy our particular objective and constraints. The second problem is to reorder the sparse right-hand side vectors to improve the data access locality during the parallel solution of a sparse triangular system with multiple right-hand sides. This is to speed up the process of eliminating the unknowns associated with the interface. We study two reordering techniques: one based on a postordering of the elimination tree and the other based on a hypergraph partitioning. To demonstrate the effect of these techniques on the performance of PDSLin, we present the numerical results of solving large-scale linear systems arising from two applications of our interest: numerical simulations of modeling accelerator cavities and of modeling fusion devices.

1 Introduction

In recent years, the Schur complement method [27] has gained popularity as a framework to develop a scalable parallel hybrid (direct/iterative) linear solver [14, 17, 29]. In this method, the original linear system $Ax = b$ is first reordered into a system of the following block structure:

$$\left(\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \dots & F_k & C \end{array}\right) \left(\begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_k \\ y \end{array}\right) = \left(\begin{array}{c} f_1 \\ f_2 \\ \vdots \\ f_k \\ g \end{array}\right), \tag{1}$$

where D_ℓ is referred to as the ℓ-th *interior subdomain*, C consists of *separators*, and E_ℓ and F_ℓ are the *interfaces* between D_ℓ and C. To compute the solution of the linear system (1), we first compute the solution vector y on the interface by solving the Schur complement system,

$$Sy = \widehat{g}, \tag{2}$$

*Lawrence Berkeley National Laboratory, Berkeley, CA94720, U.S.A. ic.yamazaki@gmail.com and xsli@lbl.gov.
†ENSEEIHT-IRIT, 2 rue Camichel, 31071, Toulouse, France. frouet@enseeiht.fr.
‡CNRS and LIP, ENS-Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. bora.ucar@ens-lyon.fr.

where the Schur complement S is defined as

$$S = C - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} E_\ell \ , \tag{3}$$

and $\widehat{g} = g - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} f_\ell$. Then, to compute the solution vector u_ℓ on the ℓ-th subdomain, we solve the ℓ-th subdomain system

$$D_\ell u_\ell = f_\ell - E_\ell y \ . \tag{4}$$

PDSLin (Parallel Domain decomposition Schur complement based Linear solver) is a solver which implements the outlined Schur complement method. It is designed to solve large-scale highly-indefinite linear systems of equations.

PDSLin first uses the parallel direct solver SuperLU_DIST [13] to factorize the mutually-independent interior subdomains D_ℓ in parallel. SuperLU_DIST may not be effective factorizing the large-scale global matrix A using large number of processors, but it is efficient factorizing D_ℓ using tens, or even hundreds of processors in some cases. To avoid the potentially large memory required to explicitly form the Schur complement S, PDSLin uses a preconditioned iterative method to solve the Schur complement system (2) as summarized below.

To provide a flexible preconditioner for solving the Schur complement system (2), PDSLin explicitly computes an approximation $\widetilde{S}$ to the global Schur complement S. Specifically, from the initial partition (1), PDSLin first extracts a *local* matrix A_ℓ associated with each subdomain D_ℓ:

$$A_\ell = \begin{pmatrix} D_\ell & \widehat{E}_\ell \\ \widehat{F}_\ell & O \end{pmatrix},$$

where $\widehat{E}_\ell$ and $\widehat{F}_\ell$ consist of the nonzero columns and rows of E_ℓ and F_ℓ, respectively. Then, the LU factors of D_ℓ are computed using SuperLU_DIST, i.e., $P_\ell D_\ell \bar{P}_\ell = L_\ell U_\ell$, where P_ℓ and $\bar{P}_\ell$ are the row and column permutation matrices, respectively. Next, the update matrix T_ℓ is computed as follows:

$$\begin{aligned} T_\ell &= \widehat{F}_\ell D_\ell^{-1} \widehat{E}_\ell & (5)\\ &= (\widehat{F}_\ell \bar{P}_\ell U_\ell^{-1})(L_\ell^{-1} P_\ell \widehat{E}_\ell) & (6)\\ &= W_\ell G_\ell \ , & (7) \end{aligned}$$

where $W_\ell = \widehat{F}_\ell \bar{P}_\ell U_\ell^{-1}$ and $G_\ell = L_\ell^{-1} P_\ell \widehat{E}_\ell$. A large amount of fill may occur in W_ℓ and G_ℓ. To reduce the memory and computational costs, PDSLin computes the approximations $\widetilde{W}_\ell$ and $\widetilde{G}_\ell$ of W and G, respectively, by discarding nonzeros with magnitudes less than a prescribed threshold. Then, as an approximate update matrix $\widetilde{T}_\ell = \widetilde{W}_\ell \widetilde{G}_\ell$ is computed, it is gathered to form an approximate Schur complement

$$\widehat{S} = C - \sum_{\ell=1}^{k} R_{F_\ell} \widetilde{T}_\ell R_{E_\ell}^T \ ,$$

where R_{E_ℓ} and R_{F_ℓ} are interpolation matrices to map the columns and rows of $\widehat{E}_\ell$ and $\widehat{F}_\ell$ to those of E_ℓ and F_ℓ, respectively. To further reduce the costs, small nonzeros are discarded from $\widehat{S}$ to form its approximation $\widetilde{S}$. Finally, the LU factors of $\widetilde{S}$ are computed using SuperLU_DIST and used as a preconditioner when solving the Schur complement system.

A salient feature of PDSLin is that it uses *two-level parallelism*: each subdomain can be processed in parallel with multiple processors, using a distributed-memory solver such as SuperLU_DIST, and multiple subdomains are processed in parallel by multiple groups of processors. This implies that the number of subdomains can be far smaller than the number of processors to be used. In practice, we limit the number of subdomains to a few tens. This is far enough to enable high degree parallelism. For example, in [29], we show the results using 4,096 processors; the number of subdomains is set to 64, which means that every subdomain is processed with 64 processors—this does not pose any scaling problem for the direct solver used within the subdomain. Hence, PDSLin has the potential to be scalable on large number of processors by bringing together much of the progress made in both direct and iterative linear solvers. A detailed description of our parallel implementation can be found in [29].

The construction of the approximate Schur complement $\widetilde{S}$ is the most challenging step of PDSLin. In this paper, we study the following two combinatorial problems to improve the performance of PDSLin at this step: computing the partition (1) with multiple constraints to improve parallel load balance of PDSLin at various stages during execution (Section 3); and reordering columns of $\widehat{E}_\ell$ and $\widehat{F}_\ell^T$ to improve data locality of the triangular solver to form G_ℓ and W_ℓ respectively (Section 4). To demonstrate the effectiveness of these techniques, in Section 5, we present experimental results of solving large-scale highly-indefinite linear systems of equations arising from numerical simulations of modeling accelerator cavities [2] and those of modeling fusion devices [1]. We then conclude in Section 6.

2 Preliminaries

In this section, we provide some basic definitions on graphs and hypergraphs and the associated partitioning problems.

2.1 Graphs and graph partitioning

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ such that each edge is a distinct pair of vertices. Usually, nonnegative weights, e.g., $w(i)$, and nonnegative costs, e.g., $c(j)$, are associated with the i-th vertex and the j-th edge, respectively. There are two types of graph partitioning problems: graph partitioning by edge separators (GPES) and graph partitioning by vertex separators (GPVS).

For a given integer $k \geq 2$, GPES asks for a partition of the vertex set $\mathcal{V}$ into k parts $\Pi_k(\mathcal{V}) = (\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$ such that the following balance constraint and objective are satisfied. First, if we let $W(\mathcal{V}_\ell) = \sum_{v_i \in \mathcal{V}_\ell} w(i)$ be the weight of the ℓ-th part, then the balance constraint is formally defined as

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \varepsilon \,, \tag{8}$$

where W_{max} and W_{avg} are the largest and average part weights, respectively, and ε is a given allowable imbalance ratio. Then, the objective is to minimize the total cost of the cut edges (or of the edge separator) and hence to minimize $cutsize(\Pi_k) = \sum_{e_j \text{is a cut edge}} c(j)$.

The *adjacency graph* of a symmetric matrix M is $\mathcal{G}(M) = (\mathcal{V}, \mathcal{E})$, where the vertex v_i corresponds to the i-th row of M, and there is an edge (i, j) for each nonzero (i, j)-th element m_{ij} of M. A

k-way GPES of the adjacency graph $\mathcal{G}(M)$ can be used to permute M into a $k \times k$ block structure

$$PMP^T = \begin{pmatrix} M_{11} & M_{12} & \cdots & M_{1k} \\ M_{21} & M_{22} & \cdots & M_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k1} & M_{k2} & \cdots & M_{kk} \end{pmatrix}, \tag{9}$$

where the permutation matrix P permutes the rows associated with the vertices in $\mathcal{V}_i$ before those associated with $\mathcal{V}_j$ for $1 \leq i < j \leq k$. With unit vertex weights and edge costs, we can balance the sizes of the diagonal blocks and minimize the number of nonzeros in the off-diagonal blocks.

For a given integer $k \geq 2$, GPVS asks for a partition of the vertex set $\mathcal{V}$ into $k+1$ parts $\Pi_k(\mathcal{V}) = (\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k; \mathcal{V}_S)$ such that there is no edge between $\mathcal{V}_i$ and $\mathcal{V}_j$ for $1 \leq i < j \leq k$. The part $\mathcal{V}_S$ is called the vertex separator. The objective of GPVS is to minimize the size (or weight) of $\mathcal{V}_S$ and to balance the weights of the k parts $(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$.

A k-way GPVS of $\mathcal{G}(M)$ can be used to permute M into a $(k+1) \times (k+1)$ doubly-bordered block diagonal form

$$PMP^T = \begin{pmatrix} M_{11} & & & & M_{1S} \\ & M_{22} & & & M_{2S} \\ & & \ddots & & \vdots \\ & & & M_{kk} & M_{kS} \\ M_{S1} & M_{S2} & \cdots & M_{Sk} & M_{SS} \end{pmatrix}, \tag{10}$$

where the permutation matrix P permutes the rows of M corresponding to the vertices in $\mathcal{V}_i$ before those corresponding to the vertices in $\mathcal{V}_j$ for $1 \leq i < j \leq k$, and permutes those corresponding to the separator vertices to the end. Using unit vertex weights, GPVS formulation minimizes the separator size and balances the sizes of the diagonal blocks.

The GPES and GPVS problems are NP-complete for $k \geq 2$ [8]. There are a number of publicly-available software packages implementing efficient heuristics (see, e.g., MeTiS [20] and Scotch [25]).

2.2 Hypergraphs and hypergraph partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is a generalization of a graph, where every hyperedge (or a net) in $\mathcal{N}$ is a subset of vertices. In a k-way partition $\Pi_k(\mathcal{V}) = \{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$ of the vertex set $\mathcal{V}$, a net is said to *connect* a part if it has at least one vertex in that part. The *connectivity set* $\Lambda(j)$ of the j-th net n_j is the set of parts connected by n_j, while the *connectivity* $\lambda(j)$ of n_j is the number of parts connected by n_j, i.e., $\lambda(j) = |\Lambda(j)|$, where $|\cdot|$ denotes the cardinality of a set. The net n_j is said to be a *cut net* if $\lambda(j) > 1$.

The solution of a hypergraph partitioning problem must satisfy the same balancing constraint (8) defined for the graph partitioning problem. Due to the generalization of a hypergraph, where a net can be connected to more than two parts, the objective of a partitioning problem is to minimize a cutsize metric for which there are three standard definitions (see [11, 19]):

- `connectivity-1 (con1)` metric:

$$cutsize(\Pi_k) = \sum_{n_j \in \mathcal{N}} (\lambda(j) - 1) . \tag{11}$$

- `cut-net (cnet)` metric:

$$cutsize(\Pi_k) = \sum_{n_j \in \mathcal{N},\, \lambda(j)>1} 1 \; . \tag{12}$$

- `sum-of-external-degree (soed)` metric:

$$cutsize(\Pi_k) = \sum_{n_j \in \mathcal{N},\, \lambda(j)>1} \lambda(j) \; . \tag{13}$$

When non-unit costs are associated with the nets, the cost of a net appears as a multiplicative factor in the cutsize definitions. Each of the above metrics has been used in various application domains (for some see [4, Section 2.2]). In Section 3.3.2, we examine all of these three cutsize metrics for our partitioning problem.

In this paper, we use the column-net and the row-net hypergraph models [5, 10] of sparse matrices. The column-net hypergraph model $\mathcal{H}_{\mathcal{C}} = (\mathcal{R}, \mathcal{C})$ of an $m \times n$ sparse matrix M has m vertices and n nets. Each vertex in $\mathcal{R}$ and each net in $\mathcal{C}$ correspond to a row and a column of M, respectively. Furthermore, for a vertex r_i and net c_j, $r_i \in c_j$ if and only if $m_{ij} \neq 0$. A k-way partitioning of the column-net model can be used to permute the matrix M into a singly-bordered form

$$P_r M P_c^T = \begin{pmatrix} M_1 & & & C_1 \\ & M_2 & & C_2 \\ & & \ddots & \vdots \\ & & M_k & C_k \end{pmatrix}, \tag{14}$$

where the permutation matrices P_r and P_c are defined as follows. The matrix P_r permutes the rows of M such that the rows corresponding to the vertices in $\mathcal{V}_i$ come before those in $\mathcal{V}_j$ for $1 \leq i < j \leq k$. The matrix P_c permutes the columns corresponding to the nets that connect only $\mathcal{V}_i$ before those that connect only $\mathcal{V}_j$ for $1 \leq i < j \leq k$, and permutes the columns corresponding to the cut nets to the end. A similar partition where the interface is along the rows can be obtained using a row-net model of M, which is the column-net hypergraph model of M^T.

The hypergraph partitioning problem is NP-hard [23]. There are a number of publicly-available software packages implementing efficient hypergraph partitioning heuristics (see, e.g., Zoltan [6], PaToH [11], and Mondriaan [28]).

3 Partitioning problem and algorithms

The initial partition (1) has a significant impact on the performance of PDSLin. Our goal is to develop a partitioning algorithm which improves the parallel load balance of PDSLin and reduces its solution time. This is a very difficult problem. The main challenge is to formulate an objective function and a set of constraints that accurately measure the solution time of PDSLin. In this section, we describe a partitioning problem whose constraints and objective function are related to the performance of PDSLin. We outline, in Section 3.3, two classes of partitioning algorithms; one based on graph partitioning methods and the other based on hypergraph partitioning methods. These algorithms are designed to globally satisfy our objective function and balancing constraints.

We set some notations for this section. Recall that a k-way *vertex separator* $\mathcal{V}_S$ of $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a subset of $\mathcal{V}$ such that the removal of $\mathcal{V}_S$ divides $\mathcal{G}$ into k disconnected parts $\mathcal{G}_1(\mathcal{V}_1, \mathcal{E}_1), \mathcal{G}_2(\mathcal{V}_2, \mathcal{E}_2), \ldots,$

$\mathcal{G}_k(\mathcal{V}_k, \mathcal{E}_k)$. Let $\mathcal{V}_S^{(\ell)}$ and $\mathcal{E}_S^{(\ell)}$ respectively denote the subset of $\mathcal{V}_S$ connected to the vertices in $\mathcal{V}_\ell$, and the subset of edges $\mathcal{E}$ connecting $\mathcal{V}_S$ to $\mathcal{V}_\ell$. We assume that our coefficient matrix A has a symmetric sparsity pattern. If A has an unsymmetric sparsity pattern, then we work with the symmetrized matrix $|A^T| + |A|$.

3.1 Objective

Our primary objective is to minimize the size of the vertex separator. This is important since the number of iterations required to solve the Schur complement system (2) often increases as the size of S increases. Furthermore, minimizing the size of S is likely to reduce both the number of edges $\mathcal{E}_S^{(\ell)}$ and the number of vertices $\mathcal{V}_S^{(\ell)}$ for each ℓ, which respectively correspond to the number of nonzeros and columns in the interfaces $\widehat{E}_\ell$ (or $\widehat{F}_\ell^T$). Hence, the computation of a smaller $\widetilde{S}$ (approximate S) typically requires a smaller computational and memory costs.

3.2 Constraints

PDSLin relies on SuperLU_DIST to obtain a good *intra-processor* load balance among the processors assigned to the same subdomain [29]. Therefore, our focus here is the *inter-processor* load balance to compute $\widetilde{S}$ among the processors assigned to different subdomains.

The runtime of PDSLin is often dominated by the computation of the approximate Schur complement $\widetilde{S}$. Hence, our partitioning algorithm should balance the computational costs associated with different subdomains to compute $\widehat{S}$. Below, we list some constraints for balancing the costs of computing $\widetilde{S}$:

- *subdomain constraints*: The costs of the LU factorizations of the diagonal blocks D_ℓ should be balanced. For this, our partitioning algorithms try to balance the dimension of D_ℓ, i.e., $|\mathcal{V}_\ell|$, and/or the number of nonzeros in D_ℓ, i.e., $|\mathcal{E}_\ell|$. Even though the exact cost of an LU factorization is well understood (see [16, 24], for example), it is difficult to assign weights to a vertex, which reflect how much cost the corresponding row/column will introduce to the LU factorization. Furthermore, these costs depend on the permutation which is used to preserve the sparsity of LU factors, and on the pivoting which is used to enhance the numerical stability of factorization. Neither of them can be determined until the partition is computed.

- *interface constraints*: The solution of the sparse triangular systems (6) to compute G_ℓ and W_ℓ must be balanced. This requires not only the balanced subdomains but also the balanced interfaces. Specifically, our algorithms try to balance the numbers of columns of $\widehat{E}_\ell$, i.e., $|\mathcal{V}_S^{(\ell)}|$, or they try to balance the numbers of nonzeros in $\widehat{E}_\ell$, i.e., $|\mathcal{E}_S^{(\ell)}|$. Balancing these interface constraints also helps to balance the cost of sparse matrix-matrix multiplication (7), which can become expensive when many subdomains are generated.

3.3 Partitioning algorithms

A parallel nested graph dissection (NGD) algorithm is a standard graph partitioning algorithm used for matrix factorization. In NGD, the graph is recursively bisected using a GPVS algorithm until a desired number k of parts is obtained. This can be visualized as a binary tree, where the leaf nodes represent the k parts, and the internal nodes represent the separators at each level of bisection. By

aggregating the separator vertices into $\mathcal{V}_S$, one can permute the matrix into the doubly-bordered block diagonal form (10). Typically, the subdomain constraint of balancing the subdomain sizes is locally enforced at each bisection. However, since the bisections are performed independently from each other at each branch of the tree, the imbalance of the subdomains in the global partition may grow as more subdomains are extracted. Although this method often effectively addresses our objective of reducing the separator size, it does not address most of our constraints stated above. We use the performance of this standard algorithm implemented in software package like PT-SCOTCH [25] or ParMETIS [21] as the baseline algorithm, with which the performance of our partitioning algorithms is compared.

During our search for a viable partitioning algorithm, we investigated a large number of algorithms. Some of them turned out to be inferior to the others and some looked promising. We believe that documenting our efforts, even if some were ineffectual, is useful. This is especially important for the hybrid solver developers since all the hybrid solvers face equally complicated set of constraints and objective(s). Some of the successful approaches that we catalog here may work well for other solvers, or may be adapted for those solvers' requirements. Some of the unsuccessful approaches may help other developers invest their time in more potentially fruitful alternatives.

3.3.1 Graph partitioning methods

As discussed above, the NGD algorithm often obtains small separators but fails to meet our constraints. Our first attempt was to take the output of a k-way partition from NGD and refine it to balance our constraints by moving vertices locally. For this, we first merged the separator vertices and their neighboring vertices into a wide separator (borrowing the term from [7, 18]). Then, we used an algorithm similar to those discussed below to extract a new vertex separator from the wide separator while trying to satisfy the balancing constraints. This process can be repeated to refine the partitions further. Unfortunately, the initial partition from NGD has already minimized the separator size locally, and this local refinement cannot improve the balancing constraints without significantly increasing the vertex separator size. Our experimental results have shown that this approach increases the separator size significantly while obtaining only small improvements in the balancing constraints. As a result, the runtime of PDSLin increased.

More successful approaches take a k-way GPES $\Pi_k(\mathcal{V}) = (\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$ as the initial partitioning. A reason for their success may be that the partition from GPES is not locally optimized with respect to the separator size or constraints, and local refinement can improve both the objective function and the constraints. Specifically, this approach first defines a wide separator $\mathcal{V}_T$, which consists of all the end points of the edges in the edge separator of GPES. Then, it extracts the vertex separator $\mathcal{V}_S$ from $\mathcal{V}_T$. Let $\mathcal{G}_T = (\mathcal{V}_T, \mathcal{E}_T)$ where $\mathcal{E}_T = \mathcal{E} \cap (\mathcal{V}_T \times \mathcal{V}_T) \setminus \left(\bigcup_{\ell=1}^{k} \mathcal{V}_\ell \times \mathcal{V}_\ell \right)$. In other words, let $\mathcal{G}_T$ be a subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defined on the wide separator $\mathcal{V}_T$ and the edges between two wide separator vertices iff those two vertices are adjacent and they were in two different parts of the given partition $\Pi_k(\mathcal{V})$. In order for $\mathcal{V}_S$ to be a vertex separator with fewest vertices, all the edges in $\mathcal{G}_T$ should have at least one end point in $\mathcal{V}_S$, and $\mathcal{V}_S$ should have the minimum cardinality. In other words, our wide-to-narrow separator refinement problem is a variant of the minimum vertex cover (VC) problem for the induced graph $\mathcal{G}_T$; this is NP-hard for $k > 2$ [15, p.46].

There exists a $\frac{1}{2}$-approximate VC heuristic algorithm [12, Section 35.1]. The algorithm is an iterative one. At each iteration, it adds two end points of an arbitrary edge of $\mathcal{G}_T$ into the cover, and deletes all edges incident on these two end points. The algorithm continues until there is no edge

remaining in $\mathcal{G}_T$. To satisfy our particular objective and multi-constraints in the final partition, we have modified this $\frac{1}{2}$-approximate algorithm such that an edge satisfying some criteria is chosen at each iteration. For instance, to reduce the separator size and subdomain imbalance at the same time, we pick an end point with the largest degree from the largest subdomain, then the other end point is picked from the largest subdomain among the neighboring points of the first point. Our experiments have shown that the quality of the partition can be improved when a vertex rather than an edge is chosen at each iteration. This vertex-based VC algorithm does not have any approximation guarantee with respect to the size of $\mathcal{V}_S$, but in our experiments, it is shown to improve the performance of PDSLin over the edge-based VC algorithm. Below, we describe this algorithm in detail.

The input graph to Algorithm 1 is the edge separator induced subgraph $\mathcal{G}_T = (\mathcal{V}_T, \mathcal{E}_T)$. We first set $\mathcal{V}_S$ to be an empty set, and then move a vertex from $\mathcal{V}_T$ to $\mathcal{V}_S$ at each step. The following two-step approach is used to select the vertex to be moved. In the first step, we choose the part $\mathcal{V}_\ell$ from which the least number of vertices has been moved to $\mathcal{V}_S$. The motivation for this is to maintain a balance among the parts (already obtained in the initial GPES). This heuristic may also help to balance $|\mathcal{V}_S^{(\ell)}|$ since the number of vertices moved from $\mathcal{V}_\ell$ to $\mathcal{V}_S$ is a lower-bound on $|\mathcal{V}_S^{(\ell)}|$. Then, at the second step, to minimize the separator size, we use a greedy algorithm that chooses the vertex with the largest degree from $\mathcal{V}_\ell \cap \mathcal{V}_T$. The pseudo-code of the VC algorithm with this `min-dim(S)` heuristic is shown below. To tie-break at step 5.1), our implementation chooses the partition with the smallest index ℓ. Hence, with this particular heuristic, this algorithm picks the vertex with the largest degree from each partition in a round-robin fashion.

Algorithm 1 Vertex-based VC algorithm

1: let $\mathcal{V}_T^{(\ell)} = \mathcal{V}_T \cap \mathcal{V}_\ell$ and $\widehat{\mathcal{V}}_S^{(\ell)} \leftarrow \{\}$ // $\widehat{\mathcal{V}}_S^{(\ell)}$ is the set of vertices in $\mathcal{V}_S$ coming from $\mathcal{V}_\ell$
2: compute $deg(v_i)$ for each $v_i \in \mathcal{V}_T$ // the degree of v_i in $\mathcal{G}_T(\mathcal{V}_T, \mathcal{E}_T)$
3: **while** $\mathcal{E}_T \neq \{\}$ **do**
4: Choose a vertex v_i from $\mathcal{V}_T$ in two steps:
5: 1) choose $\mathcal{V}_T^{(\ell)}$ with the smallest $score(\ell) = |\widehat{\mathcal{V}}_S^{(\ell)}|$
6: 2) choose the vertex $v_i \in \mathcal{V}_T^{(\ell)}$ with the largest $deg(v_i)$
7: $\widehat{\mathcal{V}}_S^{(\ell)} \leftarrow \widehat{\mathcal{V}}_S^{(\ell)} \cup \{v_i\}$
8: $\mathcal{V}_T^{(\ell)} \leftarrow \mathcal{V}_T^{(\ell)} \setminus \{v_i\}$
9: Remove every edge incident on v_i from $\mathcal{E}_T$
10: Update $deg(v_j)$ for all the neighbors v_j of v_i
11: **end while**
12: Return $\mathcal{V}_S = \cup_\ell \widehat{\mathcal{V}}_S^{(\ell)}$

In Section 5, we present numerical results to show that this vertex-based VC algorithm improved the subdomain balance from that obtained by the NGD algorithm. However, this algorithm computed the separator whose size was about three times as large as that from the NGD algorithm, and it did not significantly improve the interface balance. To improve the interface balance, instead of the `min-dim(S)` heuristic, we have tried moving the vertex which obtains the best balance of $|\mathcal{V}_S^{(\ell)}|$ for the current vertex separator. Unfortunately, this `bal-ncl(E)` heuristic further increased the separator size and worsened the interface balance, especially in terms of number of columns. Numerical results of these two heuristics `min-dim(S)` and `bal-ncl(E)` are presented in Section 5.1.

Through these experiments, we found that post-processing to locally improve the balancing constraints of a given k-way partition from GPVS or GPES can significantly increase the separator size, and as a result, can increase the solution time of PDSLin.

3.3.2 Hypergraph partitioning methods

Hypergraph-based algorithms have been proposed to reorder a matrix A into the doubly-bordered form (1). These approaches first use a structural factorization of A and then use a hypergraph partitioning method. For example, the approach discussed in [9] uses the structural factorization

$$\mathrm{str}(A) \equiv \mathrm{str}(M^T M), \tag{15}$$

where $\mathrm{str}(A)$ represents the nonzero structure of a matrix A. Once this factorization is obtained, the column-net hypergraph $\mathcal{H}_{\mathcal{C}}(M)$ of M is used to obtain a singly-bordered form (14) of M. Subsequently, we have

$$\mathrm{str}(P_c A P_c^T) \equiv \mathrm{str}\begin{pmatrix} M_1^T M_1 & & & & M_1^T C_1 \\ & M_2^T M_2 & & & M_2^T C_2 \\ & & \ddots & & \vdots \\ & & & M_k^T M_k & M_k^T C_k \\ C_1^T M_1 & C_2^T M_2 & \dots & C_k^T M_k & \sum_{\ell=1}^{k} C_\ell^T C_\ell \end{pmatrix}. \tag{16}$$

When we assign unit vertex weights and unit net costs, and use the cut-net metric (12), a k-way column-net hypergraph partitioning of $\mathcal{H}_{\mathcal{C}}(M)$ minimizes the separator size in $P_c A P_c^T$ and balances the number of rows in the blocks of $P_r M P_c$. Unfortunately, this constraint does not satisfy any of our balancing constraints, and it has been shown experimentally that this approach can yield drastic imbalance in the diagonal block sizes of $P_c A P_c^T$ [22]. In [9], a partitioning method that balances the number of columns in the diagonal blocks of $P_r M P_c$ was proposed. This would balance the subdomain sizes in $P_c A P_c^T$. However, its implementation is not publicly available, and it does not balance the number of nonzeros in the diagonal blocks nor addresses our interface constraints.

To satisfy our specific constraints and objective, we propose a recursive hypergraph bisection (RHB) method. This is based on the partitioning of the column-net hypergraph $\mathcal{H}_{\mathcal{C}}(M)$ to permute the matrix $M^T M$ into the doubly-bordered form (16). As described above, we have multiple balance constraints (subdomains constraints and interface constraints); furthermore, these constraints cannot be assessed by looking at a set of a priori given vertex weights (they are said to be *complex* [22]). The essence of the method is to use, at each bisection, the information from the previous bisection steps to dynamically assign vertex weights which approximate the partitioning constraints in Section 3.2. Since we do not have any information at the first-level bisection, a unit weight is assigned to each vertex. Then the subsequent recursive bisection steps use the partial (or coarse) partition information to set weights and constraints and use multi-constraint bisection routines. This way, at each bisection step, the two parts will approximately satisfy a balance constraint, as the real balance can only be determined after the bisection. We illustrate this framework in Algorithm 2.

Weights have to be assigned to every vertex. For example, we found the following two weights to be the most effective among many weighting schemes we have tried:

Algorithm 2 $RHB(A, R, C, K, low, up)$

Input: A: a sparse matrix. R: row indices. C: column indices. K: number of parts. low, up: id of the lowest and highest numbered parts.

Output: *partition*: partition information for the rows

1: Form the model of the matrix $A(R, C)$
2: **if** This is not the first bisection step **then**
3: Use previous bisection information to set up the constraints
4: **end if**
5: Partition into two $\langle R_1, R_2 \rangle \leftarrow$BisectRows$(A(R, C))$ // with standard tools
6: Set $partition(R_1) \leftarrow low$ and set $partition(R_2) \leftarrow up$
7: Create the two column sets, either use net splitting or net discarding, giving C_1 and C_2
8: $RHB(A, R_1, C_1, K/2, low, (low + up - 1)/2)$ // recursive bisection
9: $RHB(A, R_2, C_2, K/2, (low + up - 1)/2 + 1, up)$ // recursive bisection

- $w_1(i) = \text{nnz}(M_\ell(i, :))$: An upper-bound on the number of nonzeros in D_ℓ for the current partition is given by $\sum_{v(i) \in M_\ell} w_1(i)^2$. Hence, this weight tries to balance the numbers of nonzeros in the subdomains after the next-level bisection by predicting them based on the current partition.

- $w_2(i) = \text{nnz}(M(i, :))$: This is simply the nonzeros in the corresponding row in the matrix M. An upper-bound on the total number of nonzeros introduced in the ℓ-th interface and separator by $v(i) \in M_\ell$ of the current partition is $\sum_{v(i) \in M_\ell} (w_2(i)^2 - w_1(i)^2)$. Hence, this weight is designed to balance the numbers of nonzeros in the interfaces when it is used as a complementary constraint to $w_1(i)$.

These weights can be used as either single or multiple constraints at each bisection. Notice that the weights $w_1(\cdot)$ changes at each bisection step, and RHB is different from a standard partitioning method with static vertex weights. We did not try to use $w_2(\cdot)$ as a single constraint alone because this is equivalent to the standard hypergraph partitioning methods.

In this RHB algorithm, we can use any of the three standard cut-metrics, which have the following meanings in our partitioning problem:

- `con1` of (11): This corresponds to the total number of nonzero columns in the interfaces $C_1, C_2, \ldots, C_k$ of M, and gives an upper-bound on the total number of nonzero columns in the interfaces $E_1, E_2, \ldots, E_k$ of A, since $E_i = M_i^T C_i$.

- `cnet` of (12): This corresponds to the number of columns in C_ℓ, which is the separator size of $P_c A P_c^T$.

- `soed` of (13): This sums the above two functions together, and tries to minimize both the separator size and the total number of nonzero columns in the interfaces at the same time.

The `con1` and `cnet` metrics have been described in enough details elsewhere, see e.g., [11, 22]. On the other hand, the implementation of the soed metric was not discussed either in [11] or in [22]. Therefore, we summarize our implementation here. Initially we set the cost of each net to two. Then, when a net is cut during bisections, we divide its cost by two and round up the cost to the

next smallest integer; this implies that the cost of a net is either 2 (the net is not cut) or 1 (the net is cut). Then, we proceed by following the net-splitting technique (see [11]). In this technique, for a cut net n whose vertices are in the two parts $\mathcal{V}_A$ and $\mathcal{V}_B$, a net $n_A = n \cap \mathcal{V}_A$ is put in part A and another one $n_B = n \cap \mathcal{V}_B$ is put in part B to continue with recursive bisections. When a net is cut, two new nets with cost 1 are created; therefore, the sum of the costs of the nets that represent the same net in the initial hypergraph is the connectivity λ of that net. Therefore, summing the cost of all cut nets provides the `soed` metric. An approach similar to RHB was described in [22].

In Section 5, we present numerical results to demonstrate that this RHB algorithm satisfies the balancing constraints of Section 3.2 better than the NGD algorithm, while increasing the separator size only slightly. As a result, the runtime of PDSLin was reduced using the RHB algorithm.

4 Reordering sparse right-hand sides for triangular solution

When solving the triangular systems to form G_ℓ as in (6), the sparsity of the right-hand side (RHS) columns $\widehat{E}_\ell$ is considered (a similar argument follows for forming W_ℓ with $\widehat{F}^T$). Furthermore, since there could be thousands of columns in $\widehat{E}_\ell$, these columns are partitioned into m parts, and the triangular system is simultaneously solved for the multiple columns within each part. There are several advantages of the simultaneous solution with multiple columns: 1) the symbolic algorithm needs to be invoked only once for a part, 2) the total number of messages is reduced, and 3) the data locality of accessing the L-factor may be improved. However, the disadvantage is that we need to pad zeros so that these multiple columns have the same nonzero pattern. This introduces unnecessary operations with zeros. In this section, we develop two techniques to reorder the columns of $\widehat{E}_\ell$ in order to maximize the structural similarity among the adjacent columns and hence minimize the number of padded zeros. For the rest of this section, we drop the subscript ℓ in D_ℓ, G_ℓ, $\widehat{E}_\ell$, and use ℓ to denote the ℓ-th part of the m-way partition of $\widehat{E}$. Detailed discussion of our triangular solver implementation can be found in [29].

4.1 Reordering based on elimination tree structure

The first technique is based on a postordering of the elimination tree (e-tree for short) of the matrix D; for an unsymmetric D, we use the e-tree of the symmetrized matrix $|D| + |D^T|$. Each node of e-tree corresponds to a column of the matrix and the nodes are always in a topological order, where a parent is numbered after its children. The e-tree structure gives the column dependency during the factorization of D. Moreover, it can be used to determine where the nonzero fill-ins would be generated during the triangular solution $D^{-1}b$ when b is a sparse vector. Specifically, if the i-th element $b(i)$ of b is nonzero, then the fill-ins will be generated at the positions corresponding to the nodes on the fill-in path from the i-th node to the root of the e-tree [16].

Our reordering technique works as follows. Given the e-tree of D, we permute the rows and columns of D so that the corresponding nodes of e-tree are in a postorder, that is, all the nodes in a subtree are numbered consecutively. We then permute the rows of the RHS columns $\widehat{E}$ conforming to the row permutation of D. Finally, the columns of $\widehat{E}$ are permuted such that their row indices of the first nonzeros are in ascending order. The reason this ordering may reduce the number of padded zeros is the following. Let i and j be the first nonzero indices in two adjacent columns. Since the RHS columns are sorted by the first nonzero row indices, the two nodes i and j are likely to be close together in the postordered e-tree, and the two fill-in paths from the i-th node

and the j-th node are likely to have a large degree of overlapping in the e-tree. As a result, this reordering technique is likely to increase the structural similarity among adjacent columns. Similar topological orderings have been previously used for triangular solution with multiple sparse RHSs, nullspace computations, and computing elements of the inverse of a sparse matrix [3, 26]. This simple heuristic is easy to implement and is effective in practice. However, it only considers the first nonzeros in the columns, and ignores the fill-ins generated by other nonzeros.

4.2 Reordering based on a hypergraph model

Our second reordering technique is based on a hypergraph model. To partition the RHS columns $\widehat{E}$ into m parts, we use the row-net hypergraph model of the solution vectors G, whose nonzero structure is obtained by a symbolic triangular solution. Our goal is to partition the columns of $\widehat{E}$ into m parts, where the similarity of the row structure among the corresponding columns of G in the same part is maximized.

Let B be the number of columns in each part, and consider a partition $\Pi_m = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_m\}$ of the columns of G into m parts. To simplify our discussion, we assume that the number of columns is divisible by B. Let r_i denote the set of columns of G, whose i-th row entry is nonzero, i.e., $r_i \equiv \{j : G(i,j) \neq 0\}$. Then, for a given part $\mathcal{V}_\ell$, the zeros to be padded in the i-th row is given by the formula

$$cost(r_i, \mathcal{V}_\ell) = \begin{cases} |\mathcal{V}_\ell| - |r_i \cap \mathcal{V}_\ell| & \text{if } r_i \cap \mathcal{V}_\ell \neq \emptyset \\ 0 & \text{otherwise} \end{cases} . \tag{17}$$

If the i-th row does not have any nonzero in any columns of $\mathcal{V}_\ell$, then clearly no zeros are padded in the i-th row of $\mathcal{V}_\ell$. On the other hand, if $\mathcal{V}_\ell$ has a nonzero in the i-th row, then for each column in $\mathcal{V}_\ell$ for which $G(i,j) = 0$, there will be a padded zero. Hence, this cost function counts the number of padded zeros in the i-th row of $\mathcal{V}_\ell$. The total cost of Π_m is the total number of padded zeros and given by

$$cost(\Pi_m) = \sum_{i=1}^{n_G} \sum_{\mathcal{V}_\ell \in \Lambda_i} (|\mathcal{V}_\ell| - |r_i \cap \mathcal{V}_\ell|) , \tag{18}$$

where n_G is the number of rows in G.

Since each part has B columns, the cost function (18) reduces to

$$cost(\Pi_m) = \sum_{i=1}^{n_G} (\lambda_i B - |r_i|) . \tag{19}$$

We can further manipulate the formula (19) and obtain

$$\begin{aligned} \sum_{i=1}^{n_G} (\lambda_i B - |r_i|) &= \sum_{i=1}^{n_G} \lambda_i B - nnz(G) \\ &= \sum_{i=1}^{n_G} (\lambda_i - 1) B + \sum_{i=1}^{n_G} B - nnz(G) \\ &= \sum_{i=1}^{n_G} (\lambda_i - 1) B + n_G B - nnz(G) . \end{aligned}$$

name	source	n	nnz	type
tdr190k	accelerator (Omega3P)	$1,110,242$	$43,318,292$	symmetric
tdr455k	accelerator (Omega3P)	$2,738,556$	$112,756,352$	symmetric
dds.quad	accelerator (Omega3P)	$380,698$	$15,844,364$	symmetric
dds.linear	accelerator (Omega3P)	$834,575$	$13,100,653$	symmetric
matrix211	fusion (M3D-C[1])	$801,378$	$55,758,438$	unsymmetric

Table 1: Test matrices

Hence, for a given G, the cost function (19) and the connectivity-1 metric (11) with each net having the constant cost of B differ only by the constant value $(n_G B - nnz(G)$. Therefore, one can minimize (19) by minimizing (11).

In our numerical experiments, we used PaToH to partition the first $m \times B$ columns of G enforcing each part to have B columns by setting the imbalance parameter ε of (8) to be zero. The remaining columns of G are gathered into one part at the end.

5 Numerical results

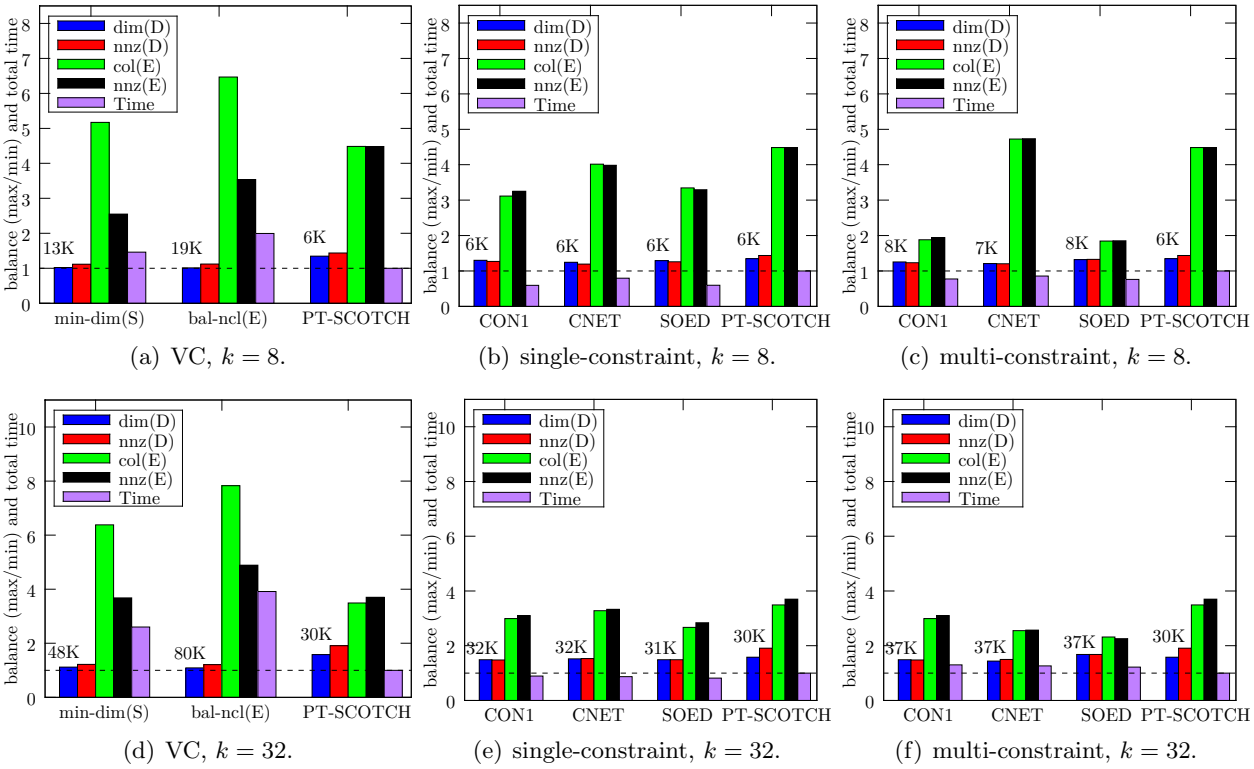
We now present numerical results of partitioning and reordering techniques described in this paper. The numerical experiments were conducted on a Cray XE6 machine at the National Energy Research Scientific Computing Center (NERSC). Each node of the machine has two 12-core AMD 2.1GHz Magny-Cours processors and 64GB of memory. The codes were written in C, and the **pgcc** compiler with **-fastsse** optimization flag were used to compile the codes. The test matrices were taken from numerical simulations of modeling particle accelerator cavities [2] and those of modeling fusion devices [1]. Some properties of the test matrices are shown in Table 1.

5.1 Partitioning with multiple constraints

We first study the performance of the partitioning techniques discussed in Section 3, namely, the vertex-cover (VC) algorithm and the recursive hypergraph bisection (RHB) algorithm. For the VC algorithm, we examined two heuristics `min-dim(S)` and `bal-ncl(E)`, while for the RHB algorithms, we considered three cut-metrics `con1`, `cnet`, and `soed`. The performance of these two algorithms is compared against that of our baseline algorithm, a nested graph dissection (NGD) algorithm implemented in PT-SCOTCH.

Figure 1 shows the results of partitioning and the solution time of PDSLin with the test matrix **tdr190k**. The number shown above each group of the bars is the separator size. We performed two sets of tests: one generating 8 subdomains (the top three plots (a)–(c)), and the other generating 32 subdomains (the bottom three plots (d)–(f)). The load balance metric is computed as W_{max}/W_{min} among all the subdomains. Since PT-SCOTCH was used as our baseline comparison, we include its data in every plot in the rightmost group of bars, even though they are the same in (a)–(c) and in (d)–(f), respectively.

In the figure, we see that the VC algorithm (subfigures (a) and (d)) improved the interior subdomain balance (shown with the label dim(D)) from those of the NGD algorithm. On the other hand, the balances of the interfaces (shown with the label col(E)) stayed about the same, or



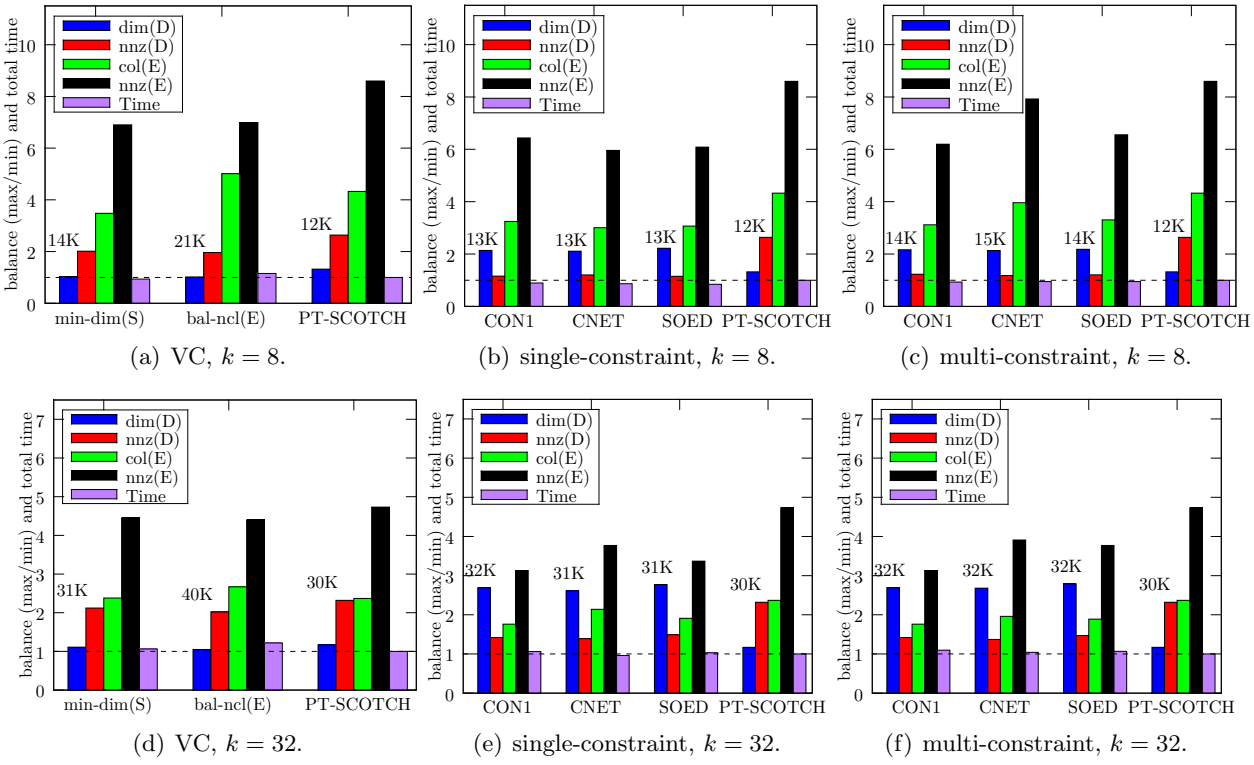
(a) VC, $k = 8$. (b) single-constraint, $k = 8$. (c) multi-constraint, $k = 8$.

(d) VC, $k = 32$. (e) single-constraint, $k = 32$. (f) multi-constraint, $k = 32$.

Figure 1: Load balancing and solution time with matrix **tdr190k**. In the legend, "dim(D)" refers to the dimension of D_ℓ, "nnz(D") refers to the number of nonzeros in D_ℓ, "col(E)" refers to the number of nonzero columns in E_ℓ, "nnz(E)" refers to the number of nonzeros in E_ℓ, and "time" refers to the total runtime of PDSLin. The number on top of each group of bars is the separator size.

worsened from those of the NGD algorithm. This is primarily because the VC algorithm obtained larger separators than the NGD algorithm, and it became more difficult to balance the interface as the separator size increases. On the other hand, with a larger number of subdomains ($k = 32$), both the single-constraint and the multi-constraint RHB algorithms improved both subdomain and interface balances with only a modest increase in the separator size, although the increase was slightly greater using the multi-constraint algorithms.

In Figure 1, the last bar of each group of bars shows the solution time of PDSLin, which is normalized to the baseline time using the NGD algorithm. First, let us compare the solution time using the VC algorithm with that using the NGD algorithm. The VC algorithm improved the subdomain balances but worsened the interface balances from those of the NGD algorithm. As a result, the VC algorithm improved the balance in the LU factorization times but worsened those in the sparse triangular solution times and in the times for sparse matrix-matrix multiplication (used to update the Schur complement). At the same time, because of the smaller subdomains with the VC algorithm, its LU factorization time was less than that of the NGD algorithm. On the other hand, the triangular solution and matrix multiplication time increased primarily due to the increase in the number of right-hand sides $\widehat{E}_\ell$ and in the amount of fill in G_ℓ. At the end, the time to compute the preconditioner increased using the VC algorithm. The main disadvantage of the



(a) VC, $k = 8$. (b) single-constraint, $k = 8$. (c) multi-constraint, $k = 8$.

(d) VC, $k = 32$. (e) single-constraint, $k = 32$. (f) multi-constraint, $k = 32$.

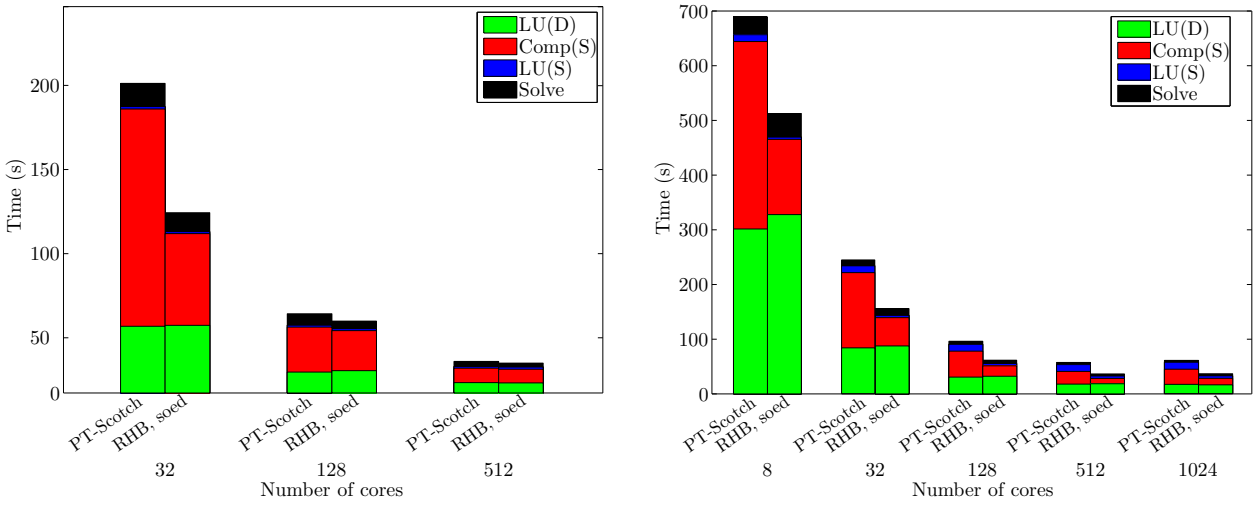
Figure 2: Load balancing and solution time with matrix **matrix211**.

VC algorithm was the significantly larger separator (2 to 3 times larger than with NGD). On the other hand, the RHB algorithms improved the load balance without increasing the maximum time required by a processor. For $k = 8$, the solution time was reduced by a factor 1.68 using the single constraint algorithm with the `soed metric` compared to the NGD (Figure 1 (b), the rightmost bar in each group); for $k = 32$, it was reduced by a factor 1.22 using the same strategy (Figure 1 (e)).

Let us summarize the above results. When the number of subdomains was moderate (e.g., $k = 32$), the RHB algorithm improved the load balance which well offset the modest increase in the separator size. As a result, the runtime of PDSLin was reduced. This is encouraging since the modest number of subdomains gives a good trade-off between the parallelism and the small Schur complement, and allows us to efficiently solve large-scale linear systems using a large number of processors. We also note that the single-constraint RHB algorithm usually gave a better result than the multi-constraint algorithm.

In Figure 2, we display the partitioning and runtime results for **matrix211**. In the figure, we see that using either single or multiple constraints, the balance in the numbers of nonzeros in the subdomains was improved from that using PT-SCOTCH, for both $k = 8$ and $k = 32$. Unfortunately, the balance in the dimensions of the subdomains, which was not directly addressed by our partitioning algorithm, got worse. Overall, the solution time was only slightly less using the single-constraint RHB. For $k = 8$, the `soed` and `cnet` metric provide similar improvements (the runtime is decreased by a factor 1.16 and 1.19 respectively); for $k = 32$, using the `soed` metric reduces the total time by a factor 1.04. The other algorithms led to comparable runtime as that of PT-SCOTCH.

We mentioned in Section 1 that PDSLin uses a two-level parallelism approach where each subdomain is processed using a parallel solver; different groups of processors are used for each subdomain, yielding intra-domain and inter-domain parallelism. We illustrate how our RHB approach behaves with respect to this feature in Figure 3 where we compare the RHB strategy (using the soed metric) with the NGD from PT-Scotch, on matrices **tdr190k** and **tdr455k**. Independent of the number of processors, the number of subdomains k is fixed to 8. One can notice that regardless of the number of processors, the RHB strategy significantly improves the runtime of PDSLin. For these two problems, the improvement is due to the update of the Schur complement which is much faster when the RHB approach is used. This can be correlated with Figure 1(c) which shows the RHB approach with the soed metric strongly improves the imbalance of the interfaces, with a similar balance of the subdomains and a minor increase in the size of the Schur complement; this better balance of the interfaces speeds up the triangular solution process used to compute the Schur complement. As exemplified by these results RHB can improve the strong scaling of PDSLin.



(a) Matrix tdr190k.

(b) Matrix tdr455k.

Figure 3: Two-level parallelism on two matrices from our experimental set. The number of subdomains is set to 8. We compare our recursive hypergraph recursive based-approach (RHB) using the `soed` metric with the nested dissection from PT-Scotch.

For our experiments, we used the serial partitioner PaToH and a serial algorithm to compute the structural decomposition (15) of A. To avoid these serial bottlenecks, we plan to investigate the usage of a parallel partitioner and develop an efficient algorithm to compute the structural decomposition (15).

5.2 Reordering sparse right-hand-side vectors

We now study the performance of the two reordering techniques, namely the postordering of the elimination tree and the ordering based on a hypergraph partitioning model. As discussed in Section 4, the objective of the ordering is to reduce the number of the padded zeros in the solution vectors and to reduce the triangular solution time. For the numerical experiments in this section, we used the parallel nested dissection algorithm of PT-SCOTCH to extract the subdomains, and

		nnz_{G_ℓ} $\times 10^6$	nnzcol_{G_ℓ} $\times 10^3$	nnzrow_{G_ℓ} $\times 10^3$	eff. dens. $\times 10^{-2}$	fill-ratio
tdr190k	min	3.55	0.60	23.0	2.20	186
	max	7.64	2.12	30.4	4.66	338
ddq.quad	min	4.99	1.02	25.2	14.5	139
	max	13.7	3.25	13.9	39.1	290
dds.linear	min	1.46	0.31	6.62	33.6	830
	max	13.0	1.95	20.7	73.1	1330
matrix211	min	0.38	1.58	12.5	1.10	20
	max	4.44	4.74	35.0	3.71	42

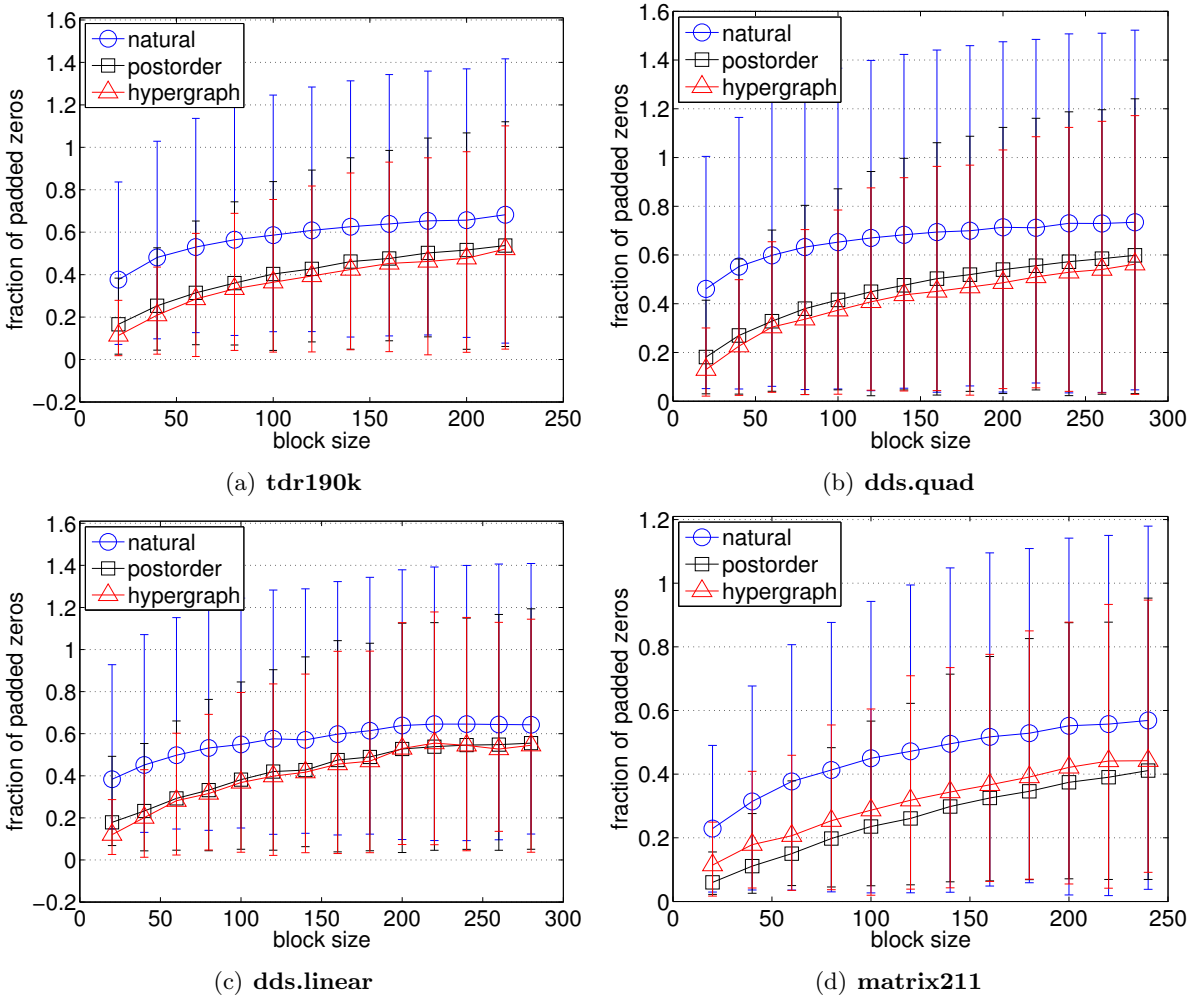
Table 2: Statistics of the eight interior subdomains and interfaces of the test matrices. "n_{D_ℓ}" is the dimension of D_ℓ; "nnz_{D_ℓ}" is the number of nonzeros in D_ℓ; "nnzcol_{G_ℓ}" is the number of columns with at least one nonzero in G_ℓ, "nnzrow_{G_ℓ}" is the number of rows with at least one nonzero in G_ℓ; and "eff. dens." is the *effective density* given by $\text{nnz}_{G_\ell}/(\text{nnzcol}_{G_\ell} \times \text{nnzrow}_{G_\ell})$, and "fill-ratio" is given by $\text{nnz}_{G_\ell}/\text{nnz}_{E_\ell}$.

a minimum degree ordering on each subdomain to preserve sparsity of the LU factors. We ran PDSLin with one-level parallel configuration, that is, the number of processors is the same as the number of subdomains. Some statistics of the partitions are shown in Table 2.

Fraction of the padded zeros. We first examine the effects of the two new ordering techniques on the fraction of the padded zeros in the supernodal blocks. Figure 4 shows the fraction of padded zeros with respect to a different partition size B. Note that it is easier to find a good ordering to reduce the number of padded zeros for a smaller B. In particular, when B is one, any ordering achieves the optimal result. We present the results with different B in order to study the performance of orderings for problems with different levels of difficulties.

For each B and each algorithm in Figure 4, the marker represents the average of the eight data points representing the eight interior subdomains. We clearly see that the fraction of the padded zeros increases as B increases. This is because as more columns are included in each part, the number of padded zeros increases. In the extreme case of block size of one, there is no padded zero. We note that the "natural ordering" is in fact the nested dissection ordering of the global matrix, and it achieves reasonable performance by reducing the fill-ins in the interface. However, we see that postordering the RHS vectors can significantly reduce the fraction of the padded zeros from the natural ordering, and the hypergraph ordering reduces this further. The numbers shown at the bottom of the plots are the maximum and average ratios of the number of padded zeros from the postordering over that from the hypergraph ordering. For example, when $B = 20$ for **tdr190k**, the postordering incurred an average of 26% more and at most 58% more padded zeros than the hypergraph ordering did. The average improvement of the hypergraph ordering over the postordering initially increased with increasing B, but saturated when B was around 60 to 100.

The situation with **matrix211** was very different. Here, we hardly see any improvement using the hypergraph model from using the postordering; i.e., the best-case average reduction of the padded zeros was only 7% with B from 200 to 280. We believe this is mainly because the interfaces of **matrix211** are much sparser, as shown by both the effective density and the fill-ratio in Table 2. The larger effective density provides more chance for the reordered columns to have similar row structures. The hypergraph model seems to exploit this property better than the postordering, and it obtained more improvement for **tdr190k**. On the other hand, the postordering approach



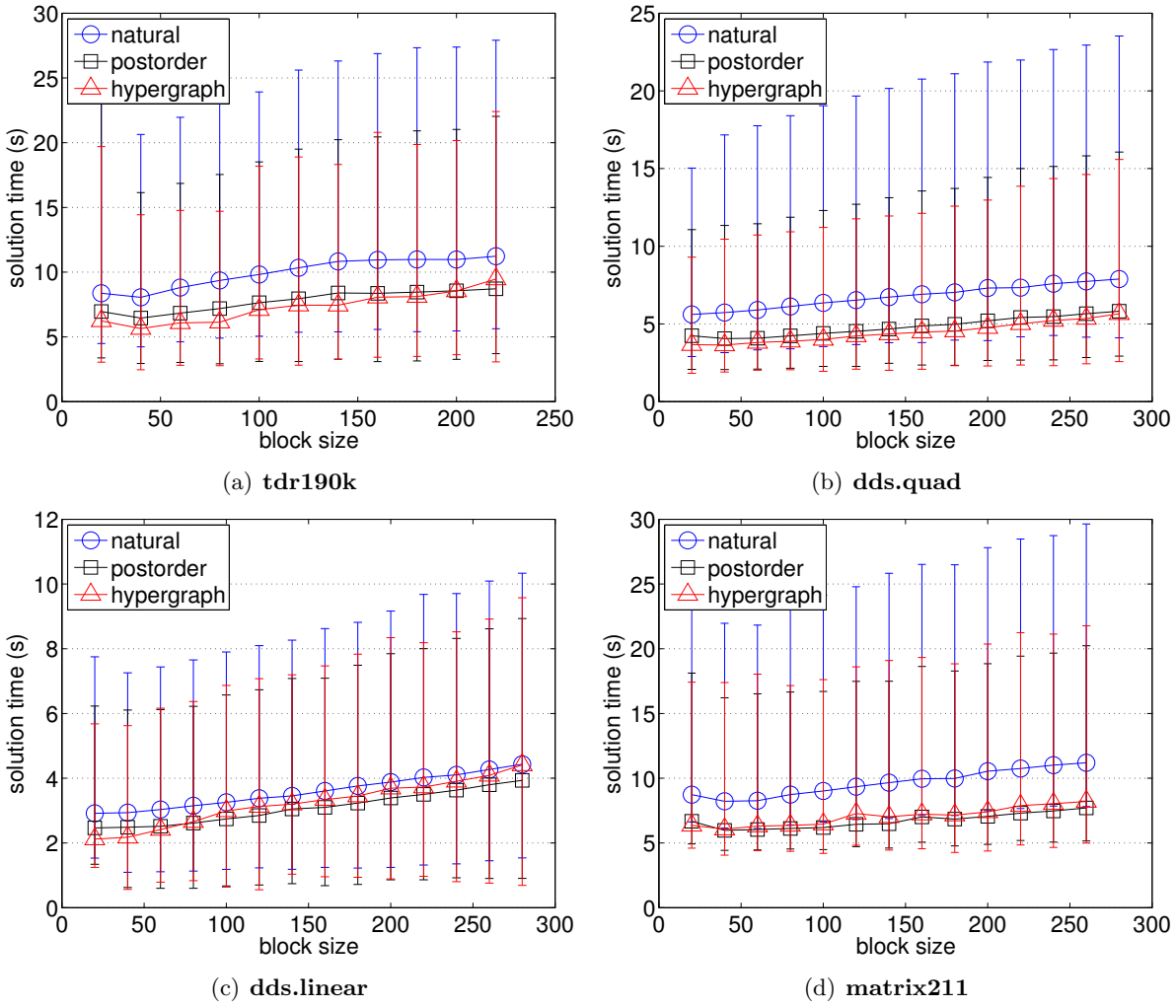
(a) **tdr190k** (b) **dds.quad**

(c) **dds.linear** (d) **matrix211**

Figure 4: Fraction of the padded zeros using different ordering schemes with varying partition block size B.

takes only the first nonzero positions into account, ignoring the other fill-in positions. This works reasonably well if the fill-ratio is small, which is precisely the case for **matrix211**. Since **matrix211** has much less fill than **tdr190k** (an order of magnitude for some subdomains), the postordering worked almost as well as the hypergraph ordering. Note that since the interfaces of **matrix211** are very sparse, larger values of B were used to achieve more benefit.

Triangular solution time. Figure 5 shows the total time spent in the triangular solves $L_\ell^{-1}E_\ell$ using the three orderings. The best time was obtained with B around 60, which is the default in our hybrid solver. We note that our numerical experiments used relatively small matrices. For larger matrices, the time spent in the sparse triangular solves increases significantly, and the triangular solves can be the computational bottleneck. Hence, we are more interested in the speedups gained over the natural ordering than the actual time. The figure shows that the improvement gained by



(a) **tdr190k** (b) **dds.quad**

(c) **dds.linear** (d) **matrix211**

Figure 5: Sparse triangular solution time using the three ordering algorithms.

the hypergraph ordering often increases as the problem becomes more difficult with larger B.

Effect of removing quasi-dense rows. Computing the hypergraph ordering can be expensive. In this section, to reduce this cost, we examine the effect of removing some rows from the solution vectors. For all the experiments below, the empty rows of the solution vectors were removed when computing the hypergraph ordering, and B is fixed to be 60.

In our experiments, we define a quasi-dense row as the one whose fraction of the number of nonzeros in the row is greater than or equal to a density threshold τ. Because a nested dissection algorithm was used to extract the interior subdomains, and each interior subdomain was connected only to a small subset of separators. We found that the majority of the rows are sparse. For example, when eight subdomains are extracted from **tdr190k** test matrix, with $\tau = 0.4$, only about 15% of the rows were quasi-dense. We also observed that the fraction of padded zeroes is largely independent of the threshold until the threshold becomes too small (i.e., $\tau < 10^{-1}$). This

shows that the hypergraph ordering is effective even when a large number of quasi-dense rows were removed. For example with $\tau = 10^{-1}$, about 50% of the rows were removed, but the fraction of the padded zero increased only by a factor of at most 1.04.

Finally, we observed that the time required to compute the hypergraph ordering and the times to solve the sparse triangular systems start to increase when the threshold becomes too small. Specifically, the solution time increased by a factor of about 1.3 when the threshold decreases from 10^{-1} to 10^{-2}. However, similar to the fraction of the padded zeros, the solution times were largely independent of the threshold until it becomes too small (i.e., $\tau < 10^{-1}$). At the same time, we clearly see that the setup time to compute the hypergraph ordering is reduced significantly by using a smaller threshold. With $\tau = 10^{-1}$, the setup time becomes insignificant. This demonstrates the potential of using the hypergraph ordering in practice when combined with the quasi-dense row removal.

6 Conclusion

In this paper, we studied two combinatorial problems to enhance the performance of a hybrid linear solver PDSLin that is based on a non-overlapping domain decomposition method (the Schur complement method). First, we have designed a number of partitioning algorithms to improve the parallel load balance of PDSLin. We summarized the numerical results with the algorithms using the matrices arising from numerical simulations of modeling accelerator cavities and of modeling fusion devices, and presented extensive results of the most effective ones. Among these algorithms, the most promising one was based on a recursive hypergraph bisection (RHB) method using (i) unit net costs; (ii) single constraints with dynamic vertex weights assigned at each bisection step; and (iii) either with `soed` or `cnet` cut-metrics. When the number of subdomains is moderate (e.g., $k = 32$), in comparison to a standard nested graph bisection algorithm, our RHB algorithm improved the load balance which could well offset the modest increase in separator size. As a result, the runtime of PDSLin was reduced.

We also studied two sparse RHS reordering strategies to improve the performance of a supernodal triangular solver, which is used to eliminate the unknowns associated with each interface; one based on a postordering of the elimination tree and the other based on a hypergraph partitioning. The numerical results have shown that these strategies reduce the number of padded zeros in the RHSs, and reduces the runtime of the triangular solver by a factor of up to 1.3.

Acknowledgments

This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. DOE under Contract No. DE-AC02-05CH11231.

References

[1] Center for Extended MHD Modeling (CEMM). http://w3.pppl.gov/cemm/.

[2] Community Petascale Project for Accelerator Science and Simulation (ComPASS). https://compass.fnal.gov.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. Accepted to the SIAM Journal on Scienfific Computing, 2012.

[4] C. Aykanat, B. Cambazoglu, and B. Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 2008.

[5] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, 2004.

[6] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007.

[7] I. Brainman and S. Toledo. Nested-dissection orderings for sparse LU with partial pivoting. *SIAM J. Matrix Analysis and Applications*, 23(4):998–112, 2002.

[8] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42:153–159, 1992.

[9] Ü. V. Çatalyürek, C. Aykanat, and E. Kayaaslan. Hypergraph partitioning-based fill-reducing ordering for symmetric matrices. *SIAM J. Scientific Computing*, 33(4):1996–2023, 2011.

[10] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[11] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[13] J. Demmel, J. Gilbert, and X.S. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: September 2007.

[14] J. Gaidamour and P. Henon. HIPS: a parallel hybrid direct/iterative solver based on a schur complement. In *Proc. PMAA*, 2008.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[16] J. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15:62–79, 1994.

[17] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.

[18] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Scientific Computing*, 22(6):2194–2215, 2001.

[19] G. Karypis and V. Kumar. *hMeTiS: Hypergraph and Circuit Partitioning.* University of Minnesota. `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview`.

[20] G. Karypis and V. Kumar. MeTiS: Serial graph partitioning and computing fill-reducing matrix ordering. University of Minnesota. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[21] G. Karypis, K. Schloegel, and V. Kumar. ParMeTiS: Parallel graph partitioning and sparse matrix ordering. University of Minnesota. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[22] K. Kaya, F.-H. Rouet, and B. Uçar. On partitioning problems with complex objectives. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 334–344. Springer, 2012.

[23] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley–Teubner, Chichester, U.K., 1990.

[24] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 1:134–172, 1990.

[25] F. Pellegrini. SCOTCH - Software package and libraries for graph, mesh and hypergraph partitioning, static mapping, and parallel and sequential sparse matrix block ordering. Laboratoire Bordelais de Recherche en Informatique (LaBRI), http://www.labri.fr/perso/pelegrin/scotch/.

[26] Tz. Slavova. *Parallel triangular solution in an out-of-core multifrontal approach for solving large sparse linear systems.* PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.

[27] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.

[28] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[29] I. Yamazaki and X. S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *Proceedings of the ninth international meeting on high performance computing for computational science, Springer's Lecture Notes in Computer Science*, pages 421–434, 2010.