View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Bilkent University Institutional Repository

PARALLEL SPARSE MATRIX-VECTOR MULTIPLIES AND ITERATIVE SOLVERS

A DISSERTATION SUBMITTED TO

THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Bora Uçar

August, 2005

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Cevdet Aykanat(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Enis Çetin

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Asst. Prof. Dr. Uğur Doğrusöz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Asst. Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Veysi İşler

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

PARALLEL SPARSE MATRIX-VECTOR MULTIPLIES AND ITERATIVE SOLVERS

Bora Uçar
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
August, 2005

Sparse matrix-vector multiply (SpMxV) operations are in the kernel of many scientific computing applications. Therefore, efficient parallelization of SpMxV operations is of prime importance to scientific computing community. Previous works on parallelizing SpMxV operations consider maintaining the load balance among processors and minimizing the total message volume. We show that the total message latency (start-up time) may be more important than the total message volume. We also stress that the maximum message volume and latency handled by a single processor are important communication cost metrics that should be minimized. We propose hypergraph models and hypergraph partitioning methods to minimize these four communication cost metrics in one dimensional and two dimensional partitioning of sparse matrices. Iterative methods used for solving linear systems appear to be the most common context in which SpMxV operations arise. Usually, these iterative methods apply a technique called preconditioning. Approximate inverse preconditioning—which can be applied to a large class of unsymmetric and symmetric matrices—replaces an SpMxV operation by a series of SpMxV operations. That is, a single SpMxV operation is only a piece of a larger computation in the iterative methods that use approximate inverse preconditioning. In these methods, there are interactions in the form of dependencies between the successive SpMxV operations. These interactions necessitate partitioning the matrices simultaneously in order to parallelize a full step of the subject class of iterative methods efficiently. We show that the simultaneous partitioning requirement gives rise to various matrix partitioning models depending on the iterative method used. We list the partitioning models for a number of widely used iterative methods. We propose operations to build a composite hypergraph by combining the previously proposed hypergraph models and show that partitioning the composite hypergraph models addresses the simultaneous matrix partitioning problem. We strove to demonstrate how the proposed partitioning

methods—both the one that addresses multiple communication cost metrics and the other that addresses the simultaneous partitioning problem—help in practice. We implemented a library and investigated the performances of the partitioning methods. These practical investigations revealed a problem that we call message ordering problem. The problem asks how to organize the send operations to minimize the completion time of a certain class of parallel programs. We show how to solve the message ordering problem optimally under reasonable assumptions.

Keywords: Sparse matrices, parallel matrix-vector multiplication, iterative methods, preconditioning, approximate inverse preconditioner, hypergraph partitioning.

ÖZET

PARALEL SEYREK MATRİS-VEKTÖR ÇARPIMI VE DOLAYLI YÖNTEMLER

Bora Uçar
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ağustos, 2005

Seyrek matris-vektör çarpımı (MxV) bir çok bilimsel hesaplama uygulamasının çekirdeğini oluşturmaktadır. Dolayısıyla, MxV çarpımlarının paralelleştirilmesi, bilimsel hesaplama çevrelerinin önem verdiği bir konudur. Bu konuda yapılmış çalışmalar yük dengelemeye ve toplam haberleşme hacmini azaltmaya odaklanmıştır. Bu tezde, toplam haberleşme sayısının da önemli olabileceği gösterilmiştir. Ayrıca, işlemcilere düşen en büyük haberleşme hacminin ve sayısının niceliğinin de önemli olabileceği gösterilmiştir. Bu dört haberleşme ölçütünün azaltılmasını sağlayacak hiperçizge modelleri ve bu modellerin bölümlenmesini sağlayacak yöntemler önerilmiştir. Bu önerilen modellerin ve yöntemlerin, tek boyutlu ve iki boyutlu matris bölümlendirilmesinde nasıl kullanılacağı gösterilmiştir. MxV işleminin en çok kullanıldıgı yer lineer sistem çözümlemelerinde kullanılan dolaylı yöntemlerdir. Bu dolaylı yöntemler çoğu zaman matris iyileştirme teknikleri kullanırlar. Matrislerin yaklaşık tersleriyle iyileştirme tekniği, bir çok simetrik ve simetrik olmayan matris çeşitlerine uygulanabilen ve çokça kullanılan bir tekniktir. Bu teknik, temel olarak, MxV işleminin yerine ardışık MxV işlemlerini koyar. Yani, bir MxV işlemi, matrislerin yaklaşık tersleriyle iyileştirme tekniğini kullanan dolaylı yöntemlerde daha büyük bir hesaplama işleminin sadece küçük bir parçasıdır. Ardışık MxV çarpımlarının arasında etkileşim vardır. Bu etkileşimler, verimli paralelleştirme için matrislerin bir arada bölümlendirilmesini zorunlu kılmaktadır. Bu tezde, bir arada bölümlendirmenin, değişik dolaylı yöntemler için değişik matris bölümlendirme modellerine yol açtıgı gösterilmiştir. Sıkça kullanılan bir çok dolaylı yöntemin hangi matris bölümlendirme modelleriyle paralelleştirilebileceği gösterilmiştir. Bu matris bölümlendirme modellerinin elde edilmesini sağlamak için, önceden önerilmiş hiperçizge modellerini birleştirerek bileşik hiperçizge modelleri geliştiren

işlemler tanımlanmıştır. Bileşik hiperçizge modellerinin bölümlenmesi ile matrislerin bir arada bölümlendirilebileceği gösterilmiştir. Yukarıda bahsedilen çalışmaların pratikte işe yarayıp yaramadıklarını görmek için, paralel MxV işlemini yapan bir program yazdık. Bu programla yaptığımız deneyler sırasında, daha genel bir paralel program sınıfının çalışma süresinin gönder işlemlerinin sırasına bağlı olduğunu gördük. En iyi gönder işlemi sırasının bazı varsayımlar altında nasıl bulunabileceğini gösterdik.

Anahtar sözcükler: Seyrek matrisler, paralel matris-vektör çarpımı, dolaylı yöntemler, matris iyileştirme, matrislerin yaklaşık tersleriyle iyileştirme, hiperçizge bölümleme.

Acknowledgement

Öncelikle tez danışmanım Prof. Dr. Cevdet Aykanat'a çok teşekkür ederim. Onun bilimsel araştırmaya duyduğu heyecan ve yaptığı çalışmalara gösterdiği özen, benim heyecanımı canlı tutmak ve yaptığım çalışmalara özen göstermek için çabalamama sebep oldu. Umarım onun yetiştirmek istediği gibi bir araştırmacı olmuşumdur.

Tez jürimde yer alıp, değerli zamanlarını bu tezi okumaya ve iyileştirmeye harcayan Prof. Dr. Enis Çetin'e, Asst. Prof. Dr. Uğur Doğrusöz'e, Asst. Prof. Dr. Uğur Güdükbay'a ve Assoc. Prof. Dr. Veysi İşler'e çok teşekkür ederim.

Prof. Dr. Mustafa Ç. Pınar'dan çok şey öğrendim. Ondan aldığım dersler ve sorularıma verdiği cevaplar ufkumu genişletti; üretkenliği yapılan işi bitirmenin nasıl bir şey olduğunu gösterdi; akademisyen olmanın, insana entellektüel sorumluluklar yüklediğini hatırlattı.

Prof. Dr. Mehmet Baray'a ve Prof Dr. H. Altay Güvenir'e kalacak yer ve ofis sağladıkları için teşkkür ederim.

Annem Ayten Uçar'a ve babam Hasan Uçar'a beni hep destekledikleri ve seçimlerimi sorgulamadıkları için ne kadar teşekkür etsem azdır. Çalışkan olmanın bir erdem olduğunu, zorluklarla başa çıkmayı, sabırlı olmayı ve de "adam" olmaya dair ne varsa hepsini onlardan öğrendim. Umarım sevgilerine ve ilgilerine lâyık oluyorumdur.

Ağabeyim Uğraş Uçar'a her zaman aklında ve kalbinde olduğumu hissterdiği için teşekkür ederim.

Kayın validem Prof. Dr. Gülsün Baskan'a ve kayın pederim Prof. Dr. Semih Baskan'a bana inandıkları ve güvendikleri için teşekkür ederim.

Sevgili karım Funda Başak'a... Gösterdiğin ilgi, destek, ve sevgi olmasaydı bu tez olur muydu bilmem, olursa nasıl olurdu yine bilmem. Nasıl olursa olsun bu tezi yapmış olmak bu kadar anlamlı olmazdı, ben hiç tam olmazdım, hayat bu kadar güzel olmazdı, ben bunları bilirim. Sana minnettarım.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is devoted to parallelizing sparse matrix-vector multiply (SpMxV) operations. Parallelization of SpMxV operations is an important problem not only because these operations abound in scientific computing, but also because SpMxV operations characterize a wide range of applications which have irregular computational patterns. An SpMxV can be considered as a reduction operation from input space to output space. Hence, solving problems arising in the parallelization of SpMxV operations amounts to solving many such problems arising in a broader context. Besides, the SpMxV operation is a fine-grain computation. That is, it is hard to achieve satisfactory speedup and scalability in the parallel SpMxV operations. Therefore, guaranteeing speedup and scalability for SpMxV will most probably guarantee speedup and scalability in applications that are similar in nature.

We address the SpMxV parallelization problem in the context of iterative methods in which SpMxV operations are performed at each iteration. Such methods are used in so many applications including Google's PageRank computations [79], image deblurring [75], and linear system solutions [5]. An efficient parallelization of SpMxV computations requires the distribution of nonzeros of the input matrix among processors in such a way that the computational loads of the processors are almost equal and the cost of interprocessor communication is low. The nonzero distribution can be one dimensional (1D) or two dimensional

(2D). In 1D rowwise distribution, nonzeros in a row are assigned to the same processor. Similarly, in 1D columnwise distribution, nonzeros in a column are assigned to the same processor. In other words, 1D partitioning approach preserves the row or column integrities. In 2D distribution, the row or column integrities are not preserved and the distribution can be done even on a nonzero basis, i.e., nonzeros can be assigned to processors arbitrarily.

The standard graph partitioning model has been widely used for 1D partitioning of square matrices with symmetric nonzero pattern. This model represents the SpMxV operation as a weighted undirected graph and partitions the vertices in such a way that the parts are equally weighted and the total weight of the edges crossing between the parts is minimized. The partitioning constraint and objective correspond to, respectively, maintaining the computational load balance and minimizing the total message volume. In recent works, Çatalyürek and Aykanat [19, 20], and Hendrickson [49] mentioned the limitations of this standard approach. First, it tries to minimize a wrong objective function, since the edge-cut metric does not model the actual communication volume. Second, it can only express square matrices and produce symmetric partitioning by enforcing identical partitions on the input and output vectors. Symmetric partitioning is desirable for parallel iterative solvers working on symmetric matrices, because it avoids the communication of vector entries during the linear vector operations between the input vectors and output vectors. However, this symmetric partitioning is a limitation for the iterative solvers working on unsymmetric square or rectangular matrices when the input and output vectors do not undergo linear vector operations.

Recently, Çatalyürek, Aykanat, and Pınar [3, 20, 21, 82] proposed hypergraph models for partitioning unsymmetric square and rectangular matrices as well as symmetric matrices with the flexibility of producing unsymmetric partitions on the input and output vectors. Hendrickson and Kolda [52] proposed a bipartite graph model for partitioning rectangular matrices with the same flexibility. A distinct advantage of the hypergraph model over the bipartite graph model is that the hypergraph model correctly encodes the total message volume into its partitioning objective. Several recently proposed alternative partitioning models for parallel computing are discussed in the excellent survey by Hendrickson and Kolda [51]. As noted in the survey, most of the partitioning models mainly consider minimizing the total message volume. However, the communication overhead is a function of the message latency (start-up time) as well. Depending on the machine architecture and the problem size, the communication overhead due to the message latency may be much higher than the overhead due to the message volume [35]. None of the works, listed in the survey, addresses minimizing the total message latency. Furthermore, the maximum message volume and latency handled by a single processor are also crucial cost metrics to be considered in partitionings. As also noted in the survey [51], new approaches that encapsulate these four communication-cost metrics are needed. In Chapter 3, we propose a two phase approach for minimizing these four communication-cost metrics in 1D partitioning of sparse matrices. The material presented in there appears in the literature as [99].

The literature on 2D matrix partitioning is rare. The 2D checkerboard partitioning approaches proposed in [56, 72, 76] are suitable for dense matrices or sparse matrices with structured nonzero patterns that are difficult to exploit. In particular, these approaches do not exploit sparsity to reduce the communication volume. Çatalyürek and Aykanat [19, 23, 24] proposed hypergraph models for 2D sparse matrix partitionings. In the *checkerboard* partitioning model, a matrix is partitioned into row and column blocks. In the *jagged-like* model, matrix is first partitioned into row blocks (or column blocks), and then each row block (or column block) is partitioned into column blocks (or row blocks) independently. In the *fine-grain* model, a matrix is partitioned on nonzero basis. Later, Vastenhouw and Bisseling [105] proposed another 2D partitioning approach. Their approach partitions the matrix into two rectangular blocks each of which further partitioned recursively. This approach produces non-Cartesian partitionings. The *fine-grain* model is reported to achieve better partitionings than the other models in terms of the total communication volume metric [23]. However, it also generates worse partitionings than the other models in terms of the total number of messages metric [23]. In Chapter 4, we adopt our two phase approach from Chapter 3 to minimize the four communication-cost metrics in 2D partitioning of sparse matrices. The work presented in Chapter 4 is independent of the 2D partitioning model. However, we specifically discuss the fine-grain case, because other 2D partitioning models reduce the total number of messages implicitly. The work presented in Chapter 4 appears in the literature as [97].

Usually, iterative methods used for solving linear systems employ *preconditioning* techniques. Roughly speaking, preconditioning techniques modify the given linear system to accelerate convergence. Applications of explicit preconditioners in the form of approximate inverses or factored approximate inverses are amenable to parallelization. Because, these techniques require SpMxV operations with the approximate inverse or factors of the approximate inverse at each step. In other words, preconditioned iterative methods perform SpMxV operations with both coefficient and preconditioner matrices in a step. Therefore, parallelizing a full step of these methods requires the coefficient and preconditioner matrices to be well partitioned, e.g., processors' loads are balanced and communication costs are low in both multiply operations. To meet this requirement, the coefficient and preconditioner matrices should be partitioned simultaneously. Simultaneous partitioning problem is formulated in terms of bipartite graph partitioning [52]. However, the formulation is based on the cut edges and hence has the same shortcoming in capturing the total communication volume. In Chapter 5, we propose methods to combine previously proposed hypergraph models [21] to build composite hypergraph models for partitioning the preconditioner and coefficient matrices simultaneously. In particular, we show how to use the composite models to obtain 1D partitions on a matrix and its approximate inverse preconditioner for efficiently parallelizing a full step in the preconditioned iterative methods. Our contribution in Chapter 5 is that we extend hypergraph models to obtain simultaneous partitionings on more than one matrix with the goals of minimizing the total communication volume and maintaining the computational load balance. The work presented in Chapter 5 is submitted to a journal [102].

The SpMxV algorithms that are based on 2D partitionings of the matrices and the sparse matrix-sparse matrix-vector multiply operations that are performed in the preconditioned iterative methods possess a common trait. In these operations, the computations take place between two irregular communication phases. Such settings give rise to a problem that we call message ordering problem. In such cases, the order in which messages are sent affects the completion time of the parallel programs. We formally define the message ordering problem in Chapter 6 and solve it optimally under reasonable assumptions. The work presented in this chapter appears in [101].

We have developed a library which provides efficient implementation of SpMxV operations. The library includes subroutines which operate on 1D and 2D partitioned matrices. The library also provides building blocks of the iterative methods. The algorithms implemented in the library are fine tuned in order to overlap communications and computations to the most possible extent. To the best of our knowledge, none of the publicly available libraries have the same extent in overlapping computations and communications. The internals of the library are discussed in Chapter 7. A preliminary version of the material presented in Chapter 7 appears in [98].

Chapter 2

Preliminaries

In this chapter, we review parallel algorithms for matrix-vector multiplies [52, 95, 98, 99] and summarize the hypergraph partitioning methods [21, 99] which enable efficient parallelization.

2.1 Parallel SpMxV based on 1D partitioning

Suppose that the rows and columns of an $m \times n$ matrix A are permuted into a $K \times K$ block structure

$$A_{BL} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1K} \\ A_{21} & A_{22} & \cdots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{K1} & A_{K2} & \cdots & A_{KK} \end{bmatrix} \tag{2.1}$$

for *rowwise* or *columnwise* partitioning, where K is the number of processors. Block $A_{k\ell}$ is of size $m_k \times n_\ell$, where $\sum_k m_k = m$ and $\sum_\ell n_\ell = n$. In rowwise partitioning, each processor P_k holds the kth row stripe $[A_{k1} \cdots A_{kK}]$ of size $m_k \times n$. In columnwise partitioning, P_k holds the kth column stripe $[A_{1k}^T \cdots A_{Kk}^T]^T$ of size $m \times n_k$. In rowwise partitioning, the row stripes should have nearly equal number of nonzeros for having the computational load balance among processors. The same requirement exists for the column stripes in columnwise partitioning.

2.1.1 Row-parallel algorithm

Consider matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix is partitioned rowwise. A rowwise partition of matrix A defines a partition on the output vector y. The input vector x is assumed to be partitioned conformably with the column permutation of matrix A. In particular, y and x vectors are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. That is, processor P_k holds x_k and is responsible for computing y_k.

In [52, 86, 87, 95, 99], authors discuss the implementation of parallel SpMxV operations where the matrix is partitioned rowwise. The common algorithm executes the following steps at each processor P_k:

1. For each nonzero off-diagonal block $A_{\ell k}$, send sparse vector $\hat{x}_k^\ell$ to processor P_ℓ, where $\hat{x}_k^\ell$ contains only those entries of x_k corresponding to the nonzero columns in $A_{\ell k}$.
2. Compute the diagonal block product $y_k^k = A_{kk} \times x_k$, and set $y_k = y_k^k$.
3. For each nonzero off-diagonal block $A_{k\ell}$, receive $\hat{x}_\ell^k$ from processor P_ℓ, then compute $y_k^\ell = A_{k\ell} \times \hat{x}_\ell^k$, and update $y_k = y_k + y_k^\ell$.

Since the matrix is distributed rowwise, we call the above algorithm *row-parallel*. In Step 1, P_k might be sending the same x_k-vector entry to different processors according to the sparsity pattern of the respective column of A. This multicast-like operation is referred to here as *expand* operation.

2.1.2 Column-parallel algorithm

Consider matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix A is partitioned columnwise. The columnwise partition of matrix A defines a partition on the input vector x. The output vector y is assumed to be partitioned conformably with the row permutation of matrix A. In particular, y and x vectors are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. That is, processor P_k holds x_k and is responsible for computing y_k. Since the matrix is distributed columnwise, we derive a *column-parallel algorithm* for this case. The column-parallel algorithm executes the following steps at processor P_k:

1. For each nonzero off-diagonal block $A_{\ell k}$, form sparse vector $\hat{y}_\ell^k$ which contains only those results of $y_\ell^k = A_{\ell k} \times x_k$ corresponding to the nonzero rows in $A_{\ell k}$. Send $\hat{y}_\ell^k$ to processor P_ℓ.
2. Compute the diagonal block product $y_k^k = A_{kk} \times x_k$, and set $y_k = y_k^k$.
3. For each nonzero off-diagonal block $A_{k\ell}$ receive partial-result vector $\hat{y}_k^\ell$ from processor P_ℓ, and update $y_k = y_k + \hat{y}_k^\ell$.

The multinode accumulation on the w_k-vector entries is referred to here as *fold* operation.

2.2 Parallel SpMxV based on 2D partitioning

Consider the matrix-vector multiply of the form $y \leftarrow Ax$, where y and x are column vectors of size m and n, respectively, and the matrix is partitioned in two dimensions among K processors. The vectors y and x are partitioned as $y = [y_1^T \cdots y_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where y_k and x_k are column vectors of size m_k and n_k, respectively. As before we have $\sum_k m_k = m$ and $\sum_\ell n_\ell = n$. Processor

P_k holds x_k and is responsible for computing y_k. Nonzeros of a processor P_k can be visualized as a sparse matrix A^k

$$A^k = \begin{bmatrix} A^k_{11} & \cdots & A^k_{1k} & \cdots & A^k_{1K} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A^k_{k1} & \cdots & A^k_{kk} & \cdots & A^k_{kK} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A^k_{K1} & \cdots & A^k_{Kk} & \cdots & A^k_{KK} \end{bmatrix} \tag{2.2}$$

of size $m \times n$, where $A = \sum A^k$. Here, the blocks in row-block stripe $A^k_{k*} = \{A^k_{k1}, \cdots, A^k_{kk}, \cdots, A^k_{kK}\}$ have row dimension of size m_k. Similarly, the blocks in column-block stripe $A^k_{*k} = \{A^k_{1k}, \cdots, A^k_{kk}, \cdots, A^k_{Kk}\}$ have column dimension of size n_k. The x-vector entries that are to be used by processor P_k are represented as $x^k = [x^k_1, \cdots, x^k_k, \cdots, x^k_K]$, where x^k_k corresponds to x_k and other x^k_ℓ are belonging to some other processor P_ℓ. The y-vector entries that processor P_k computes partial results for are represented as $y^k = [y^k_1, \cdots, y^k_k, \cdots, y^k_K]$, where y^k_k corresponds to y_k and other y^k_ℓ are to be sent to some other processor P_ℓ. Since the parallelism is achieved on nonzero basis rather than complete rows or columns, we derive a *row-column-parallel* SpMxV algorithm. This algorithm executes the following steps at each processor P_k:

1. For each $\ell \neq k$ having nonzero column-block stripe A^ℓ_{*k}, send sparse vector $\hat{x}^\ell_k$ to processor P_ℓ, where $\hat{x}^\ell_k$ contains only those entries of x_k corresponding to the nonzero columns in A^ℓ_{*k}.

2. Compute the column-block stripe product $y^k = A^k_{*k} \times x^k_k$.

3. For each nonzero column-block stripe $A^k_{*\ell}$, receive $\hat{x}^k_\ell$ from processor P_ℓ, then compute $y^k = y^k + A^k_{*\ell} \times \hat{x}^k_\ell$, and set $y_k = y^k_k$.

4. For each nonzero row-block stripe $A^k_{\ell *}$, form sparse partial-result vector $\hat{y}^k_\ell$ which contains only those results of $y^k_\ell = A^k_{\ell *} \times x^k$ corresponding to the nonzero rows in $A^k_{\ell *}$. Send $\hat{y}^k_\ell$ to processor P_ℓ.

5. For each $\ell \neq k$ having nonzero row-block stripe A^ℓ_{k*} receive partial-result vector $\hat{y}^\ell_k$ from processor P_ℓ, and update $y_k = y_k + \hat{y}^\ell_k$.

2.3 Hypergraph partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. Every net n_i is a subset of vertices. The vertices of a net are also called its *pins*. The size of a net n_i is equal to the number of its pins, i.e., $|n_i|$. The set of nets that contain vertex v_j is denoted by $Nets(v_j)$, which is also extended to a set of vertices appropriately. The degree of a vertex v_j is denoted by $d_j = |Nets(v_j)|$. Weights can be associated with vertices. We use $w(v_j)$ to denote the weight of the vertex v_j.

$\Pi = \{\mathcal{V}_1, \cdots, \mathcal{V}_K\}$ is a K-way vertex partition of $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ if each part $\mathcal{V}_k$ is non empty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. In Π, a net is said to *connect* a part if it has at least one pin in that part. The *connectivity set* Λ_i of a net n_i is the set of parts connected by n_i. The *connectivity* $\lambda_i = |\Lambda_i|$ of a net n_i is the number of parts connected by n_i. A net is said to be cut if it connects more than one part and uncut otherwise. The cut and uncut nets are also referred to as external and internal nets. In Π, weight of a part is the sum of the weights of vertices in that part, e.g., $w(\mathcal{V}_k) = \sum_{v_j \in \mathcal{V}_k} w(v_j)$.

In the hypergraph partitioning problem, the objective is to minimize the *cutsize*:

$$cutsize(\Pi) = \sum_{n_i \in \mathcal{N}} (\lambda_i - 1). \tag{2.3}$$

This objective function is widely used in the VLSI community [71] and in the scientific computing community [3, 21, 99], and it is referred to as the *connectivity* -1 cutsize metric. The partitioning constraint is to maintain a balance on part weights, i.e.,

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \epsilon, \tag{2.4}$$

where W_{max} is the weight of the part with the maximum weight, W_{avg} is the average part weight, and ϵ is a predetermined imbalance ratio. This problem is NP-hard [71].

A recent variant of the above problem is the multi-constraint hypergraph partitioning problem [19, 24, 65] in which each vertex has a vector of weights associated with it. In this problem, the partitioning objective is the same as that given in Eq. 2.3, however, the partitioning constraint is to satisfy a balancing constraint associated with each weight. Another variant is the multi-objective hypergraph partitioning [1, 90, 92] in which there are two or more objectives to be minimized. Specifically, a given net contributes different costs to different objectives.

The multilevel approach [17, 55] is frequently used in graph and hypergraph partitioning tools. The approach consists of three phases: coarsening, initial partitioning, and uncoarsening. In the first phase, a multilevel clustering is applied starting from the original graph/hypergraph by adopting various matching/clustering heuristics until the number of vertices in the coarsened graph/hypergraph falls below a predetermined threshold. In the second phase, a partition is obtained on the coarsest graph/hypergraph using various heuristics. In the third phase, the partition found in the second phase is successively projected back towards the original graph/hypergraph by refining the projected partitions on the intermediate level graphs/hypergraphs using various heuristics. A common refinement heuristic is FM, which is a localized iterative improvement method proposed for graph/hypergraph bipartitioning by Fiduccia and Mattheyses [38] as a faster implementation of the KL algorithm proposed by Kernighan and Lin [67]. The multilevel paradigm overcame the localized nature of the refinement heuristics and led to successful partitioning tools [22, 47, 54, 63, 66]. The multilevel paradigm is also used in addressing the aforementioned variants of the hypergraph partitioning problem.

2.4 Hypergraph models for 1D partitioning

It is inherent in the parallel SpMxV algorithms given above and existent in the literature [21, 51, 52, 99] that in partitioning a matrix the key is to find permutation matrices P and Q such that most of the nonzeros of the matrix $PAQ = A_{BL}$ (Eq. 2.1) are in the diagonal blocks. If the matrix is partitioned rowwise, then the permutation P denotes both the partition on the rows of the matrix and the partition on the output vector. The permutation Q denotes the partition on the input vector.

The previously proposed computational hypergraph models [21] find permutations P and Q by modeling sparse matrices with hypergraphs. In the column-net hypergraph model, the matrix A is represented as a hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ for rowwise decomposition. Vertex and net sets $\mathcal{V}_{\mathcal{R}}$ and $\mathcal{N}_{\mathcal{C}}$ correspond to the rows and columns of A, respectively. There exist one vertex v_i and one net n_j for each row i and column j, respectively. The net n_j contains the vertices corresponding to the rows that have a nonzero in column j. That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$. Each vertex v_i corresponds to the atomic task of computing the inner product of the row i with the column vector x. Hence, the computational weight associated with the vertex v_i is equal to the number of nonzeros in row i. The nets of $\mathcal{H}$ represent the dependency relations of the atomic tasks on the x-vector entries. Therefore, each net n_j denotes the set of atomic tasks that need x_j.

Figure 2.1 shows a matrix and its column-net hypergraph model. In the figure, the white and black circles represent, respectively, the vertices and nets, and straight lines show pins. A four-way partition on the hypergraph is shown by four big circles encompassing the vertices of the hypergraph.

Given a partition Π on a column-net hypergraph $\mathcal{H}$, the permutations P and Q can be found as follows. The permutation P is completely defined by the vertex partition. The rows corresponding to the vertices in $\mathcal{V}_k$ are mapped to processor P_k and therefore permuted before the rows corresponding to the vertices in $\mathcal{V}_\ell$ for $1 \leq k < \ell \leq K$. In Fig. 2.1, the permutation on the rows of A

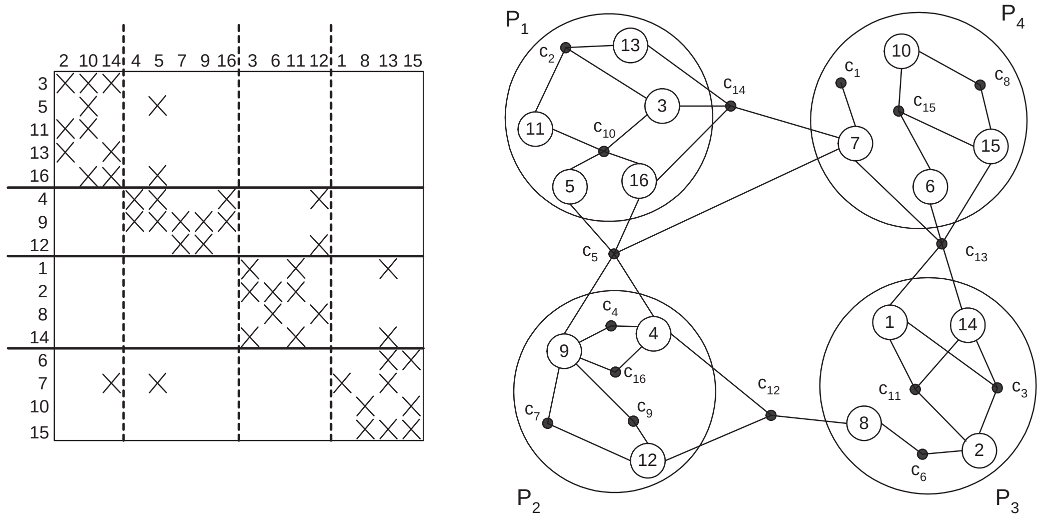


Figure 2.1: A matrix, its column-net hypergraph model, and a four-way rowwise partitioning.

is shown by the permuted row indices, where the horizontal solid lines separate row stripes that belong to different processors. There are many ways to define permutation Q under the partition Π. However, we seek *consistent permutations* which map the column j, associated with net n_j, into any one of the parts in Λ_j. For example, in Fig. 2.1, the net c_5 connects the parts P_1, P_2, and P_4. Therefore, column 5 should be mapped either to the part P_1 or P_2 or P_4 in any consistent permutation. The figure shows a consistent permutation on the columns of A, where the vertical dashed lines separate virtual column stripes that belong to different processors. Once the permutations are found, the rows of the matrix and the vectors are distributed among the processors as discussed in §2.1.1 and §2.1.2. For example, in Fig. 2.1, the processor P_2 is set to be responsible for computing the inner products of x with the rows 4, 9, and 12 which reside in the second row stripe. In the figure, P_2 holds x_4, x_5, x_7, x_9, and x_{16} and thus expands x_5 to the processors P_1 and P_4. Observe that the net c_5 connects the parts P_1, P_2, and P_4. This association between the connectivity of nets and the communication requirements is not accidental as shown by the following theorem.

Theorem 2.1 *Let* Π *be a partition on the column-net hypergraph model of a* *given matrix A. Let P be the row permutation induced by the vertex partition Π, and Q be a consistent column permutation. Then, the cutsize of the partition Π quantifies the total communication volume in the row-parallel $y \leftarrow Ax$ multiply.*

Proof. Consider the internal nets. Because of the consistency of the permutation Q, the x-vector entries associated with these nets are mapped to the unique processor that needs them. Hence, no communication occurs for the x-vector entries associated with the internal nets. Consider an external net n_e with the connectivity set Λ_e. Each processor in the set Λ_e needs x_e. One of them owns x_e as imposed by the consistent permutation Q. The owner should send x_e to each processor in Λ_e. That is, for each x_e there are a total of $|\Lambda_e| - 1 = \lambda_e - 1$ messages carrying x_e. The overall sum of these quantities matches the cutsize definition given in Eq. 2.3. Details can be found in [21]. □

Using Theorem 2.1, it is concluded in [21] that hypergraph partitioning objective and constraint correspond, respectively, to minimizing the total communication volume and maintaining the computational load balance. In Fig 2.1, the cutsize and hence the total communication volume is five words, and the part weights and hence the computational loads of the processors are 12, 12, 11, and 11.

In [21], the permutation Q is generated by using a policy which maps n_i to the part holding v_i. This policy is chosen to generate symmetric partitioning. Note that if symmetric partitioning is required, then there is no freedom in defining Q. If, however, unsymmetric partitioning is allowed, it is possible to exploit the leeway in defining a consistent permutation to achieve several goals. For example, we [99] exploit the leeway to minimize the total number of messages and to obtain balance on the communication volume loads of the processors, where these two metrics are defined in terms of sends. Vastenhouw and Bisseling [105] exploit the leeway in order to minimize the maximum communication volume load of a processor defined in terms of sends and receives.

The row-net hypergraph model for sparse matrices [21] can be used to obtain columnwise partitioning on a matrix A. In the row-net model, the vertices

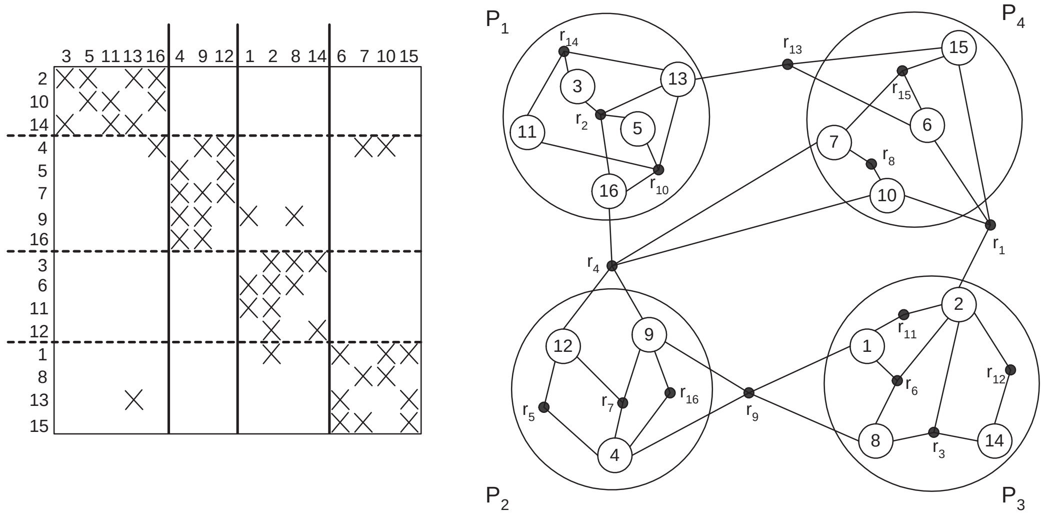


Figure 2.2: A matrix, its row-net hypergraph model, and a four-way columnwise partitioning.

and nets represent the columns and rows of A, respectively. Figure 2.2 shows a matrix and its row-net hypergraph model. Partitioning the row-net hypergraph minimizes the total communication volume in column-parallel SpMxV and maintains balance on the computational loads of the processors through generating permutation matrices P and Q as before. In this model, Q is completely determined by the vertex partition on the hypergraph, and P is required to be a consistent permutation. In the figure, the vertical solid lines separate column stripes, and the horizontal dashed lines separate virtual row stripes that belong to different processors. The column stripes determine the computational loads of processors. The virtual row stripes designate which processor will fold on which y-vector entries. For example, in Fig. 2.2, the processor P_2 is set to be responsible for folding the y-vector entries that correspond to the rows in the second virtual row stripe. Therefore, the processors P_1 and P_4 have to send their contribution for y_4 to P_2. Again, there is the same association between the connectivity of the nets and the total communication volume. In the figure, the cutsize and hence the total communication volume is five words.

Chapter 3

Communication cost metrics for 1D SpMxV

This chapter addresses the problem of one-dimensional partitioning of structurally unsymmetric square and rectangular sparse matrices for parallel matrix-vector and matrix-transpose-vector multiplies. The objective is to minimize the communication cost while maintaining the balance on computational loads of processors. Most of the existing partitioning models consider only the total message volume hoping that minimizing this communication-cost metric is likely to reduce other metrics. However, the total message latency (start-up time) may be more important than the total message volume. Furthermore, the maximum message volume and latency handled by a single processor are also important metrics. We propose a two-phase approach that encapsulates the minimization of all these four communication-cost metrics. The objective in the first phase is to minimize the total message volume while maintaining the computational-load balance. The objective in the second phase is to encapsulate the remaining three communication-cost metrics. We propose communication-hypergraph and partitioning models for the second phase. We then present several methods for partitioning communication hypergraphs. Experiments on a wide range of test matrices show that the proposed approach yields very effective partitioning results. A parallel implementation on a PC cluster verifies that the theoretical improvements shown by partitioning results hold in practice.

3.1 Introduction

Repeated matrix-vector and matrix-transpose-vector multiplies that involve the same large, sparse, structurally unsymmetric square or rectangular matrix are the kernel operations in various iterative algorithms. For example, iterative methods such as the conjugate gradient normal equation error and residual methods (CGNE and CGNR) [42, 86] and the standard quasi-minimal residual method (QMR) [40], used for solving unsymmetric linear systems, require computations of the form $y \leftarrow Ax$ and $w \leftarrow A^T z$ in each iteration, where A is an unsymmetric square coefficient matrix. The LSQR [80] method, used for solving the least squares problem, and the Lanczos method [42], used for computing the singular value decomposition, require frequent computations of the form $y \leftarrow Ax$ and $w \leftarrow A^T z$, where A is a rectangular matrix. Iterative methods used in solving the normal equations that arise in interior point methods for linear programming require repeated computations of the form $y \leftarrow AD^2A^T z$, where A is a rectangular constraint matrix and D is a diagonal matrix. Rather than forming the coefficient matrix AD^2A^T, which may be quite dense, the above computation is performed as $w \leftarrow A^T z$, $x \leftarrow D^2 w$ and $y \leftarrow Ax$. The surrogate constraint method [77, 78, 96, 107], which is used for solving the linear feasibility problem, requires decoupled matrix-vector and matrix-transpose vector multiplies involving the same rectangular matrix.

In the framework of this chapter, we assume that no computational dependency exists between the input and output vectors x and y of the $y \leftarrow Ax$ multiply. The same assumption applies to the input and output vectors z and w of the $w \leftarrow A^T z$ multiply. In some of the above applications, the input vector of the second multiply is obtained from the output vector of the first one—and vice versa—through linear vector operations because of intra- and inter-iteration dependencies. So, linear operations may occur only between the vectors that belong to the same space. In this setting, w and x are input-space vectors, whereas z and y are output-space vectors. These assumptions hold naturally in some of the above applications that involve a rectangular matrix. Since input- and output-space vectors are of different dimensions, they cannot undergo linear vector operations. In the remaining applications, which involve a square matrix, a computational dependency does not exist between input- and output-space vectors because of the nature of the underlying method. Our goal is the parallelization of the computations in the above iterative algorithms through rowwise or columnwise partitioning of matrix A in such a way that the communication overhead is minimized and the computational-load balance is maintained.

In this chapter, we do not address the efficient parallelization of matrix-vector multiplies involving more than one matrix with different sparsity patterns. Handling such cases requires simultaneous partitioning of the participating matrices in a method that considers the complicated interaction among the efficient parallelizations of the respective matrix-vector multiplies. The most notable cases are the preconditioned iterative methods that use an explicit preconditioner such as an approximate inverse [6, 11, 46] $M \approx A^{-1}$. These methods involve matrix-vector multiplies with M and A. The present work can be used in such cases by partitioning matrices independently. However, this approach would suffer from communication required for reordering the vector entries between the two matrix-vector multiplies. We address the simultaneous partitioning problem in Chapter 5.

In this chapter, we propose a two-phase approach for minimizing multiple communication-cost metrics. The objective in the first phase is to minimize the total message volume while maintaining the computational-load balance. This objective is achieved through partitioning matrix A within the framework of the existing 1D matrix partitioning methods. The partitioning obtained in the first phase is an input to the second phase so that it determines the computational loads of processors while setting a lower bound on the total message volume. The objective in the second phase is to encapsulate the remaining three communication-cost metrics while trying to attain the total message volume bound as much as possible. The metrics minimized in the second phase are not simple functions of the cut edges or hyperedges or vertex weights defined in the existing graph and hypergraph models even in the multi-objective [90] and multi-constraint [64] frameworks. Besides, these metrics cannot be assessed before a partition is defined. Hence, we anticipate a two phase approach. Pınar and Hendrickson [83] also adopt a multiphase approach for handling complex partitioning objectives. Here, we focus on the second phase and do not go back and forth between the phases. Therefore, our contribution can be seen as a post-process to the existing partitioning methods. For the second phase, we propose a *communication-hypergraph* partitioning model. The vertices of the communication hypergraph, with proper weighting, represent primitive communication operations, and the nets represent processors. By partitioning the communication hypergraph into equally weighted parts such that nets are split among as few vertex parts as possible, the model maintains the balance on message-volume loads of processors and minimizes the total message count. The model also enables incorporating the minimization of the maximum message-count metric.

We present how to perform matrix-vector and matrix-transpose vector multiplies with the same coefficient matrix in §3.2 and suggest the reader review the background material on the parallel matrix-vector multiplies and hypergraph partitioning problem given in Chapter 2. The proposed communication-hypergraph and partitioning models are discussed in §3.3. Section 3.4 presents three methods for partitioning communication hypergraphs. Experimental results are presented and discussed in §3.5.

3.2 Background

Recall from Chapter 2 that we permute the rows and columns of an $m \times n$ matrix A into a $K \times K$ block structure

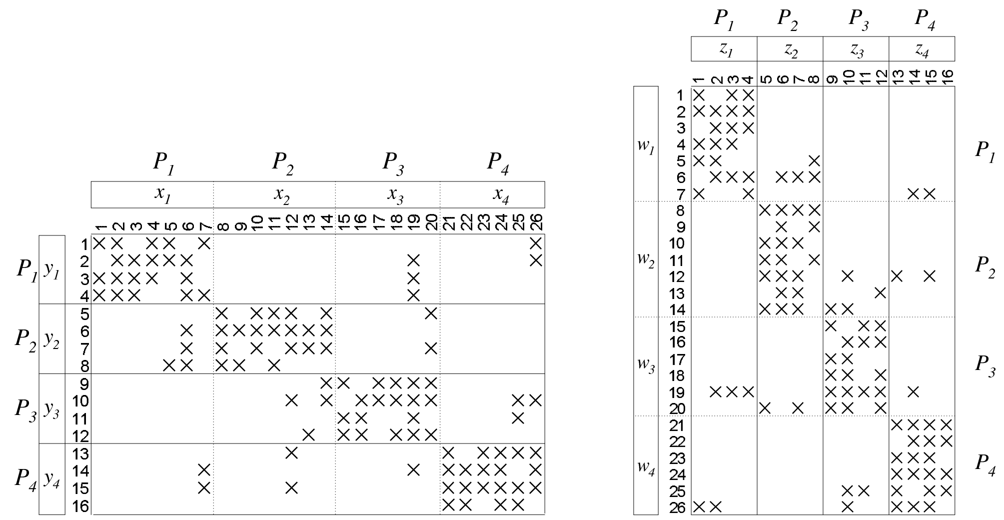
$$A_{BL} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1K} \\ A_{21} & A_{22} & \cdots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{K1} & A_{K2} & \cdots & A_{KK} \end{bmatrix} \tag{3.1}$$

for *rowwise* or *columnwise* partitioning, where K is the number of processors. Block $A_{k\ell}$ is of size $m_k \times n_\ell$, where $\sum_k m_k = m$ and $\sum_\ell n_\ell = n$. In rowwise partitioning, each processor P_k holds the kth row stripe $[A_{k1} \cdots A_{kK}]$ of size $m_k \times n$. In columnwise partitioning, P_k holds the kth column stripe $[A_{1k}^T \cdots A_{Kk}^T]^T$ of size $m \times n_k$.

3.2.1 Matrix-vector and matrix-transpose-vector multiplies

Consider an iterative algorithm involving repeated matrix-vector and matrix-transpose-vector multiplies of the form $y \leftarrow Ax$ and $w \leftarrow A^T z$. A rowwise partition of A induces a columnwise partition of A^T. So, the partition on the z vector defined by the columnwise partition of A^T will be conformable with that on the y vector. That is, $z = [z_1^T \cdots z_K^T]^T$ and $y = [y_1^T \cdots y_K^T]^T$, where z_k and y_k are both of size m_k for $k = 1, \ldots, K$. In a dual manner, the columnwise permutation of A induces a rowwise permutation of A^T. So, the partition on the w vector induced by the rowwise permutation of A^T will be conformable with that on the x vector. That is, $w = [w_1^T \cdots w_K^T]^T$ and $x = [x_1^T \cdots x_K^T]^T$, where w_k and x_k are both of size n_k for $k = 1, \ldots, K$.

We use a *column-parallel* algorithm for $w \leftarrow A^T z$ and use the row-parallel algorithm for $y \leftarrow Ax$ thus obtain a *row-column-parallel* algorithm. In $y \leftarrow Ax$, processor P_k holds x_k and computes y_k. In $w \leftarrow A^T z$, P_k holds z_k and computes w_k.



(a) (b)

Figure 3.1: 4×4 block structures of a sample matrix A: (a) A_{BL} for row-parallel $y \leftarrow Ax$ and (b) $(A^T)_{BL}$ for column-parallel $w \leftarrow A^T z$.

3.2.2 Analyzing communication requirements of SpMxV

Here, we restate and summarize the facts given in [20, 52] for the communication requirement in the row-parallel $y \leftarrow Ax$ and column-parallel $w \leftarrow A^T z$. We will use Fig. 3.1 for a better understanding of these facts. Figure 3.1 displays 4×4 block structures of a 16×26 sample matrix A and its transpose. In Fig. 3.1(a), horizontal solid lines identify a partition on the rows of A and on vector y, whereas vertical dashed lines identify *virtual column stripes* inducing a partition on vector x. In Fig. 3.1(b), vertical solid lines identify a partition on the columns of A^T and on vector z, whereas horizontal dashed lines identify *virtual row stripes* inducing a partition on vector w. The computational-load balance is maintained by assigning $25, 26, 25$, and 25 nonzeros to processors P_1, P_2, P_3, and P_4, respectively.

FACT 1 *The number of messages sent by processor P_k in row-parallel $y \leftarrow Ax$ is equal to the number of nonzero off-diagonal blocks in the kth virtual column stripe of A. The volume of messages sent by P_k is equal to the sum of the number of nonzero columns in each off-diagonal block in the kth virtual column* *stripe.*

In Fig 3.1(a), P_2, holding x-vector block $x_2 = x[8:14]$, sends vector $\hat{x}_2^3 = x[12:14]$ to P_3 because of nonzero columns $12, 13$, and 14 in A_{32}. P_3 needs those entries to compute $y[9]$, $y[10]$, and $y[12]$. Similarly, P_2 sends $\hat{x}_2^4 = x[12]$ to P_4 because of the nonzero column 12 in A_{42}. So, the number of messages sent by P_2 is 2 with a total volume of 4 words. Note that P_2 effectively expands $x[12]$ to P_3 and P_4.

FACT 2 *The number of messages sent by processor P_k in column-parallel $w \leftarrow A^T z$ is equal to the number of nonzero off-diagonal blocks in the kth column stripe of A^T. The volume of messages sent by P_k is equal to the sum of the number of nonzero rows in each off-diagonal block in the kth column stripe of A^T.*

In Fig. 3.1(b), P_3, holding z-vector block $z_3 = z[9:12]$, computes the off-diagonal block products $w_2^3 = (A^T)_{23} \times z_3$ and $w_4^3 = (A^T)_{43} \times z_3$. It then forms vectors $\hat{w}_2^3$ and $\hat{w}_4^3$ to be sent to P_2 and P_4, respectively. $\hat{w}_2^3$ contains its contribution to $w[12:14]$ due to the nonzero rows 12, 13, and 14 in $(A^T)_{23}$. Accordingly, $\hat{w}_4^3$ contains its contribution to $w[25:26]$ due to the nonzero rows 25 and 26 in $(A^T)_{43}$. So, P_3 sends 2 messages with a total volume of 5 words.

FACT 3 *Communication patterns of $y \leftarrow Ax$ and $w \leftarrow A^T z$ multiplies are duals of each other. If a processor P_k sends a message to P_ℓ containing some x_k entries in $y \leftarrow Ax$, then P_ℓ sends a message to P_k containing its contributions to the corresponding w_k entries in $w \leftarrow A^T z$.*

Consider the communication between processors P_2 and P_3. In $y \leftarrow Ax$, P_2 sends a message to P_3 containing $x[12:14]$, whereas in $w \leftarrow A^T z$, P_3 sends a dual message to P_2 containing its contributions to $w[12:14]$.

FACT 4 *The total number of messages in $y \leftarrow Ax$ or $w \leftarrow A^T z$ multiply is equal to the number of nonzero off-diagonal blocks in A or A^T. The total volume* *of messages is equal to the sum of the number of nonzero columns or rows in each off-diagonal block in A or A^T, respectively.*

In Figure 3.1, there are 9 nonzero off-diagonal blocks, containing a total of 13 nonzero columns or rows in A or A^T. Hence, the total number of messages in $y \leftarrow Ax$ or $w \leftarrow A^T z$ is 9, and the total volume of messages is 13 words.

3.3 Models for minimizing communication cost

In this section, we present our hypergraph partitioning models for the second phase of the proposed two-phase approach. We assume that a K-way rowwise partition of matrix A is obtained in the first phase with the objective of minimizing the total message volume while maintaining the computational-load balance.

3.3.1 Row-parallel $y \leftarrow Ax$

Let A_{BL} denote a block-structured form (see Eq. 3.1) of A for the given rowwise partition.

3.3.1.1 Communication-hypergraph model

We identify two sets of columns in A_{BL}: *internal* and *coupling*. Internal columns have nonzeros only in one row stripe. The x-vector entries that are associated with these columns should be assigned to the respective processors to avoid unnecessary communication. Coupling columns have nonzeros in more than one row stripe. The x-vector entries associated with the coupling columns, referred to as x_C, necessitate communication. The proposed approach considers partitioning these x_C-vector entries to reduce the total message count and the maximum message volume. Consequences of this partitioning on the total message volume will be addressed in §3.3.4.

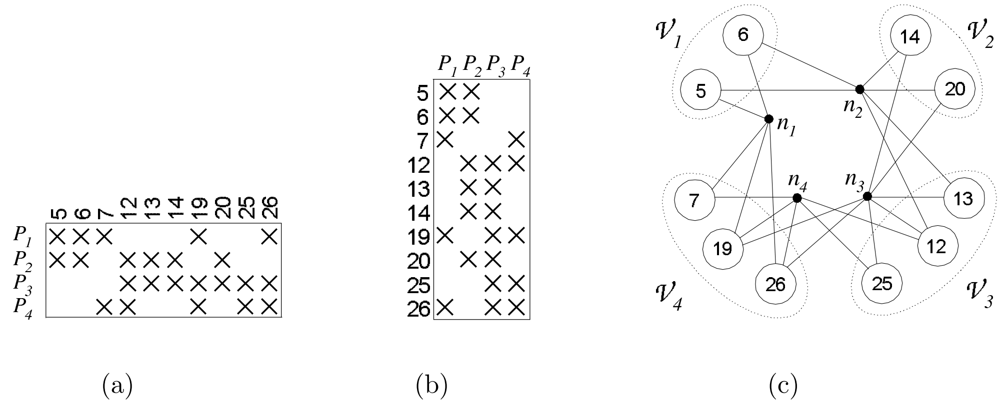


Figure 3.2: Communication matrices (a) C for row-parallel $y \leftarrow Ax$ (b) C^T for column-parallel $w \leftarrow A^T z$, and (c) the associated communication hypergraph and its 4-way partition.

We propose a rowwise compression of A_{BL} to construct a matrix C, referred to here as the *communication matrix*, which summarizes the communication requirement of row-parallel $y \leftarrow Ax$. First, for each $k = 1, \ldots, K$, we compress the kth row stripe into a single row with the sparsity pattern being equal to the union of the sparsities of all rows in that row stripe. Then, we discard the internal columns of A_{BL} from the column set of C. Note that a nonzero entry c_{kj} remains in C if coupling column j has at least one nonzero in the kth row stripe. Therefore, rows of C correspond to processors in such a way that the nonzeros in row k identify the subset of x_C-vector entries needed by processor P_k. In other words, nonzeros in column j of C identify the set of processors that need $x_C[j]$. Since the columns of C correspond to the coupling columns of A_{BL}, C has $N_C = |x_C|$ columns each of which has at least 2 nonzeros. Figure 3.2(a) illustrates communication matrix C obtained from A_{BL} shown in Fig. 3.1(a). For example, the 4th row of matrix C has nonzeros in columns 7, 12, 19, 25, and 26 corresponding to the nonzero coupling columns in the 4th row stripe of A_{BL}. So, these nonzeros summarize the need of processor P_4 for x_C-vector entries $x[7], x[12], x[19], x[25]$, and $x[26]$ in row-parallel $y \leftarrow Ax$.

Here, we exploit the *row-net* hypergraph model for sparse matrix representation [20, 21] to construct a *communication hypergraph* from matrix C. In this model, communication matrix C is represented as a hypergraph $\mathcal{H}_C = (\mathcal{V}, \mathcal{N})$

on N_C vertices and K nets. Vertex and net sets $\mathcal{V}$ and $\mathcal{N}$ correspond to the columns and rows of matrix C, respectively. There exist one vertex v_j for each column j, and one net n_k for each row k. So, vertex v_j represents $x_C[j]$, and net n_k represents processor P_k. Net n_k contains vertices corresponding to the columns that have a nonzero in row k, i.e., $v_j \in n_k$ if and only if $c_{kj} \neq 0$. $Nets(v_j)$ contains the set of nets corresponding to the rows that have a nonzero in column j. In the proposed model, each vertex v_j corresponds to the atomic task of expanding $x_C[j]$. Figure 3.2(c) shows the communication hypergraph obtained from the communication matrix C. In this figure, white and black circles represent, respectively, vertices and nets, and straight lines show the pins of nets.

3.3.1.2 Minimizing total latency and maximum volume

Here, we will show that minimizing the total latency and maintaining the balance on message-volume loads of processors can be modeled as a hypergraph partitioning problem on the communication hypergraph. Consider a K-way partition $\Pi = \{\mathcal{V}_1, \cdots, \mathcal{V}_K\}$ of communication hypergraph $\mathcal{H}_C$. Without loss of generality, we assume that part $\mathcal{V}_k$ is assigned to processor P_k for $k = 1, \ldots, K$. The consistency of the proposed model for accurate representation of the total latency requirement depends on the condition that each net n_k connects part $\mathcal{V}_k$ in Π, i.e., $\mathcal{V}_k \in \Lambda_k$. We first assume that this condition holds and discuss the appropriateness of the assumption later in §3.3.4.

Since Π is defined as a partition on the vertex set of $\mathcal{H}_C$, it induces a processor assignment for the atomic expand operations. Assigning vertex v_j to part $\mathcal{V}_\ell$ is decoded as assigning the responsibility of expanding $x_C[j]$ to processor P_ℓ. The destination set $\mathcal{E}_j$ in this expand operation is the set of processors corresponding to the nets that contain v_j except P_ℓ, i.e., $\mathcal{E}_j = Nets(v_j) - \{P_\ell\}$. If $v_j \in n_\ell$, then $|\mathcal{E}_j| = d_j - 1$, otherwise $|\mathcal{E}_j| = d_j$. That is, the message-volume requirement of expanding $x_C[j]$ will be $d_j - 1$ or d_j words in the former and latter cases. Here, we prefer to associate a weight of $d_j - 1$ with each vertex v_j because the latter case is expected to be rare in partitionings. In this way, satisfying the partitioning constraint in Eq. 2.4 $\left(\frac{W_{max}-W_{avg}}{W_{avg}} \leq \epsilon\right)$ relates to maintaining the

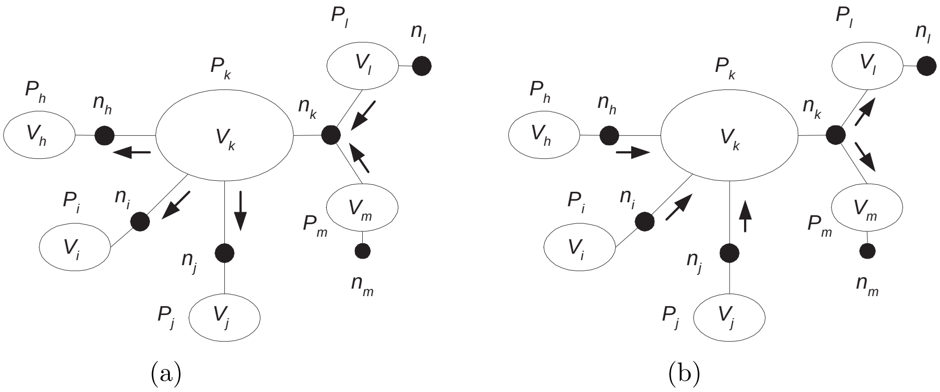


Figure 3.3: Generic communication-hypergraph partitions for showing incoming and outgoing messages of processor P_k in (a) row-parallel $y \leftarrow Ax$, and (b) column-parallel $w \leftarrow A^T z$.

balance on message-volume loads of processors. Here, the message-volume load of a processor refers to the volume of outgoing messages. We prefer to omit the incoming volume in considering the message-volume load of a processor with the assumption that each processor has enough amount of local computation that overlaps with incoming messages in the network.

Consider a net n_k with the connectivity set Λ_k in partition Π. Let $\mathcal{V}_\ell$ be a part in Λ_k other than $\mathcal{V}_k$. Also, let v_j be a vertex of net n_k in $\mathcal{V}_\ell$. Since $v_j \in \mathcal{V}_\ell$ and $v_j \in n_k$, processor P_ℓ will be sending $x_C[j]$ to processor P_k due to the associated expand assignment. A similar *send* requirement is incurred by all other vertices of net n_k in $\mathcal{V}_\ell$. That is, the vertices of net n_k that lie in $\mathcal{V}_\ell$ show that P_ℓ must gather all x_C-vector entries corresponding to vertices in $n_k \cap \mathcal{V}_\ell$ into a single message to be sent to P_k. The size of this message will be $|n_k \cap \mathcal{V}_\ell|$ words. So, a net n_k with the connectivity set Λ_k shows that P_k will be receiving a message from each processor in Λ_k except itself. Hence, a net n_k with the connectivity λ_k shows $\lambda_k - 1$ messages to be received by P_k because $\mathcal{V}_k \in \Lambda_k$ (due to the consistency condition). The sum of the connectivity-1 values of all K nets, i.e., $\sum_{n_k}(\lambda_k - 1)$, will give the total number of messages received. As the total number of incoming messages is equal to the total number of outgoing messages, minimizing the objective function in Eq. 2.3$\Big(cutsize(\Pi) = \sum_{n_i \in \mathcal{N}}(\lambda_i - 1)\Big)$ corresponds to minimizing the total message latency.

Figure 3.3(a) shows a partition of a generic communication hypergraph to clarify the above concepts. The main purpose of the figure is to show the number rather than the volume of messages, so multiple pins of a net in a part are contracted into a single pin. Arrows along the pins show the directions of the communication in the underlying expand operations. Figure 3.3(a) shows processor P_k receiving messages from processors P_ℓ and P_m because net n_k connects parts $\mathcal{V}_k$, $\mathcal{V}_\ell$, and $\mathcal{V}_m$. The figure also shows P_k sending messages to three different processors P_h, P_i, and P_j due to nets n_h, n_i, and n_j connecting part $\mathcal{V}_k$. Hence, the number of messages sent by P_k is equal to $|Nets(\mathcal{V}_k)| - 1$.

3.3.2 Column-parallel $\boldsymbol{w \leftarrow A^T z}$

Let $(A^T)_{BL}$ denote a block-structured form (see Eq. 3.1) of A^T for the given rowwise partition of A.

3.3.2.1 Communication-hypergraph model

A communication hypergraph for column-parallel $w \leftarrow A^T z$ can be obtained from $(A^T)_{BL}$ as follows. We first determine the internal and coupling *rows* to form w_C, i.e., the w-vector entries that necessitate communication. We then apply a columnwise compression, similar to that in §3.3.1.1, to obtain communication matrix C^T. Figure 3.2(b) illustrates communication matrix C^T obtained from the block structure of $(A^T)_{BL}$ shown in Fig. 3.1(b). Finally, we exploit the *column-net* hypergraph model for sparse matrix representation [20, 21] to construct a communication hypergraph from matrix C^T. The row-net and column-net hypergraph models are duals of each other. The column-net representation of a matrix is equivalent to the row-net representation of its transpose and vice versa. Therefore, the resulting communication hypergraph derived from C^T will be topologically identical to that of the row-parallel $y \leftarrow Ax$ with dual communication-requirement association. For example, the communication hypergraph shown in Fig. 3.2(c) represents communication matrix C^T as well. In this hypergraph, net n_k represents processor P_k as before. However, vertices of net n_k denote the set of w_C-vector entries for which processor P_k generates partial results. Each vertex v_j corresponds to the atomic task of folding on $w_C[j]$. Hence, $Nets(v_j)$ denote the set of processors that generate a partial result for $w_C[j]$.

3.3.2.2 Minimizing total latency and maximum volume

Consider a K-way partition $\Pi = \{\mathcal{V}_1, \cdots, \mathcal{V}_K\}$ of communication-hypergraph $\mathcal{H}_C$ with the same part-to-processor assignment and consistency condition as in §3.3.1.2. Since the vertices of $\mathcal{H}_C$ correspond to fold operations, assigning a vertex v_j to part $\mathcal{V}_\ell$ in Π is decoded as assigning the responsibility of folding on $w_C[j]$ to processor P_ℓ. Consider a net n_k with the connectivity set Λ_k. Let $\mathcal{V}_\ell$ be a part in Λ_k other than $\mathcal{V}_k$. Also, let v_j be a vertex of net n_k in $\mathcal{V}_\ell$. Since $v_j \in \mathcal{V}_\ell$ and $v_j \in n_k$, processor P_k will be sending its partial result for $w_C[j]$ to P_ℓ because of the associated fold assignment to P_ℓ. A similar *send* requirement is incurred to P_k by all other vertices of net n_k in $\mathcal{V}_\ell$. That is, the vertices of net n_k that lie in $\mathcal{V}_\ell$ show that P_k must gather all partial w_C results corresponding to vertices in $n_k \cap \mathcal{V}_\ell$ into a single message to be sent to P_ℓ. The size of this message will be $|n_k \cap \mathcal{V}_\ell|$ words. So, a net n_k with connectivity set Λ_k shows that P_k will be sending a message to each processor in Λ_k except itself. Hence, a net n_k with the connectivity λ_k shows $\lambda_k - 1$ messages to be sent by P_k, since $\mathcal{V}_k \in \Lambda_k$ (due to the consistency condition). The sum of the connectivity-1 values of all K nets, i.e., $\sum_{n_k}(\lambda_k - 1)$, will give the total number of messages to be sent. So, minimizing the objective function in Eq. 2.3 corresponds to minimizing the total message latency.

As vertices of $\mathcal{H}_C$ represent atomic fold operations, the weighted sum of vertices in a part will relate to the volume of incoming messages of the respective processor with vertex degree weighting. However, as mentioned earlier, we prefer to define the message-volume load of a processor as the volume of outgoing messages. Each vertex v_j of net n_k that lies in a part other than $\mathcal{V}_k$ incurs one word of message-volume load to processor P_k. In other words, each vertex of net n_k that lies in part $\mathcal{V}_k$ relieves P_k of sending a word. Thus, the message-volume load of P_k can be computed in terms of the vertices in part $\mathcal{V}_k$ as $|n_k| - |n_k \cap \mathcal{V}_k|$. Here, we prefer to associate unit weights with vertices so that maintaining the partitioning constraint in Eq. 2.4 corresponds to an approximate message-volume load balancing. This approximation will prove to be a reasonable one if the net sizes are close to each other.

Figure 3.3(b) shows a partition of a generic communication hypergraph to illustrate the number of messages. Arrows along the pins of nets show the directions of messages for fold operations. Figure 3.3(b) shows processor P_k sending messages to processors P_ℓ and P_m because net n_k connects parts $\mathcal{V}_k$, $\mathcal{V}_\ell$, and $\mathcal{V}_m$. Hence, the number of messages sent by P_k is equal to $\lambda_k - 1$.

3.3.3 Row-column-parallel $y \leftarrow Ax$ and $w \leftarrow A^T z$

To minimize the total message count in $y \leftarrow Ax$ and $w \leftarrow A^T z$, we use the same communication hypergraph $\mathcal{H}_C$ with different vertex weightings. As in §3.3.1.2 and §3.3.2.2, the cutsize of a partition of $\mathcal{H}_C$ quantifies the total number of messages sent both in $y \leftarrow Ax$ and $w \leftarrow A^T z$. This property is in accordance with Facts 3 and 4 given in §3.2.2. So, minimizing the objective function in Eq. 2.3 corresponds to minimizing the total message count in row-column-parallel $y \leftarrow Ax$ and $w \leftarrow A^T z$.

Vertex weighting for maintaining the message-volume balance needs special attention. If there is a synchronization point between $w \leftarrow A^T z$ and $y \leftarrow Ax$, the *multi-constraint* partitioning [64] should be adopted with two different weightings to impose a communication-volume balance in both multiply phases. If there is no synchronization point between the two multiplies (e.g., $y \leftarrow AA^T z$), we recommend to impose a balance on aggregate message-volume loads of processors by associating an aggregate weight of $(d_j - 1) + 1 = d_j$ with each vertex v_j.

3.3.4 Remarks on partitioning models

Consider a net n_k which does not satisfy the consistency condition in a partition Π of $\mathcal{H}_C$. Since $\mathcal{V}_k \notin \Lambda_k$, processor P_k will be receiving a message from each processor in Λ_k in row-parallel $y \leftarrow Ax$. Recall that P_k needs the x_C-vector entries represented by the vertices in net n_k independent of the connectivity between part $\mathcal{V}_k$ and net n_k. In a dual manner, P_k will be sending a message to each processor in Λ_k in column-parallel $w \leftarrow A^T z$. So, net n_k with the connectivity λ_k will incur λ_k incoming or outgoing messages instead of $\lambda_k - 1$ messages determined by the cutsize of Π. That is, our model undercounts the actual number of messages by one for each net dissatisfying the consistency condition. In the worst case, this deviation may be as high as K messages in total. This deficiency of the proposed model may be overcome by enforcing the consistency condition through exploiting the *partitioning with fixed vertices* feature, which exists in some of the hypergraph-partitioning tools [2, 22]. We discuss such a method in §3.4.1.

Partitioning x_C-vector entries affects the message-volume requirement determined in the first phase. The message-volume requirement induced by the partitioning in the first phase is equal to $nnz(C) - N_C$ for row-parallel $y \leftarrow Ax$. Here, $nnz(C)$ and N_C denote, respectively, the number of nonzeros and the number of columns in communication matrix C. Consider $x_C[j]$ corresponding to column j of C. Assigning $x_C[j]$ to any one of the processors corresponding to the rows of C that have a nonzero in column j will not change the message-volume requirement. However, assigning it to some other processor will increase the message-volume requirement for expanding $x_C[j]$ by one word. In a partition Π of communication hypergraph $\mathcal{H}_C$, this case corresponds to having a vertex $v_j \in \mathcal{V}_k$ while $v_j \notin n_k$. In other words, processor P_k holds and expands $x_C[j]$ although it does not need it for local computations. A dual discussion holds for column-parallel $w \leftarrow A^T z$, where such a vertex-to-part assignment corresponds to assigning the responsibility of folding on a particular w_C-vector entry to a processor which does not generate partial result for that entry. In the worst case, the increase in the message-volume may be as high as N_C words in total for both types of multiplies. In hypergraph-theoretic view, the total message volume will be in between $\sum_k |n_k| - |\mathcal{V}|$ and $\sum_k |n_k|$, where $\sum_k |n_k| = nnz(C)$ and $|\mathcal{V}| = N_C$.

The proposed communication-hypergraph partitioning models exactly encode the total number of messages and the maximum message volume per processor metrics into the hypergraph partitioning objective and constraint, respectively, under the above conditions. The models do not directly encapsulate the metric of maximum number of messages per processor, however, it is possible to address this metric within the partitioning framework. We give a method in §3.4.3 to address this issue.

The allowed imbalance ratio (ϵ) is an important parameter in the proposed models. Choosing a large value for ϵ relaxes the partitioning constraint. Thus, large ϵ values enable the associated partitioning methods to achieve better partitioning objectives through enlarging the feasible search space. Hence, large ϵ values favor the total message-count metric. On the other hand, small ϵ values favor the maximum message-volume metric by imposing a tighter constraint on the part weights. Thus, ϵ should be chosen according to the target machine architecture and problem characteristics to trade the total latency for the maximum volume.

3.3.5 Illustration on the sample matrix

Figure 3.2(c) displays a 4-way partition of the communication hypergraph, where closed dashed curves denote parts. Nets and their associated parts are kept close to each other for a better appearance. Note that the consistency condition is satisfied for the given partition. In the figure, net n_2 with the connectivity set $\Lambda_2 = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$ shows processor P_2 receiving messages from processors P_1 and P_3 in row-parallel $y \leftarrow Ax$. In a dual manner, net n_2 shows P_2 sending messages to P_1 and P_3 in column-parallel $w \leftarrow A^T z$. Since the connectivities of nets n_1, n_2, n_3, and n_4 are, respectively, 2, 3, 3, and 2, the total message count is equal to $(2-1)+(3-1)+(3-1)+(2-1)=6$ in both types of multiplies. So, the proposed approach reduces the number of messages from 9 (see §3.2.2) to 6

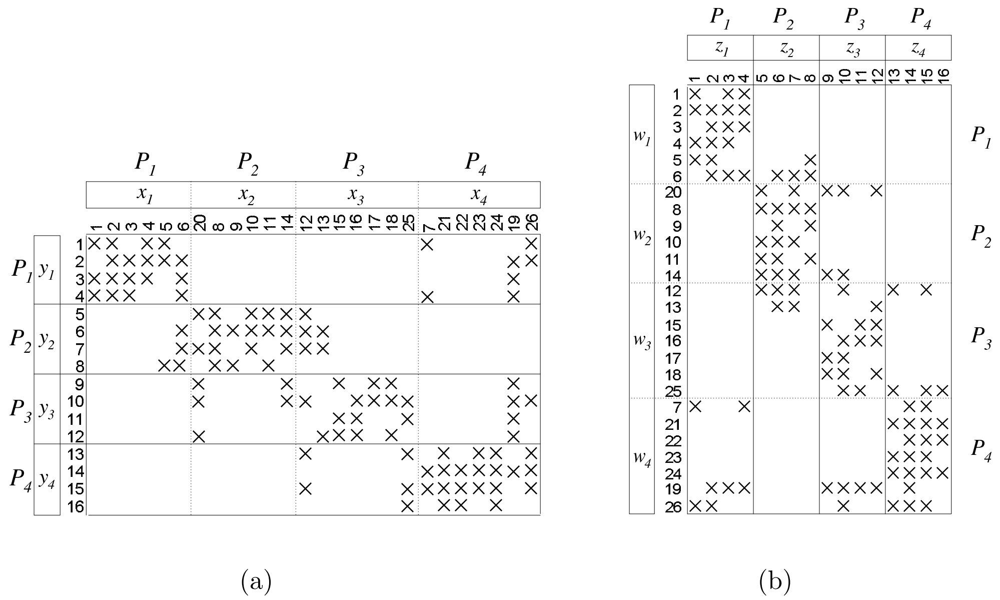


Figure 3.4: Final 4×4 block structures: (a) A_{BL} for row-parallel $y \leftarrow Ax$, and (b) $(A^T)_{BL}$ for column-parallel $w \leftarrow A^T z$, induced by 4-way communication-hypergraph partition in Fig. 3.2(c).

by yielding the given partition of x_C-vector (w_C-vector) entries.

In the proposed two-phase approach, partitioning x_C-vector entries in the second phase can also be regarded as re-permuting coupling columns of A_{BL} obtained in the first phase. In a dual manner, partitioning w_C-vector entries can be regarded as re-permuting coupling rows of $(A^T)_{BL}$. Figure 3.4 shows the re-permuted A_{BL} and $(A^T)_{BL}$ matrices induced by the sample communication-hypergraph partition shown in Fig. 3.2(c). The total message count is 6 as enumerated by the total number of nonzero off-diagonal blocks according to Fact 4 thus matching the cutsize of the partition given in Fig. 3.2(c).

As seen in Fig 3.2(c), each vertex in each part is a pin of the net associated with that part. So, for both types of multiplies, the sample partitioning does not increase the total message volume and it remains at its lower bound which is $\sum_k |n_k| - |\mathcal{V}| = (5+6+7+5) - 10 = 13$ words. This value can also be verified from the re-permuted matrices given in Fig. 3.4 by enumerating the total number of nonzero columns in the off-diagonal blocks according to Fact 4.

For row-parallel $y \leftarrow Ax$, the message-volume load estimates of processors are 2, 2, 4, and 5 words according to the vertex weighting proposed in §3.3.1.2. These estimates are expected to be exact since each vertex in each part is a pin of the net associated with that part. This expectation can be verified from the re-permuted A_{BL} matrix given in Fig. 3.4(a) by counting the number of nonzero columns in the off-diagonal blocks of the virtual column stripes according to Fact 1.

For column-parallel $w \leftarrow A^T z$, the message-volume load estimates of processors are 2, 2, 3, and 3 words according to the unit vertex weighting proposed in §3.3.2.2. However, the actual message-volume loads of processors are 3, 4, 4, and 2 words. These values can be obtained from Fig. 3.4(b) by counting the number of nonzero rows in the off-diagonal blocks of the virtual row stripes according to Fact 2. The above values yield an estimated imbalance ratio of 20% and an actual imbalance ratio of 23%. The discrepancy between the actual and estimated imbalance ratios is because of the differences in net sizes.

3.4 Algorithms for communication-hypergraph partitioning

We present three methods for partitioning communication hypergraphs. Method *PaToH-fix* is presented to show the feasibility of using a publicly available tool to partition communication hypergraphs. Method *MSN* involves some tailoring towards partitioning communication hypergraphs. Method *MSNmax* tries to incorporate the minimization of the maximum message count per processor into the *MSN* method. In these three methods, minimizing the cutsize while maintaining the partitioning constraint corresponds to minimizing the total number of messages while maintaining the balance on communication-volume loads of processors according to the models proposed in §3.3.1.2 and §3.3.2.2.

3.4.1 PaToH-fix: Recursive bipartitioning with fixed vertices

We use the multilevel hypergraph-partitioning tool PaToH [22] for partitioning communication hypergraphs. Recall that the communication-hypergraph partitioning differs from the conventional hypergraph partitioning because of the net-to-part association needed to satisfy the consistency condition mentioned in §3.3.1.2 and §3.3.2.2. We exploit the *partitioning with fixed vertices* feature supported by PaToH to achieve this net-to-part association as follows. The communication hypergraph is augmented with K zero-weighted artificial vertices of degree one. Each artificial vertex v_k^* is added to a unique net n_k as a new pin and marked as fixed to part $\mathcal{V}_k$. This augmented hypergraph is fed to PaToH for K-way partitioning. PaToH generates K-way partitions with these K labeled vertices lying in their fixed parts thus establishing the required net-to-part association. A K-way partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ generated by PaToH is decoded as follows. The atomic communication tasks associated with the actual vertices assigned to part $\mathcal{V}_k$ are assigned to processor P_k, whereas v_k^* does not incur any communication task.

3.4.2 MSN: Direct *K*-way partitioning

Most of the partitioning tools, including PaToH, achieve K-way partitioning through recursive bisection. In this scheme, first a 2-way partition is obtained, then this 2-way partition is further bipartitioned recursively. The connectivity-1 cutsize metric (see Eq. 2.3) is easily handled through net splitting [21] during recursive bisection steps. Although the recursive-bisection paradigm is successful in K-way partitioning in general, its performance degrades for hypergraphs with large net sizes. Since communication hypergraphs have nets with large sizes, this degradation is also expected to be notable with *PaToH-fix*. In order to alleviate this problem, we have developed a multilevel direct K-way hypergraph partitioner (MSN) by integrating Sanchis's direct K-way refinement (SN) algorithm [88] to the uncoarsening step of the multilevel framework.

The coarsening step of *MSN* is essentially the same as that of PaToH. In the initial partitioning step, a K-way partition on the coarsest hypergraph is obtained by using a simple constructive approach which mainly aims to satisfy the balance constraint. In *MSN*, the net-to-part association is handled implicitly rather than by introducing artificial vertices. This association is established in the initial partitioning step through associating each part with a distinct net which connects that part, and it is maintained later in the uncoarsening step. In the uncoarsening step, the SN algorithm, which is a generalization of the two-way FM paradigm to K-way refinement [31, 89], is used. SN, starting from a K-way initial partition, performs a number of passes until it finds a locally optimum partition, where each pass consists of a sequence of vertex moves. The fundamental idea is the notion of *gain*, which is the decrease in the cutsize of a partition due to a vertex moving from a part to another. The local search strategy adopted in the SN approach repeatedly moves a vertex with the maximum gain even if that gain is negative, and records the best partition encountered during a pass. Allowing tentative moves with negative gains brings restricted hill-climbing ability to the approach.

In the SN algorithm, there are $K-1$ possible moves for each vertex. The algorithm stores the moves from a source part in $K-1$ associated priority queues—one for each possible destination part. So, the algorithm uses $K(K-1)$ priority queues with a space complexity of $O(N_C K)$, which may become a memory problem for large K. The moves with the maximum gain are selected from each of these $K(K-1)$ priority queues and the one that maintains the balance criteria is performed. After the move, only the move gains of the vertices that share a net with the moved vertex may need to be updated. This may lead to updates on at most $4K-6$ priority queues. Within a pass, a vertex is allowed to move at most once.

3.4.3 MSNmax: Considering the maximum message latency

The proposed models do not encapsulate the minimization of the maximum message latency per processor. By similar reasoning in defining the message-volume load of a processor as the volume of outgoing messages, we prefer to define the message-latency load of a processor in terms of the number of outgoing messages. Here, we propose a practical way of incorporating the minimization of the maximum message-count metric into the *MSN* method. The resulting method is referred to here as *MSNmax*. *MSNmax* differs from *MSN* only in the SN refinement scheme used in the uncoarsening phase. *MSNmax* still relies on the same gain notion and maintains updated move gains in $K(K-1)$ priority queues. The difference lies in the move selection policy, which favors the moves that reduce the message counts of overloaded processors. Here, a processor is said to be overloaded if its message count is above the average by a prescribed percentage (e.g., we used 25%). For this purpose, message counts of processors are maintained during the course of the SN refinement algorithm.

For row-parallel $y \leftarrow Ax$, the message count of a processor can be reduced by moving vertices out of the associated part. Recall that moving a vertex from a part corresponds to relieving the associated processor from the respective atomic expand task. So, only the priority queues of the overloaded parts are considered for selecting the move with the maximum gain. For column-parallel $w \leftarrow A^T z$, the message count of a processor P_k can be reduced by reducing the connectivity of the associated net n_k through moves from the parts in $\Lambda_k - \{P_k\}$. So, only the priority queues of the parts that are in the connectivity sets of the nets associated with the overloaded parts are considered. For both types of parallel multiplies, moves selected from the restricted set of priority queues are likely to decrease the message counts of overloaded processors besides decreasing the total message count.

Table 3.1: Properties of unsymmetric square and rectangular test matrices.

$M \times N$ matrix A				$K \times N_C$ communication matrix C					
				$K = 24$		$K = 64$		$K = 128$	
Name	M	N	NNZ	N_C	NNZ	N_C	NNZ	N_C	NNZ
lhr14	14270	14270	321988	11174	25188	12966	31799	13508	36039
lhr17	17576	17576	399500	13144	29416	16070	38571	16764	46182
onetone1	36057	36057	368055	8137	20431	11458	30976	13911	39936
onetone2	36057	36057	254595	3720	9155	6463	16259	11407	27264
pig-large	28254	17264	75018	1265	3347	1522	4803	1735	6193
pig-very	174193	105882	463303	4986	12015	6466	16185	7632	20121
CO9	10789	14851	101578	4458	9226	7431	21816	7887	25070
fxm4-6	22400	30732	248989	769	1650	2010	4208	4223	8924
kent	31300	16620	184710	5200	10691	11540	28832	14852	49976
mod2	34774	31728	165129	4760	9870	8634	18876	10972	24095
pltexpA4	26894	70364	143059	1961	4218	3259	7858	5035	13397
world	34506	32734	164470	5116	10405	9569	20570	13610	30881

3.5 Experiments

We have tested the performance of the proposed models and associated partitioning methods on a wide range of large unsymmetric square and rectangular sparse matrices. Properties of these matrices are listed in Table 3.1. The first four matrices, which are obtained from University of Florida Sparse Matrix Collection [32], are from the unsymmetric linear system application. The `pig-large` and `pig-very` matrices [48] are from the least squares problem. The remaining six matrices, which are obtained from Hungarian Academy of Sciences OR Lab[1], are from miscellaneous and stochastic linear programming problems. In this table, the *NNZ* column lists the number of nonzeros of the matrices.

We have tested $K = 24, 64$, and 128-way rowwise partitionings of each test matrix. For each K value, K-way partitioning of a test matrix forms a partitioning instance. Recall that the objective in the first phase of our two-phase approach is minimizing the total message volume while maintaining the computational-load balance. This objective is achieved by exploiting the recently proposed computational-hypergraph model [21]. The hypergraph-partitioning

[1]ftp://ftp.sztaki.hu/pub/oplab

tool PaToH [22] was used with default parameters to obtain K-way rowwise partitions. The computational-load imbalance values of all partitions were measured to be below 6 percent.

For the second phase, communication matrix C was constructed for every partitioning instance as described in §3.3.1.1 and §3.3.2.1. Table 3.1 displays properties of these communication matrices. Then, the communication hypergraph was constructed from each communication matrix as described in §3.3.1.1 and §3.3.2.1. Note that communication-matrix properties listed in Table 3.1 also show communication-hypergraph properties. That is, for each K value, the table effectively shows a communication hypergraph on K nets, N_C vertices, and NNZ pins.

The communication hypergraphs are partitioned using the proposed methods discussed in §3.4. In order to verify the validity of the communication hypergraph model, we compare the performance of these methods with a method called *Naive*. This method mimics the current state of the art by minimizing the communication overhead due to the message volume without spending any explicit effort towards minimizing the total message count. The *Naive* method tries to obtain a balance on the message-volume loads of processors while attaining the total message-volume requirement determined by the partitioning in the first phase. The method adopts a constructive approach which is similar to the best-fit-decreasing heuristic used in solving the NP-hard K-feasible bin packing problem [58]. Vertices of the communication hypergraph are assigned to parts in the decreasing order of vertex weights. Each vertex v_j is allowed to be assigned only to the parts in $Nets(v_j)$ to avoid increases in the message volume. Here, the best-fit criterion corresponds to assigning v_j to a part in $Nets(v_j)$ with the minimum weight thus trying to obtain a balance on the message-volume loads.

The partitioning methods, *PaToH-fix*, *MSN*, and *MSNmax* incorporate randomized algorithms. Therefore, they were run 20 times starting from different random seeds for K-way partitioning of every communication hypergraph. Randomization in the *Naive* method were realized by random permutation of the vertices before sorting. Averages of the resulting communication patterns of these

Table 3.2: Performance of the methods with varying imbalance ratios in 64-way partitionings.

Matrix	Partition Method	Total msg ϵ=0.1	ϵ=0.3	ϵ=0.5	ϵ=1.0	Max vol ϵ=0.1	ϵ=0.3	ϵ=0.5	ϵ=1.0
lhr17	Naive	1412	—	—	—	373	—	—	—
	PaToHfix	817	726	724	700	643	755	858	1042
	MSN	745	662	625	592	678	793	895	1177
	MSNmax	731	684	649	638	676	799	920	1119
pig-very	Naive	2241	—	—	—	161	—	—	—
	PaToHfix	1333	1176	1151	1097	272	316	361	448
	MSN	1407	1199	1137	1019	284	343	398	526
	MSNmax	1293	1142	1040	967	298	354	411	530
fxm4-6	Naive	—	—	—	312	—	—	—	67
	PaToHfix	212	193	193	188	70	75	81	105
	MSN	244	205	199	172	72	83	96	114
	MSNmax	247	213	208	165	70	85	94	103

runs are displayed in the following tables. In these tables, the *Total msg* and *Total vol* columns list, respectively, the total number and total volume of messages sent. The *Max msg* and *Max vol* columns list, respectively, the maximum number and maximum volume of messages sent by a single processor.

The following parameters and options are used in the proposed partitioning methods. *PaToH-fix* were run with the coarsening option of absorption clustering using pins (`ABS_HPC`), and the refinement option of Fiduccia-Mattheyses (`FM`). The scaled heavy-connectivity matching (`SHCM`) of PaToH was used in the coarsening step of the multilevel partitioning methods *MSN* and *MSNmax*. `ABS_HPC` is the default coarsening option in *PaToH-fix*. It is a quite powerful coarsening method that absorbs nets into supervertices, which helps FM-based recursive-bisection heuristics. However, we do not want nets being absorbed in *MSN* and *MSNmax* to be able to establish net-to-part association in the initial partitioning phase. So, `SHCM`, which does not aim to absorb nets, was selected.

Table 3.2 shows performance of the proposed methods with varying ϵ in 64-way partitioning of three matrices each of which is the largest (in terms of the number of nonzeros) in its application domain. The performance variation is displayed in terms of the total message-count and maximum message-volume metrics because these two metrics are exactly encoded in the proposed models. Recall that *Naive* is a constructive method and its performance does not depend on ϵ. So, the performance values for *Naive* are listed under the columns corresponding to the attained imbalance ratios. As seen in Table 3.2, by relaxing ϵ, each method can find partitions with smaller total message counts and larger maximum message-volume values. It is also observed that imbalance values of the partitions obtained by all of the proposed methods are usually very close to the given ϵ. These outcomes are in accordance with the discussion in § 3.3.4. As seen in the table, all of the proposed methods perform significantly better than the *Naive* method even with the tightest constraint of $\epsilon = 0.1$. However, the detailed performance results are displayed for $\epsilon = 1.0$ (i.e., $W_{max} \leq 2W_{avg}$ in Eq. 2.4) in the following tables. We chose such a relaxed partitioning constraint in order to discriminate among the proposed methods. It should be noted here that imbalance ratios for the message-volume loads of processors might be greater than the chosen ϵ value because of the approximation in the proposed vertex weighting scheme. For example, with $\epsilon = 1.0$, the methods *PaToH-fix*, *MSN*, and *MSNmax* produce partitions with actual imbalance ratios of 0.94, 1.26, and 1.35 for matrix `lhr17`, respectively.

Table 3.3 displays the communication patterns for $K = 64$- and 128-way partitions in row-parallel $y \leftarrow Ax$. The bottom of the table shows the average performance of the proposed methods compared with the *Naive* method. These values are obtained by first normalizing the performance results of the proposed methods with respect to those of the *Naive* method for every partitioning instance and then averaging these normalized values over the individual methods.

In terms of the total message-volume metric, *Naive* achieves the lowest values as seen in Table 3.3. This is expected since *Naive* attains the total message volume determined by the partitioning in the first phase. The increase in the total message-volume values for the proposed methods remain below 66% for all partitioning instances. As seen in the bottom of the table, these increases are below 41% on the average. Note that the total message-volume values for *Naive* are equal to the differences of the *NNZ* and N_C values of the respective communication matrix (see Table 3.1). Also note that the *NNZ* values of the

Table 3.3: Communication patterns for K-way row-parallel $y \leftarrow Ax$.

Matrix	Part.	$K = 64$				$K = 128$			
		Total		Max		Total		Max	
	method	msg	vol	msg	vol	msg	vol	msg	vol
lhr14	Naive	1318	18833	43.9	308	2900	22531	47.6	204
	PaToH-fix	676	28313	34.0	813	1417	32661	47.8	627
	MSN	561	26842	24.4	975	1247	30796	31.6	577
	MSNmax	640	24475	19.2	897	1348	28758	22.7	535
lhr17	Naive	1412	22501	45.6	373	3675	29418	58.9	265
	PaToH-fix	700	34515	36.5	1042	1867	42623	54.3	750
	MSN	592	32530	26.3	1177	1453	40009	34.0	736
	MSNmax	638	31149	22.0	1119	1599	38557	26.8	689
onetone1	Naive	1651	19518	39.9	332	4112	26025	47.4	231
	PaToH-fix	663	26789	27.2	714	1639	35741	39.1	580
	MSN	545	27109	24.1	1008	1384	35129	31.1	688
	MSNmax	610	24012	20.9	950	1507	31345	26.4	642
onetone2	Naive	995	9796	30.4	186	2049	15857	28.6	139
	PaToH-fix	429	12940	17.8	381	804	20983	25.1	423
	MSN	406	13236	17.1	510	787	20649	22.1	422
	MSNmax	420	12389	15.1	485	807	18850	20.4	381
pig-large	Naive	1220	3281	39.4	60	2723	4458	39.6	47
	PaToH-fix	759	4363	40.5	144	1764	5805	52.5	142
	MSN	619	4108	34.5	153	1551	5752	43.0	115
	MSNmax	682	3812	35.6	138	1678	5185	35.0	100
pig-very	Naive	2241	9719	56.5	161	4574	12489	78.7	117
	PaToH-fix	1097	14725	59.8	448	2533	18567	97.8	398
	MSN	1019	14349	54.5	526	2389	17317	77.3	320
	MSNmax	967	14008	55.4	530	2501	15729	80.5	317
CO9	Naive	1283	14385	41.0	369	1645	17183	48.9	289
	PaToH-fix	622	19221	34.6	567	1191	23575	35.8	434
	MSN	521	18352	27.1	687	904	20727	28.9	412
	MSNmax	513	17736	23.1	684	800	21281	25.6	492
fxm4-6	Naive	312	2198	13.6	67	562	4701	15.9	64
	PaToH-fix	188	2856	11.8	105	361	5746	13.8	129
	MSN	172	2746	10.1	114	338	5647	12.2	129
	MSNmax	165	2543	8.9	103	322	5386	11.7	124
kent	Naive	342	17292	14.1	547	1020	35124	21.9	602
	PaToH-fix	235	21200	9.2	621	740	42328	15.8	631
	MSN	190	21539	8.9	905	596	39774	19.6	866
	MSNmax	201	19666	7.0	773	614	40012	13.0	830
mod2	Naive	376	10242	22.4	366	811	13123	33.8	240
	PaToH-fix	294	16683	19.8	606	658	21409	22.6	431
	MSN	254	13353	15.2	604	575	17329	18.7	391
	MSNmax	231	14400	12.5	639	548	19009	14.4	408
pltexpA4	Naive	507	4599	21.9	116	1013	8362	25.6	99
	PaToH-fix	257	5553	17.7	243	579	10163	22.8	208
	MSN	245	5828	15.3	241	556	9705	21.7	213
	MSNmax	264	5321	13.2	214	546	9582	19.4	206
world	Naive	534	11001	27.4	387	1785	17271	44.1	222
	PaToH-fix	362	18355	21.2	603	1036	26514	35.6	488
	MSN	315	14765	16.8	595	902	23927	24.8	476
	MSNmax	287	16243	14.8	680	886	23762	20.6	476
		Normalized averages over Naive							
	Naive	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	PaToH-fix	0.56	1.41	0.81	2.03	0.59	1.38	0.91	2.38
	MSN	0.48	1.34	0.68	2.37	0.51	1.29	0.75	2.30
	MSNmax	0.49	1.28	0.60	2.26	0.52	1.24	0.63	2.21

communication matrices listed in Table 3.1 show the upper bounds on the total message-volume values for the proposed partitioning methods.

In terms of the maximum message-volume metric, the proposed partitioning methods yield worse results than the *Naive* method by a factor between 2.0 and 2.4 on the average as seen in the bottom of Table 3.3. This performance difference stems from three factors. First, *Naive* is likely to achieve small maximum message-volume values since it achieves the lowest total message-volume values. Second, the best-fit-decreasing heuristic adopted in *Naive* is an explicit effort towards achieving a balance on the message volume. Third, the relaxed partitioning constraint ($\epsilon = 1.0$) used in the proposed partitioning methods leads to higher imbalance ratios among the message-volume loads of processors.

In terms of the total message-count metric, all of the proposed methods yield significantly better results than the *Naive* method in all partitioning instances. They reduce the total message count by a factor between 1.3 and 3.0 in 64-way, and between 1.2 and 2.9 in 128-way partitionings. As seen in the bottom of Table 3.3, the reduction factor is approximately 2 on the average. Comparing the performance of the proposed methods, both *MSN* and *MSNmax* perform better than *PaToH-fix* in all partitioning instances, except 64-way partitioning of `plexpA_4` and 128-way partitioning of `onetone2`, leading to a considerable performance difference on the average. This experimental finding confirms the superiority of the direct K-way partitioning approach over recursive-bisection approach. There is no clear winner between *MSN* and *MSNmax*. *MSN* performs better than *MSNmax* in 14 out of 24 partitioning instances, leading to a slight performance difference on the average.

In terms of the maximum message-count metric, all of the proposed methods again yield considerably better results than the *Naive* method in all instances, except 64- and 128-way partitionings of `pig` matrices. However, the performance difference between the proposed methods and the *Naive* method is not as large as that in the total message-count metric. Comparing the performance of the proposed methods, both *MSN* and *MSNmax* perform better than *PaToH-fix* in all partitioning instances, except 128-way partitioning of `kent`, leading to a considerable performance difference on the average. *MSNmax* is the clear winner in the maximum message-count metric as expected. As seen in the bottom of the table, *MSNmax* yields, respectively, 40% and 37% less maximum message counts than *Naive*, for 64 and 128-way partitionings, on the average.

We have also experimented with the performance of the proposed methods for 64-way and 128-way partitionings for column-parallel $w \leftarrow A^T z$ and row-column-parallel $y \leftarrow AA^T z$ on the test matrices. Since very similar relative performance results were obtained in these experiments, we omit presentation and discussion of these experimental results due to the lack of space.

It is important to see whether the theoretical improvements obtained by our methods in the given performance metrics hold in practice. For this purpose, we have implemented row-parallel $y \leftarrow Ax$ and row-column-parallel $y \leftarrow AA^T z$ multiplies using the LAM/MPI 6.5.6 [18] message passing library. The parallel multiply programs were run on a Beowulf class [94] PC cluster with 24 compute nodes. Each node has a 400Mhz Pentium-II processor and 128MB memory. The interconnection network is comprised of a 3COM SuperStack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node. The system runs the Linux kernel 2.4.14 and the Debian GNU/Linux 3.0 distribution.

Within the current experimental framework, *MSNmax* seems to be the best choice for communication-hypergraph partitioning. So, parallel running times of the multiply programs are listed in Table 3.4 only for *MSNmax* partitioning results in comparison with those of the *Naive* method. Communication patterns for the resulting partitions are also listed in the table in order to show how improvements in performance metrics relate to improvements in parallel running times.

As seen in Table 3.4, the partitions obtained by *MSNmax* lead to considerable improvements in parallel running times compared with those of *Naive* for all matrices. The improvements in parallel running times are in between 4% and 40% in $y \leftarrow Ax$, and between 5% and 31% in $y \leftarrow AA^T z$. In row-parallel $y \leftarrow Ax$, the lowest percent improvement of 4% occurs for matrix `kent` despite

Table 3.4: Communication patterns and parallel running times in msecs for 24-way row-parallel $y \leftarrow Ax$ and row-column-parallel $y \leftarrow AA^T z$.

		$y \leftarrow Ax$					$y \leftarrow AA^T z$				
Matrix	Part.	Total		Max		Parl	Total		Max		Parl
	method	msg	vol	msg	vol	time	msg	vol	msg	vol	time
lhr14	Naive	414	14014	23	603	2.57	838	28028	46	1177	5.07
	MSNmax	176	19580	12	1601	1.90	342	42456	27	1960	3.95
lhr17	Naive	393	16272	22	691	2.79	792	32544	45	1159	5.71
	MSNmax	168	24510	17	2229	2.20	334	48554	23	2112	4.38
onetone1	Naive	362	12294	19	546	2.52	728	24588	41	788	5.49
	MSNmax	152	15153	16	1403	1.85	262	34304	24	1586	4.37
onetone2	Naive	205	5435	12	297	1.60	412	10870	24	419	3.24
	MSNmax	102	6294	9	690	1.31	186	15234	16	715	2.44
pig-large	Naive	325	2082	23	108	2.06	650	4164	42	162	3.41
	MSNmax	151	2872	20	276	1.28	312	5554	26	271	2.35
pig-very	Naive	497	7029	23	354	3.51	994	14058	46	456	7.33
	MSNmax	228	10214	23	937	2.74	428	20538	29	963	5.95
CO9	Naive	122	4768	11	437	1.74	244	9536	22	1184	3.34
	MSNmax	68	6834	9	750	1.35	152	13700	16	1430	2.99
fxm4-6	Naive	113	881	11	44	1.57	226	1762	27	108	3.18
	MSNmax	58	1005	7	96	0.95	120	2038	15	124	2.31
kent	Naive	57	5491	5	488	1.12	114	10982	9	972	2.27
	MSNmax	41	5783	5	541	1.08	86	12596	7	1025	2.12
mod2	Naive	79	5110	11	617	1.74	158	10220	22	1586	3.67
	MSNmax	59	7764	7	779	1.53	130	15890	14	2148	3.50
pltexpA4	Naive	106	2257	9	146	1.25	212	4514	20	225	2.46
	MSNmax	60	2543	8	256	0.93	120	5410	14	314	2.08
world	Naive	79	5289	9	667	1.89	158	10578	19	2204	3.73
	MSNmax	65	8316	7	836	1.66	134	13638	16	2442	3.38
Normalized averages over Naive											
	Naive	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	MSNmax	0.55	1.33	0.79	2.10	0.78	0.56	1.37	0.65	1.52	0.82

the modest improvement of 28% achieved by *MSNmax* over *Naive* in total message count. The reason seems to be the equal maximum message counts obtained by these partitioning methods. The highest percent improvement of 40% occurs for matrix `fxm4-6` for which *MSNmax* achieves significant improvements of 49% and 36% in the total and maximum message counts, respectively. However, the higher percent improvements obtained by *MSNmax* for matrix `lhr14` in message-count metrics do not lead to higher percent improvements in parallel running time. This might be attributed to *MSNmax* achieving lower percent improvements for `lhr14` in message-volume metrics compared with those for `fxm4-6`. These experimental findings confirm the difficulty of the target problem.

Table 3.5 displays partitioning times for the three largest matrices selected from different application domains. The *Phase 1 time* and *Phase 2 time* columns list, respectively, the computational-hypergraph and communication-hypergraph partitioning times. Sequential matrix-vector multiply times are also displayed to show the relative preprocessing overhead introduced by the partitioning methods. All communication-hypergraph partitionings take significantly less time than computational-hypergraph partitionings except partitioning communication hypergraph of `lhr17` with *PaToH-fix*. As expected, the communication hypergraphs are smaller than the respective computational hypergraphs. However, some communication hypergraphs might have very large net sizes because of the small number of nets. Matrix `lhr17` is an example of such a case with the large average net size of $nnz(C)/K = 1225$ in the communication hypergraph versus the small average net size of $nnz(A)/N = 22$ in the computational hypergraph. This explains the above exceptional experimental outcome because running times of matching heuristics, used in the coarsening step of PaToH, increase with the sum of squares of net sizes [21] (see also Theorem 5.5 in [52]).

Comparing the running times of communication-hypergraph partitioning methods, *Naive* takes an insignificant amount of time as seen in Table 3.5. Direct K-way partitioning approaches are expected to be faster than the recursive-bisection based *PaToH-fix* because of the single coarsening step as compared

Table 3.5: 24-way partitioning and sequential matrix-vector multiply times in msecs.

Matrix	Partitioning times				Seq. y=Ax time
	Phase 1		Phase 2		
	Method	Time	Method	Time	
lhr17	PaToH	6100	Naive	32	19.56
			PaToH-fix	13084	
			MSN	3988	
			MSNmax	3885	
pig-very	PaToH	20960	Naive	12	30.37
			PaToH-fix	2281	
			MSN	1086	
			MSNmax	1022	
fxm4-6	PaToH	2950	Naive	2	13.19
			PaToH-fix	58	
			MSN	112	
			MSNmax	81	

with $K - 1 = 23$ coarsening steps. As expected, *MSN* and *MSNmax* take considerably less time than *PaToH-fix* except in partitioning communication-hypergraph of `fxm4-6`, which has a moderate average net size. As seen in the table, the second-phase methods *MSN* and *MSNmax* introduce much less preprocessing overhead than the first phase. The partitionings obtained by *MSNmax* for `lhr17`, `pig-very`, and `fxm4-6` matrices lead to speedup values of 8.89, 11.1, and 13.9, respectively, in row-parallel matrix-vector multiply on our 24-processor PC cluster.

Chapter 4

Communication cost metrics for 2D SpMxV

In the previous chapter, we showed how to encapsulate the minimization of the total volume, the total message count, the maximum volume and the maximum message count handled by a single processor in 1D partitioning of sparse matrices. The work in the previous chapter addressed unsymmetric partitionings, i.e., the partitions on the input and output vectors were different. In this chapter, we adopt the methods proposed in the previous chapter to address the minimization of aforementioned four communication cost metrics in 2D partitioning of sparse matrices. The work presented here enables generation of symmetric partitionings as well as unsymmetric partitionings.

We show a two-phase approach for minimizing various communication-cost metrics in fine-grain partitioning of sparse matrices for parallel processing. In the first phase, we obtain a partitioning with the existing tools on the matrix to determine computational loads of the processor. In the second phase, we try to minimize the communication-cost metrics. For this purpose, we develop communication-hypergraph partitioning models. We experimentally evaluate the contributions on a PC cluster.

4.1 Preliminaries

In the fine-grain hypergraph model of Çatalyürek and Aykanat [23], an $m \times n$ matrix A with Z nonzeros is represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ with $|\mathcal{V}| = Z$ and $|\mathcal{N}| = m + n$ for 2D partitioning. There exists one vertex v_{ij} for each nonzero a_{ij}. There exists one net r_i for each row i and one net c_j for each column j. Each row-net r_i and column-net c_j contain all vertices v_{i*} and v_{*j}, respectively. Each vertex v_{ij} corresponds to scalar multiplication $a_{ij}x_j$. Hence, the computational weight associated with a vertex is 1. Each row-net r_i represents the dependency of y_i on the scalar multiplications with a_{i*}'s. Each column-net c_j represents the dependency of scalar multiplications with a_{*j}'s on x_j. With this model, the problem of 2D partitioning a matrix among K processors can be modeled as the K-way hypergraph partitioning problem. In this model, minimizing the cutsize while maintaining balance on the part weights corresponds to minimizing the total communication volume and maintaining balance on the computational loads of the processors. An external column-net represents the communication volume requirement on a x-vector entry. This communication occurs in *expand phase*, just before the scalar multiplications. An external row-net represents the communication volume requirement on a y-vector entry. This communication occurs in *fold phase*, just after the scalar multiplications. Çatalyürek and Aykanat assign the responsibility of expanding x_i and folding y_i to the processor that holds a_{ii} to obtain symmetric partitioning. Note that for the unsymmetric partitioning case, one can assign x_i to any processor holding a nonzero in column i without any additional overhead. A similar opportunity exists for y_i. In the symmetric partitioning case, however, x_i and y_i may be assigned to a processor holding nonzeros both in the row and column i. In this chapter, we try to exploit the freedom in assigning vector elements to address the four communication-cost metrics in fine-grain partitioning of sparse matrices.

A 10×10 matrix with 37 nonzeros and its 4-way fine-grain partitioning is given in Fig. 4.1(a). In the figure, the partitioning is given by the processor numbers for each nonzero. The computational load balance is achieved by assigning 9, 10, 9, and 9 nonzeros to processors in order.

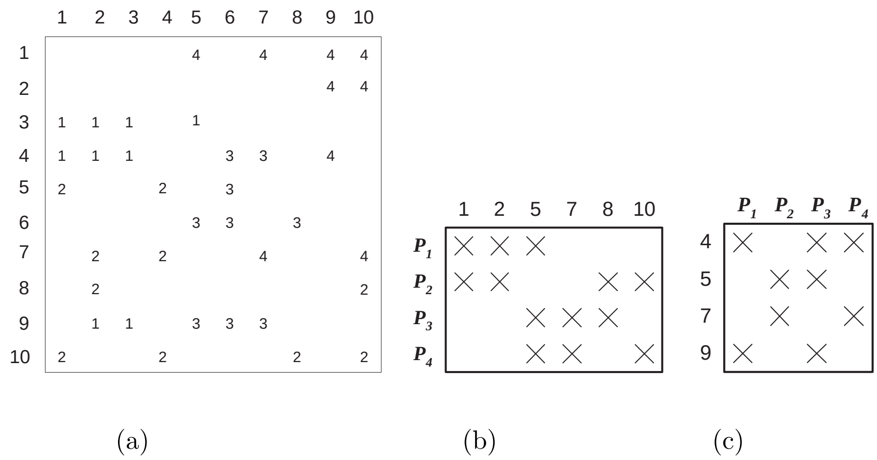


Figure 4.1: (a) A 10×10 matrix and a 4-way partitioning, (b) communication matrix C_x, and (c) communication matrix C_y.

4.2 Minimizing the total number of messages

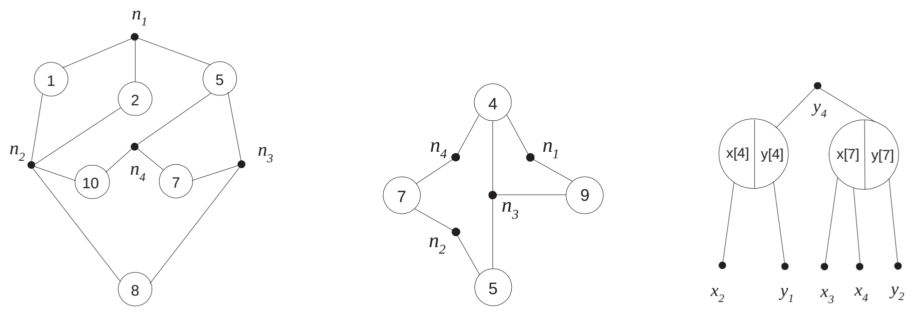
Given a K-way fine-grain partitioning of a matrix, we identify two sets of rows and columns; *internal and coupling*. The internal rows or columns have nonzeros only in one part. The coupling rows or columns have nonzeros in more than one part. The set of x-vector entries that are associated with the coupling columns, referred to here as x_C, necessitate communication. Similarly, the set of y-vector entries that are associated with the coupling rows, referred to here as y_C, necessitate communication. Note that when symmetric partitioning requirement arises, we add to x_C those x-vector entries whose corresponding entries are in y_C and vice versa. The proposed approach considers partitioning of these x_C and y_C vector entries to reduce the total message count and the maximum message volume per processor. The other vector entries are needed by only one processor and should be assigned to the respective processors to avoid redundant communication.

We propose constructing two matrices C_x and C_y, referred to here as *communication matrices*, that summarize the communication on x- and y-vector entries, respectively. The matrix C_x has K rows and $|x_C|$ columns. For each row k, we insert a nonzero in column j if processor P_k has nonzeros in column corresponding to $x_C[j]$ in the fine-grain partitioning. Hence, the rows of C_x correspond to processors in such a way that the nonzeros in the row k identify the subset of x_C vector entries that are needed by processor P_k. The matrix C_y is constructed similarly. This time we put processors in columns and y_C entries in rows. Figure 4.1(b) and (c) show the communication matrices C_x and C_y for the sample matrix given in (a).

4.2.1 Unsymmetric partitioning model

We use *row-net* and *column-net* hypergraph models for representing C_x and C_y, respectively. In the row-net hypergraph model, matrix C_x is represented as hypergraph $\mathcal{H}_x$ for columnwise partitioning. Vertex and net sets correspond to the columns and rows of matrix C_x, respectively. There exist one vertex v_j and one net n_i for each column j and row i, respectively. Net n_i contains the vertices corresponding to the columns which have a nonzero in row i. That is, $v_j \in n_i$ if $C_x[i,j] \neq 0$. In the column-net hypergraph model of C_y, the vertex and net sets correspond to the rows and columns of the matrix C_y, respectively, with similar construction. Figure 4.2(a) and (b) show communication hypergraphs $\mathcal{H}_x$ and $\mathcal{H}_y$.

A K-way partition on the vertices of $\mathcal{H}_x$ induces a processor assignment for the expand operations. Similarly, a K-way partition on the vertices of $\mathcal{H}_y$ induces a processor assignment for the fold operations. In unsymmetric partitioning case, these two assignment can be found independently. In [99] and Chapter 3, we showed how to obtain such independent partitionings in order to minimize the four communication-cost metrics. The results of that work are immediately applicable to this case.



(a) (b) (c)

Figure 4.2: Communication hypergraphs: (a)$\mathcal{H}_x$, (b) $\mathcal{H}_y$, and (c) a portion of $\mathcal{H}$ corresponding to the communication matrices grin in Fig. 4.1.

4.2.2 Symmetric partitioning model

When we require symmetric partitioning on the vectors x and y, the partitionings on $\mathcal{H}_x$ and $\mathcal{H}_y$ cannot be obtained independently. Therefore, we combine hypergraphs $\mathcal{H}_x$ and $\mathcal{H}_y$ into a single hypergraph $\mathcal{H}$ as follows. For each part P_k, we create two nets x_k and y_k. For each $x_C[i]$ and $y_C[i]$ pair, we create a single vertex v_i. For each net x_k, we insert v_i into its vertex list if processor P_k needs $x_C[i]$. For each y_k, we insert v_j into its vertex list if processor P_k contributes to $y_C[j]$. We show vertices v_4 and v_7 of $\mathcal{H}$ in Fig. 4.2(c). Since the communication occurs in two distinct phases, vertices have two weights associated with them. The first weight of a vertex v_i is the communication volume requirement incurred by $x_C[i]$; hence we associate weight $d_i - 1$ with the vertex v_i. The second weight of a vertex v_i is the communication volume requirement incurred by $y_C[i]$; as in [99] we associate a unit weight of 1 with each v_i.

In a K-way partition of $\mathcal{H}$, an x_k-type net spanning λ_{xk} parts necessitates $\lambda_{xk} - 1$ messages to be sent to processor P_k during the expand phase. The sum of these quantities over all x_k-type nets thus represents the total number of messages sent during the expand phase. Similarly, a y_k-type net spanning λ_{yk} parts necessitates $\lambda_{yk} - 1$ messages to be sent by P_k during the fold phase. Again, the sum of these quantities over all y_k-type nets represents the total number of messages sent during the fold phase. The sum of the *connectivity* $-\ 1$

values for all nets thus represents the total number of messages in fine-grain partitioning of matrices. Therefore, by minimizing the objective function in Eq. 2.3 $\left(cutsize(\Pi) = \sum_{n_i \in \mathcal{N}} (\lambda_i - 1)\right)$, partitioning $\mathcal{H}$ minimizes the total number of messages. The vertices in part $\mathcal{V}_k$ represent the x-vector entries to be expanded and the respective y-vector entries to be folded by processor P_k. The load of the expand operations are exactly represented by the first components of vertex weights if for each $v_i \in \mathcal{V}_k$ we have $v_i \in x_k$. If, however, $v_i \notin x_k$, the weight of a vertex for the expand phase will be one less than the required. We hope these shortages to occur, in some extent, for every processor to cancel the diverse effects on communication-volume load balance. The weighting scheme for the fold operations is adopted with the rationale that every $y_C[i]$ assigned to a processor P_k will relieve P_k from sending a unit-volume message. If the net sizes are close to each other than this scheme will prove to be a reasonable one. As a result, balancing part sizes for the two set of weights, e.g., satisfying Eq. 2.4 $\left(\frac{W_{max} - W_{avg}}{W_{avg}} \leq \epsilon\right)$, will relate to balancing communication-volume loads of processors in the fold and expand phases, separately.

In the above discussion, each net is associated with a certain part and hence a processor. This association is not defined in the standard hypergraph partitioning problem. We can enforce this association by adding K special vertices, one for each processor P_k, and inserting those vertices to the nets x_k and y_k. Fixing those special vertices to the respective parts and using *partitioning with fixed vertices* feature of hypergraph partitioning tools [2, 22] we can obtain the specified partitioning on $\mathcal{H}$. However, existing tools do not handle fixed vertices within multi-constraint framework. Therefore, instead of obtaining balance on communication-volume loads of processors in the expand and fold phases separately, we add up the weights of vertices and try to obtain balance on aggregate communication-volume loads of processors.

Table 4.1: Properties of test matrices and partitioning times.

Matrix	Size		Part.	
	N	NNZ	Mthd	Time
CO9	10789	249205	PTH	11.43
			CHy	0.66
CQ9	9278	221590	PTH	9.60
			CHy	0.65
creb	9648	398806	PTH	26.71
			CHy	2.51
ex3s1	17443	679857	PTH	48.88
			CHy	13.58
fom12	24284	329068	PTH	22.07
			CHy	13.76
fxm3	41340	765526	PTH	37.73
			CHy	0.29
lpl1	39951	541217	PTH	27.04
			CHy	5.09
mod2	34774	604910	PTH	32.83
			CHy	2.18
pds20	33874	320196	PTH	18.65
			CHy	6.09
pltex	26894	269736	PTH	14.29
			CHy	1.11
world	34506	582064	PTH	30.84
			CHy	2.50

4.3 Experiments

We have conducted experiments on the matrices given in Table 4.1. In the table, N and NNZ show, respectively, the dimension of the matrix and the number of nonzeros. `Part.Mthd` give the partitioning method applied: `PTH` refers to the fine-grain partitioning of Çatalyürek and Aykanat [23], `CHy` refers to partitioning communication hypergraphs with fixed-vertex option and aggregate vertex weights. For these two methods, we give timings under the column `Part.Time`, in seconds.

The results of the experiments are given in Table 4.2. The `Srl.Time` column lists the timings for serial SpMxV operations in milliseconds. We used PaToH [22] library to obtain 24-way fine-grain partitionings on the test matrices. In all partitioning instances, the computational-load imbalance were below 7 percent. For each partitioning method, we dissect the communication requirements into the expand and fold phases. For each phase, we give total volume of messages, maximum volume-load of a processor, total number of messages, and maximum number of messages per processor. In order to see whether the improvements achieved by method `CHy` in the given performance metrics hold in practice, we also give timings, best among 20 runs, for parallel SpMxV operations, in milliseconds, under the column `Prll.Time`. All timings are obtained on machines equipped with 400 MHz Intel Pentium II processor and 64 MB RAM running Linux kernel 2.4.14 and Debian GNU/Linux 3.0 distribution. The parallel SpMxV routines are implemented using LAM/MPI 6.5.6 [18].

To compare our method against `PTH`, we opted for obtaining symmetric partitioning. For each matrix, we run `PTH` 20 times starting from different random seeds and selected the partition which gives the minimum in total-volume-of-messages metric. Then, we constructed the communication hypergraph with respect to `PTH`'s best partitioning and run `CHy` 20 times, again starting from different random seeds, and selected the partition which gives the minimum in total-number-of-messages metric. Timings for these partitioning methods are for a single run. In all cases, running `CHy` adds at most half of the time required by `PTH` to the framework of fine-grain partitioning.

In all of the partitioning instances, `CHy` reduces the total number of messages to somewhere between 0.47 (`fom12`) and 0.74 (`CO9`) of `PTH`. In all partitioning instances, `CHy` increases the total volume of messages to somewhere between 1.32 (`creb`) and 1.86 (`pds20`) of `PTH`. This is expected, because a vertex v_i may be assigned to a part $\mathcal{V}_k$ while P_k does not need any of $x_C[i]$ or $y_C[i]$. However, reductions in parallel running times are seen for all matrices except `lpl1`. The highest speed-ups achieved by `PTH` and `CHy` are 5.96 and 6.38, respectively, on `fxm3`.

Table 4.2: Communication patterns and running times for 24-way parallel SpMxV.

Matrix	Part. Mthd	Expand Phase				Fold Phase				Prll. Time	Srl. Time
		Total		Max		Total		Max			
		vol	msg	vol	msg	vol	msg	vol	msg		
CO9	PTH	2477	290	524	21	4889	358	473	22	4.55	12.54
	CHy	5121	223	318	22	7367	259	714	18	4.05	
CQ9	PTH	2642	313	471	23	4973	370	430	22	4.82	11.13
	CHy	5108	218	356	20	7249	264	729	16	4.09	
creb	PTH	9344	490	750	23	12660	504	871	23	6.64	19.3
	CHy	13047	313	715	23	16157	341	1068	20	5.92	
ex3s1	PTH	7964	312	602	22	26434	356	1762	20	8.39	33.52
	CHy	19537	195	1128	23	36347	270	2252	16	7.91	
fom12	PTH	7409	228	559	23	21208	228	1143	13	5.03	19.86
	CHy	16713	96	983	10	28151	119	1541	8	4.03	
fxm3	PTH	1843	212	279	23	2662	236	282	17	6.39	38.13
	CHy	3299	142	215	16	4027	156	456	15	5.97	
lpl1	PTH	7646	226	1062	20	13752	253	961	17	5.73	29.81
	CHy	15079	166	892	22	20582	186	1507	12	5.83	
mod2	PTH	5015	267	845	23	9421	278	1135	22	6.92	30.81
	CHy	10523	181	656	23	14142	198	1517	17	5.92	
pds20	PTH	5373	299	557	23	13548	317	956	19	5.23	17.82
	CHy	14066	177	794	18	21302	201	1436	13	4.95	
pltex	PTH	1883	167	172	16	7065	273	508	20	4.27	14.64
	CHy	4533	89	311	16	8828	139	782	10	3.63	
world	PTH	4934	300	794	23	9710	316	1295	23	7.35	29.63
	CHy	10679	181	656	23	14854	205	1745	18	6.05	

Chapter 5

Preconditioned iterative methods

This chapter addresses the parallelization of the preconditioned iterative methods that use explicit preconditioners such as approximate inverses. Efficient parallelization of a full step in these methods requires the coefficient and preconditioner matrices to be well partitioned. We first show that different methods impose different partitioning requirements for the matrices. Then, we develop hypergraph models to meet those requirements. In particular, we develop models that enable us to obtain partitionings on the coefficient and preconditioner matrices simultaneously. Experiments on a set of unsymmetric sparse matrices show that the proposed models yield effective partitioning results. A parallel implementation of the right preconditioned BiCGStab method on a PC cluster verifies that the theoretical improvements obtained by the models hold in practice.

5.1 Introduction

We consider the parallelization of the preconditioned iterative methods that use explicit preconditioners such as approximate inverses or factored approximate inverses. Our objective is to develop methods for obtaining one dimensional (1D) partitions on a coefficient matrix and its preconditioner matrix or factors of the preconditioner matrix simultaneously to efficiently parallelize a full step of the preconditioned iterative methods. We assume preconditioner matrices or their sparsity patterns are available beforehand. It has been shown that the rates of convergence of iterative methods depend on the partitioning method when the preconditioners are built from partitioned coefficient matrices [30]. With the above assumption in mind, we neither deteriorate nor improve the effects of the selected preconditioners on the rate of convergence. Our assumption is justified in applications where the preconditioner matrices can be reused, see for example [14] and a discussion on it in [12]. More adequately, some preconditioner constructing methods [69, 70] require a priori sparsity patterns for the preconditioner matrices, and some works on generating desirable sparsity patterns for these methods exist in the literature [27, 28, 59].

Approximate inverse preconditioning techniques explicitly compute and store a sparse matrix $M \approx A^{-1}$ to be used as a preconditioner. Application of such preconditioners merely require one or two matrix-vector multiply operations. That is, iterative methods that use approximate inverse preconditioners perform matrix-vector multiply operations both with the coefficient and preconditioner matrices. Two types of approximate inverses exist in the literature. In the first type, an approximate inverse is stored as a single matrix, whereas in the second type it is stored as a product of two matrices. The second type of preconditioners are referred to as factored approximate inverses. Among the most notable approximate inverse preconditioners are AINV and its variants by Benzi, Tuma, Meyer, Cullum, and Haws [7, 8, 9, 10]; SPAI by Grote and Huckle [46]; FSAI by Kolotilina and Yeremin [69, 70]; MR by Chow and Saad [29]. See [6, 11, 43] for a recent survey and the use of the approximate inverse preconditioning techniques. See [74, 86] for a general treatment of the preconditioning techniques.

Hendrickson and Kolda [51] give a thorough survey of the graph partitioning models used for partitioning sparse matrices for parallel processing. In these models, the partitioning constraint of maintaining balance on part weights corresponds to maintaining computational load balance. The partitioning objective of minimizing the cutsize of a partition defined over the edges or hyperedges relates to minimizing the total communication volume. Among those models, the hypergraph models by Çatalyürek, Aykanat, and Pınar [3, 20, 21, 82] and the bipartite graph model by Hendrickson and Kolda [52, 68] are said to have more expressive power than the other models [21, 49, 51, 52]. These models have the flexibility of producing unsymmetric partitions on the input and output vectors of the sparse matrix-vector multiplies. A distinct advantage of the hypergraph models over both the standard graph and the bipartite graph models is that the partitioning objective in the hypergraph models is an exact measure of the total communication volume, whereas the objective in the graph models is an approximation [21, 49, 51, 52]. As noted in the survey [51] and in [49], all these graph and hypergraph models, except the bipartite graph model, are used to optimize a single sparse matrix-vector multiply operation. However, matrix-vector multiply operations are only a piece of a larger computation in the preconditioned iterative methods. Therefore, new partitioning models that optimize a full step in these iterative methods are needed as also stated by Hendrickson [49].

Since the proposed models are built using computational hypergraph models for sparse matrix partitioning [21], we suggest the reader review these models (discussed in Chapter 2). We discuss a procedure to analyze iterative methods in order to determine partitioning requirements for efficient parallelization and illustrate the procedure on a well known iterative method in §5.3. The partitioning requirements of a number of widely used iterative methods are also given in the same section. We propose methods to build composite hypergraph models for meeting the partitioning requirements in the preconditioned iterative methods in §5.4. We discuss the applicability of the composite hypergraph models to a few additional scientific applications and relate the models to some existing works in §5.5. The proposed methods are evaluated both theoretically and practically in §5.6.

5.2 Background

The iterative methods that use approximate inverse preconditioners perform matrix-vector multiplies with both coefficient and preconditioner matrices. Usually, these multiply operations are performed on after another without any other intermittent computation. In other words, the computational core of these methods is a chain of matrix-vector multiplies.

5.2.1 Matrix-chain-vector multiplies

Consider the computations of the form $y \leftarrow A_1 A_2 \cdots z$. Rather than forming the matrix-chain-product $A_1 A_2 \cdots$, which may be quite dense, the above computation is performed as a sequence of matrix-vector multiplies. In particular, the computations of the form $y \leftarrow AMz$ are performed as $x \leftarrow Mz$ and then $y \leftarrow Ax$. If the matrices A and M are partitioned rowwise and columnwise respectively, then the parallel matrix-chain-vector multiply executes the following steps:

1. Execute the column-parallel algorithm given in §2.1.2 to obtain $x \leftarrow Mz$.
2. Execute the row-parallel algorithm given in §2.1.1 to obtain $y \leftarrow Ax$.

In this parallel algorithm, if the permutation on the rows of M is different than the permutation on the columns of A, then the x-vector entries should be reordered in between the two multiplies. Since the reordering requires communication it should be avoided. In other words, the permutations on the columns of A and rows of M should be the same. To meet this requirement matrices should be partitioned simultaneously.

5.3 Determining partitioning requirements

In iterative methods, all vectors that participate in a linear vector operation should be partitioned conformably in order to avoid the communication of the vector entries during the operation. To obtain such conformable partitionings, we classify the vectors according to their relations to the inputs and outputs of the matrix-vector multiplies. In particular, we call a vector to be in the *input-space* of a matrix A if it is multiplied with A or it undergoes linear vector operations with other input-space vectors. Accordingly, we call a vector to be in the *output-space* of a matrix A if it is obtained by multiplying A with another vector or it undergoes linear vector operations with other output-space vectors. For example, in $y \leftarrow Ax$ multiply, the y vector is in the output-space of A, whereas x is in the input-space of A.

In some iterative methods, e.g., the conjugate gradients [42], the input-space and out-space of the A matrix coincide, i.e., the input-space vectors undergo linear vector operations with the output-space vectors. Such methods require symmetric partitioning PAP^T in which all vectors are partitioned conformably with the permutation P. In some other methods, the input-space and output-space of A differ. Such methods allow unsymmetric partitioning PAQ in which all output-space vectors are partitioned conformably with P, whereas all input-space vectors are partitioned conformably with Q. If the method involves more than one multiply with different matrices, the output-space of one matrix may coincide with the input-space of another one. In this case, the output-space permutation for the first one becomes an input-space permutation for the other one.

All vectors in a full step of an iterative algorithm should be analyzed in terms of their relations to the input- and output-spaces of all matrices to determine the partitioning requirements. We analyze the right preconditioned BiCGStab[1] method [104] given in Fig. 5.1 and determine its partitioning requirements as an example. There are ten vectors in the method: $r, b, \tilde{r}, x, p, v, \hat{p}, s, \hat{s}$, and t. Because of the linear vector operations in lines 1,2,4,7,10,14,15,19,20, and 21, the vectors $r, b, \tilde{r}, p, v, s, t$, and x should be partitioned conformably. All these vectors are in the output-space of A because of the matrix-vector multiplies in the lines 13 and 18. We are left with the vectors $\hat{p}$ and $\hat{s}$. Because of the matrix-vector multiplies in the lines 13 and 18, these two are in the input-space

[1]Please note that the given code works with preconditioned x vector, i.e, the solution vector x obtained at the termination is a solution to $AMx = b$. That is, in order to get a solution to $Ax = b$ we have to multiply x with the approximate inverse preconditioner M at the termination. However, using $\hat{p}$ and $\hat{s}$ instead of p and s in line 20 would yield the solution to $Ax = b$ without any other operations as given by Barret et. al [5].

BiCGStab(A,M,x,b)#Solve Ax=b using the right preconditioner M
begin
  $r^{(0)} = b - AMx^{(0)}$ for some initial $x^{(0)} = x$
  $\tilde{r} = r^{(0)}$
  for $i = 1, 2, \cdots$ **do**
    $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
    if $\rho_{i-1} = 0$ method fails
    if $i = 1$
      $p^{(i)} = r^{(i-1)}$
    else
      $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
      $p^{(i)} = r^{(i-1)} + \beta_{i-1}\left(p^{(i-1)} - \omega_{i-1}v^{(i-1)}\right)$
    endif
    $\hat{p} = Mp^{(i)}$
    $v^{(i)} = A\hat{p}$
    $\alpha_i = \rho_{i-1}/\tilde{r}^T v^{(i)}$
    $s = r^{(i-1)} - \alpha_i v^{(i)}$
    check norm of s; if small enough; set $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ and stop
    $\hat{s} = Ms$
    $t = A\hat{s}$
    $\omega_i = t^T s / t^T t$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} + \omega_i s$
    $r^{(i)} = s - \omega_i t$
    check convergence; continue if necessary
    for continuation it is necessary that $\omega_i \neq 0$
  endfor
end

Figure 5.1: Preconditioned BiCGStab using the approximate inverse M as a right preconditioner.

Table 5.1: Iterative methods and partitioning requirements.

Method	Partitioning requirement	Number of distinct vector partitionings
BiCGStab right precond. [5, 46]	$PAMP^T$	2
BiCGStab right factor. precond. [5]	$PAM_1M_2P^T$	3
symmetric GMRES right precond. [25]	PAP^T–PMP^T	1
GMRES right precond. [86]	$PAMP^T$	2
GMRES left precond. [86]	$PMAP^T$	2
TFQMR symmetric precond [41]	PAP^T–$PM_2M_1P^T$	2
TFQMR original form [39]	$PM_1AM_2P^T$	3
CGNE [86]	PAQ–PMP^T	2
CGNR [86]	QAP^T–PMP^T	2
CGS right precond. [5]	$PAMP^T$	2
PCG [5, 42, 70]	PAP^T–PMM^TP^T	2
	PAP^T–PMP^T	2

of A, and thus can have a different partition Q. Since we have completed the classification of vectors, we can determine the partitioning requirements for the A and M matrices. The input- and output-spaces of A differ. Therefore, the partitioning requirement for the A matrix is PAQ. The vectors $\hat{p}$ and $\hat{s}$ are in the output-space of M because of the matrix-vector multiplies in the lines 12 and 17. Therefore, the output-space of M coincides with the input-space of A. Similarly, the input-space of M coincides with the output-space of A through vectors p and s. Therefore, the partitioning requirement for the M matrix is Q^TMP^T. The overall requirement is thus PAQ and Q^TMP^T. We express this requirement as $PAMP^T$ to simplify the notation.

We examined a number of widely used preconditioned iterative methods whose codes are given in the literature. We noticed that different methods have different partitioning requirements as shown in Table 5.1. Several caveats are necessary for the table to be useful.

1. We analyze the methods in their original form as given in the references, i.e., we do not consider any type of code optimizations for performance

gains.

2. If any two matrices are written consecutively, then the two matrix-vector multiplies involving the matrices follow each other without any interleaving synchronization. In such cases, the input-space of the first matrix and the output-space of the second matrix coincide. In other words, there is an arbitrary permutation matrix and its transpose in between the two matrices which designates a distinct vector partitioning. For example, $PAMP^T$ means unsymmetric partitions PAQ for A and Q^TMP^T for M. We write the number of distinct vector partitionings for each method in the rightmost column of Table 5.1.

3. Listing two partitioning requirements separated by "–" means that there is at least one synchronization point between the two matrix-vector multiplies. Therefore, we distinguish the partitioning requirement PAP^TPMP^T from PAP^T–PMP^T. The first one only states that the output-spaces of the matrices coincide with the input-spaces of the matrices. The second partitioning requirement, however, further states that the two matrix-vector multiplies are interleaved with synchronizations.

4. Factored approximate inverse M_1M_2 can be used (table contains such an example for BiCGStab) instead of M by just writing the factors consecutively in place of M to determine their partitioning requirement. For example, the use of a factored approximate inverse in the right preconditioned CGNE necessitates PAQ–$PM_1M_2P^T$ which in turn gives the requirements PAQ, PM_1R, and $R^TM_2P^T$.

5. The given requirements are independent of the dimensions along which the matrices are partitioned. That is, matrices can be partitioned rowwise or columnwise, whichever is preferred.

In choosing a partitioning dimension, three issues should be considered. The first issue is the individual matrix characteristics, i.e., the number of nonzeros per rows and columns. If, for example, a matrix has dense rows but no dense columns, then it is advisable to partition it along the columns [52]. Partitioning along the rows will probably lead to a poor load balance among the processors. Besides, a dense row will span a large number of column segments in the off-diagonal blocks of the block structure and thus will lead to a high volume of communication. However, in columnwise partitioning it will be easy to balance the computational loads. Furthermore, a dense row will span at most one nonzero row segment in each off diagonal blocks of the block structure and thus will contribute at most one word of communication volume per each off diagonal block.

The second issue in choosing a partitioning dimension is the relation between the partitioning requirement and the set of atomic tasks to be partitioned. For example, in the $PAMP^T$ case, we have four partitioning choices for the pair of A (of size $m \times n$) and M (of size $n \times m$) matrices: *rowwise-rowwise* (RR), *rowwise-columnwise* (RC), *columnwise-rowwise* (CR), and *columnwise-columnwise* (CC). In the RR scheme, the partitioning determines the output-space permutation for the two matrices. Since the output-spaces of the two matrices differ there are a total of $m+n$ tasks to partition. In the CC scheme, the partitioning determines the input-space permutation. Since the input-spaces differ there are a total of $n+m$ tasks to partition. In the RC and CR schemes, the partitioning determines the permutation for the coinciding input- and output-spaces. Therefore, in the RC and CR schemes, the number of tasks reduces to m and n, respectively.

The third issue is the arrangement of computations and communications. Consider the partitioning requirement of $PAMP^T$ for the multiplies of the form $y \leftarrow AMz$ which are performed as $x \leftarrow Mz$ and then $y \leftarrow Ax$. For each multiply, there exists an expand or a fold communication operation. The partitioning dimension determines whether these communications take place before or after the local computations. In the RC partitioning scheme, the fold and expand operations take place successively in between the two multiplies. There are dependencies between these two communication operations; before expanding a particular x-vector entry it should be folded. Because of these dependencies, the successive fold and expand operations are likely to incur a local synchronization point which separates the two multiplies. Therefore, processors' loads should be balanced for individual matrix-vector multiplies in this partitioning scheme. The

RR and CC schemes have either an expand or a fold in between the two multiply operations. Such communications do not incur synchronization points under the assumption that each processor has enough local computation which overlaps with the incoming messages. If the two matrices have comparable number of nonzeros, processors' loads should be balanced for individual matrix-vector multiplies in these two partitioning schemes for scalability. The CR scheme is unique in that the two multiply operations are performed without any communication in between the two multiplies. In this partitioning scheme, processors' loads should be balanced in terms of their total loads in the two multiplies.

5.4 Building composite hypergraph models

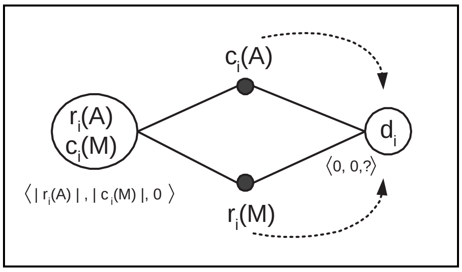
We combine individual hypergraph representations of the coefficient matrix A and the preconditioner matrix M or its factors M_1 and M_2 into a composite hypergraph whose partitioning meets the requirements given in §5.3. We define four operations to combine the hypergraph representations of the individual matrices. These four operations are called vertex amalgamation, vertex weighting, vertex insertion, and pin addition. The first operation is used to enforce identical partitions on the vertices of the individual hypergraphs. The second operation is used to enable load balancing. The last two operations are used to define mapping policies for the nets of the individual hypergraphs. The key point in all these operations is to preserve the identities of the nets of the individual hypergraphs.

In the following discussion, we assume that A is to be partitioned rowwise, and M and M_1 are to be partitioned columnwise, and M_2 is to be partitioned rowwise. That is, we have column-net hypergraphs for A and M_2 and row-net hypergraph for M_1.

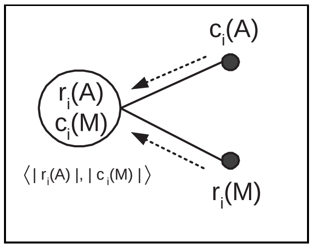
Vertex amalgamation. This operation is used to enforce identical partitions along the partitioning dimensions of the matrices. In this operation, the vertices of the individual hypergraphs are combined into a single vertex. The nets of the resulting composite is set to the union of the nets of the constituent vertices. For example, in the PAP^T–PMP^T case, we amalgamate the row-vertex $r_i(A)$ with the column-vertex $c_i(M)$ into v_i so that $Nets(v_i) = Nets\,(r_i(A)) \cup Nets(c_i(M))$ (see Fig. 5.2(b)). In a partitioning, v_i being in a part $\mathcal{V}_k$ shows the processor P_k being responsible for performing multiplications with the ith row of A and the ith column of M.

Vertex weighting. This operation is used to enable load balancing. Remember that in some of the iterative methods there are synchronization points between different matrix-vector multiplies. That is, computations occur in phases. Therefore, we define multiple weights for vertices; one for each computation phase. For a certain phase, the weight of a vertex is set to the weight of the constituent vertex in the hypergraph of the matrix associated with that phase. Consider right preconditioned symmetric-GMRES [25] and its partitioning requirement PAP^T–PMP^T. As seen in Fig. 5.2(b), we amalgamate vertices of the individual hypergraphs. Since the application of the preconditioner M occurs in a different phase, the vertex shown has two weights. The first weight represents the computational load associated with the ith row of A, i.e., $|r_i(A)|$. The second weight represents the computational load associated with the ith column of M, i.e., $|c_i(M)|$. In some cases, different matrix-vector multiplies occur successively without any interleaving synchronization. In these cases, the weight of a composite vertex is set to the sum of the weights of the constituting vertices. Consider the TFQMR method using symmetric preconditioning and its partitioning requirement PAP^T–$PM_1M_2P^T$. Since M_1 and M_2 are partitioned columnwise and rowwise, respectively, there is no synchronization between their respective matrix-vector multiplies. Therefore, the weights of $c_i(M_1)$ and $r_i(M_2)$ are added up as seen in Fig. 5.2(c).

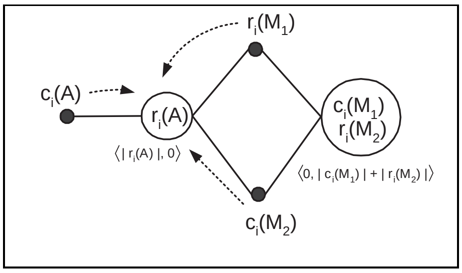
Vertex insertion. This operation is used to make mapping policies for the nets. It is used to have the same partitioning for the vectors associated with two different set of nets when the partitioning on vertices is different than what is required. A new dummy vertex d_i is created whose nets are the ith nets of the individual hypergraphs and a policy on mapping these nets with d_i is made. Consider the partitioning requirement $PAMP^T$. As shown in Fig. 5.2(a),



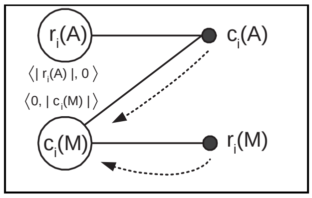
(a) $PAMP^T$



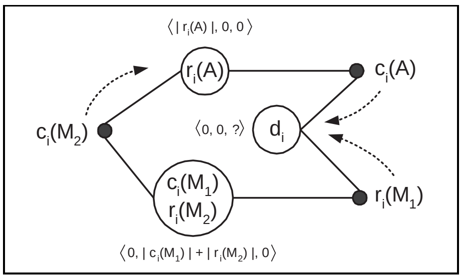
(b) PAP^T–PMP^T



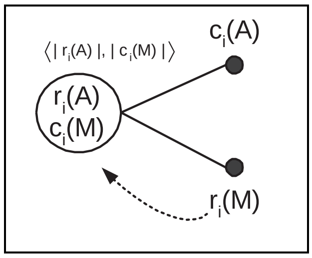
(c) PAP^T–$PM_1M_2P^T$



(d) QAP–P^TMP



(e) $PAM_1M_2P^T$



(f) PAQ–PMP^T

Figure 5.2: Composite hypergraph models for different partitioning requirements. The d_i vertices are created while building the composite hypergraphs. The nets of the d_i vertices are fully shown. Other vertices are coming from the individual hypergraphs. We use $c_i(\cdot)$ and $r_i(\cdot)$ to represent, respectively, the ith column and ith row of the matrices. We use $|c_i(\cdot)|$ and $|r_i(\cdot)|$ to represent the number of nonzero elements in the ith column and row respectively. The weights of the vertices are given in between $\langle$'s and $\rangle$'s next to the vertices. The nets are labeled with a single $r_i(\cdot)$ or a single $c_i(\cdot)$ according to their counterparts in the individual hypergraphs. The dashed lines originating from a net and pointing at some vertex displays the mapping policy on the respective net. These policies are made in the vertex insertion and pin addition operations.

we create the vertex d_i and connect it to the column-net $c_i(A)$ and the row-net $r_i(M)$. The vertex d_i being in a part $\mathcal{V}_k$ shows that the processor P_k is responsible for folding and holding the ith entries of the vectors in the output-space of M, and it also shows that P_k is responsible for expanding and holding the ith entries of the vectors in the input-space of A. Note that the computational cost associated with d_i depends on the connectivity of its nets. Therefore, we cannot assign exact weights to those vertices before partitioning take place.

Pin addition. This operation is used to define mapping policies for the nets. Different than the vertex insertion operation, the pin addition operation connects the nets to the existing vertices. Recall that we obtain partitioning on the vectors associated with nets by mapping a net to a part holding one of its vertices. Suppose all vertices of a net are permuted according to P, and we seek another permutation Q on the vectors associated with that net. In this case, we connect the net to an appropriate vertex which will be permuted according to Q and make the mapping policy for that net. Consider the BiCGStab method with the factored approximate inverse preconditioner M_1M_2 and the method's partitioning requirement $PAM_1M_2P^T$. Since we partition M_2 by rows, and the vectors in its input-space are to be partitioned with P^T conforming to the rowwise partition of A, we connect the net $c_i(M_2)$ to the vertex $r_i(A)$ as shown in Fig. 5.2(e). The pins along which the policies are made must be present in a composite hypergraph. If they are missing in the individual hypergraphs, then they must be added. In Figs. 5.2(a)–(f), existence of all pins along the direction of the dashed lines (mapping policies) that are not pointing to a dummy vertex are subject to pin-addition operation. The existence of other pins, for example the one between the composite vertex and the nets $c_i(A)$ and $r_i(M)$ in Fig. 5.2(a), depends on the sparsity patterns of the matrices.

Given a partition on the composite hypergraph, we extract the row and column permutation matrices for each matrix. The partition on the vertices define a permutation for either the rows or the columns of each of the matrices according to the partitioning dimensions. To define permutations for the other dimensions of the matrices, we obtain a consistent permutation on the nets of the composite hypergraph and then project this permutation to the nets of the individual hypergraphs. While forming the composite hypergraph, we define a mapping policy for some nets. These policies enable us to map those nets consistently. The remaining nets are not restricted to be mapped with a specific vertex; we arbitrarily map these nets to a part in their connectivity set. Observe that the resulting permutation on the composite hypergraph is consistent, whereas the projected permutations on the individual hypergraphs do not have to be as such, because of the vertex insertion and pin addition operations. However, the consistency of the permutation on the nets of the composite hypergraph is sufficient to have the following theorem.

Theorem 5.1 *The cutsize of a partition in a composite hypergraph formed by applying the above operations on individual hypergraph representations of a number of matrices quantifies the total volume of communication in the respective sparse matrix-vector multiplies.*

Proof. In order to prove the theorem, we again add the connectivity of the external nets under a consistent permutation.

There are two types of external nets. The first type of nets are those that have at least one original vertex in each part in their connectivity set. These nets are handled using the same arguments in the proof of Theorem 2.1.

The second type of nets are those that are connected to a part only through the newly added vertices. Observe that each such net contains one newly added vertex which defines the permutation policy. Consider a column net c_i of this type. Let $P_k \in \Lambda_i$ be the part that holds the new vertex v_i. The other vertices of the net c_i represent the atomic inner product operations that needs the vector entry, say x_i, associated with c_i. Since we map the net c_i with the vertex v_i, the owner of x_i is P_k. Hence, P_k has to send x_i to the processors in $\Lambda_i - \{P_k\}$. Therefore, the volume of communication associated with net c_i is again $\lambda_i - 1$. The row nets of the second type are handled similarly. Therefore, the overall sum of the *connectivity* -1 of the nets again corresponds to the total communication volume. □

Guidelines for combining hypergraphs. To build a composite hypergraph the followings should be applied.

G1. Determine partitioning requirements for each matrix through analyzing vector operations as discussed in §5.3.

G2. Decide on the partitioning dimension. Generate row-net hypergraph model for the matrices to be partitioned columnwise. Generate column-net hypergraph model for the matrices to be partitioned rowwise.

G3. Apply vertex operations:

(i) If the partitioning requirements impose identical partitions on any two vertices, then apply vertex amalgamation to those vertices.

(ii) For each vertex of the composite hypergraph, apply the vertex weighting operation. If there is no synchronization point between the associated matrix-vector multiplies or there is a local synchronization point and the matrices do not have comparable number of nonzeros, then add up the weights of the constituting vertices, else associate multiple weights.

G4. Apply net operations (make policies on mapping nets):

(i) If a net ought to be mapped with a specific vertex, then make a policy for that net. If the net is not connected to the specific vertex, then apply the pin addition operation.

(ii) If two nets ought to be mapped together and independent of the existing vertices, then apply vertex insertion.

Illustration. Consider the right preconditioned BiCGStab method and its partitioning requirement $PAMP^T$ obtained in §5.3 according to G1. Let A and M be the matrices shown in Fig. 5.3(b). Suppose that A is to be partitioned columnwise and M is to be partitioned rowwise. We generate row-net and column-net hypergraph models of A and M, respectively according to G2 as shown in Figs. 2.2 and 2.1. The partitioning requirement imposes identical partitionings on the columns of A and rows of M. Hence, we amalgamate vertices of

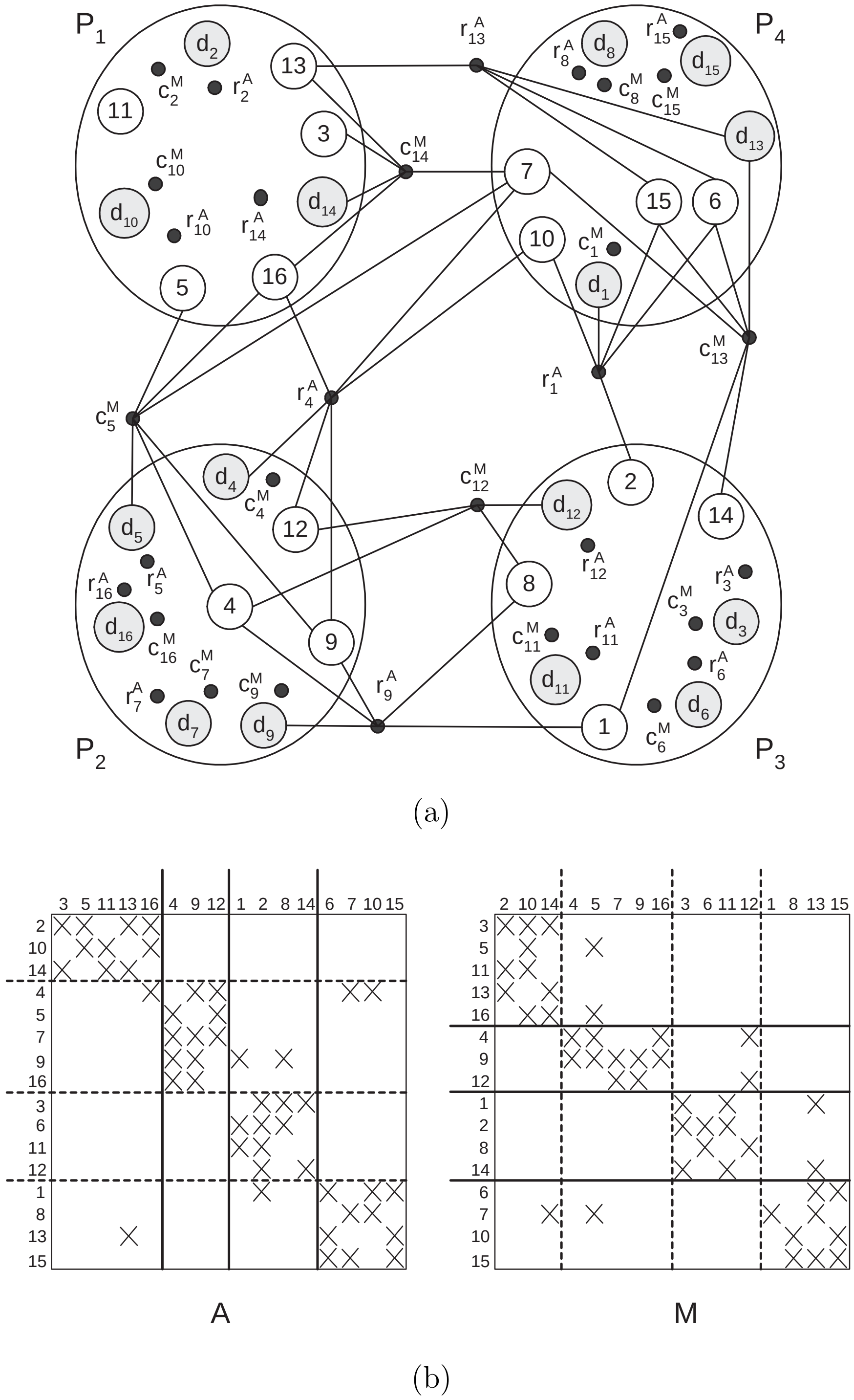


Figure 5.3: (a) A composite hypergraph formed by row-net hypergraph of A and column-net hypergraph of M and a partitioning which meets the requirement $PAMP^T$. The pins of the internal nets are not shown. (b) A columnwise partitioning of A and a rowwise partitioning of M induced by the composite hypergraph partitioning.

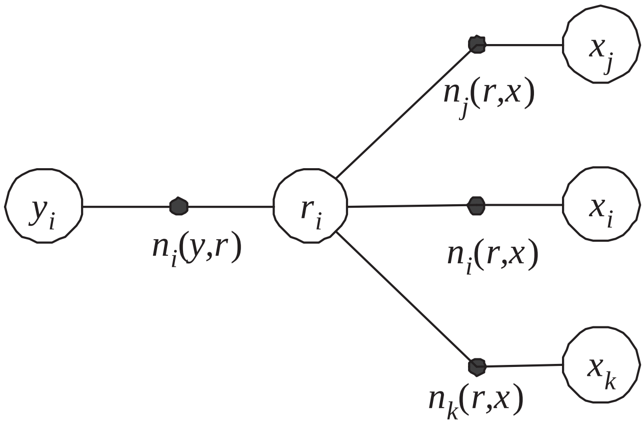
A and M according to G3.i. The method has no synchronization point between the two multiplies, and the separated expand and fold operations do not cause synchronization. Therefore, we just add the vertex weights. Corresponding nets of the two hypergraphs have to be permuted together and independent of the existing vertices. Therefore, we apply vertex insertion according to G4.ii. The resulting hypergraph is shown in Fig 5.3(a). In this figure, the dummy vertices are shaded. Other vertices and nets are inherited from the previous figures. In order to distinguish the nets, the source matrix names are written next to them. The pins of the internal nets are not shown for the sake of clarity. The nets have to be permuted to the part that holds the associated dummy vertices. The permutations on the matrices induced from the composite hypergraph partitioning are shown in Fig. 5.3(b). As seen from the figure, the cutsize and hence the total volume of communication is 10 where each multiply contributes five.

5.5 Further notes

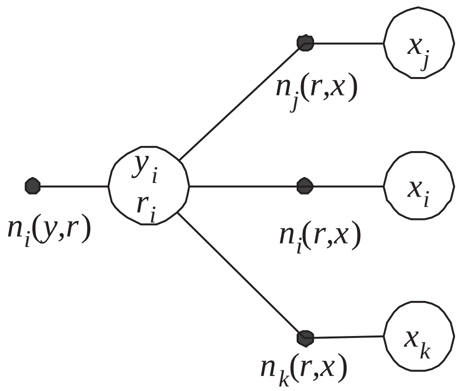
5.5.1 Revisiting hypergraph models for 1D partitioning

Consider the computations of the form $y \leftarrow Ax$ under rowwise partitioning of the matrix A. Since we partition A, x, and the resulting vector y, there should be three types of vertices in a hypergraph: y-vertices, row-vertices, and x-vertices. Each y-vertex depends on a particular row-vertex, i.e., y_i and r_i are connected with a specific net $n_i(y, r)$ for all i. Since the x-vertices are the sources that the computations depend on, the row-vertices depend on the x-vertices, i.e., x_i is connected to the row-vertices which correspond to the rows that have nonzeros in column i with a specific net $n_i(x, r)$. Figure 5.4(a) shows the hypergraph. In this hypergraph, there are $m+m+n$ vertices and $m+n$ nets. This hypergraph model is the most general model for 1D rowwise partitioning, because by partitioning the vertices of this hypergraph we can specify partitions on all operands of the matrix-vector multiply operation.

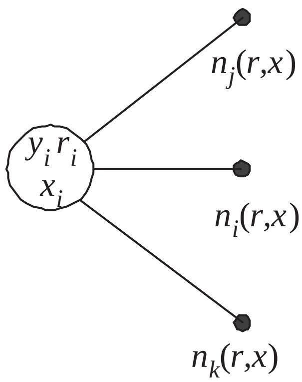
Now, we show how to modify the hypergraph specified above by applying the



(a) General 1D partitioning model.



(b) 1D unsymmetric partitioning model with the owner computes rule.



(c) 1D symmetric partitioning model with the owner computes rule.

Figure 5.4: (a) All operands of the SpMxV operation are partitioned. (b) Vertex amalgamation operation is applied to enforce the owner computes rule. (c) Vertex amalgamation operation is applied to obtain 1D symmetric partitioning.

operations proposed in this chapter to obtain 1D unsymmetric and symmetric partitionings. First, we can apply owner computes rule, i.e., y_i should be computed by the processor which owns r_i. This requires amalgamating the vertices y_i and r_i for all i. A portion of the resulting hypergraph is shown in Fig. 5.4(b). Since nets of size one does not contribute to the partitioning cost, we can delete the net $n_i(y, r)$ from the model. Partitioning the resulting hypergraph will produce nonsymmetric partitions. Suppose we are seeking symmetric partitions, i.e., the processor which owns r_i and y_i should own x_i. This time, we have to amalgamate the vertices y_i/r_i and x_i for all i. A portion of the resulting hypergraph is shown in Fig. 5.4(c). Partitioning the resulting hypergraph will produce symmetric partitions. Note that the hypergraph shown in Fig. 5.4(c) is the column-net hypergraph model discussed in [21]. Observe that the vertex amalgamation operation between the vertex x_i and y_i/r_i connects the ith vertex to the ith net. This observation clarifies the issue that in 1D computational hypergraph model of Çatalyürek and Aykanat, all of the diagonal entries of the matrices should be nonzero if symmetric partitioning is sought.

We think that it is possible to simplify building composite models through using the generalized hypergraph models. We will report this issue in [102]

5.5.2 Investigations on the composite models

Consider the partitioning requirements for the CGNE and CGNR methods given in Table 5.1. In the CGNE method, when the matrix A is partitioned rowwise, we have a leeway in defining a consistent column permutation for A as shown in Fig. 5.2(f). Similarly, when the matrix A is partitioned columnwise, we have the same leeway in defining a consistent row permutation for A in the CGNR method which requires QAP–PMP^T. In such cases, we can use this freedom to minimize other communication cost metrics as mentioned in Chapter 3. However, since the techniques in [99, 105] can only be applied to the communications regarding A, we expect a limited improvement here.

Consider composite hypergraphs in which two nets are permuted with a dummy vertex as shown in Figs. 5.2(a) and 5.2(e). These dummy vertices are added to have a vertex partition induce a particular permutation on the nets. Without changing the computational load distribution, any two nets that share a dummy vertex can be re-permuted. In [97], we used this freedom in permuting two nets together to minimize multiple communication cost metrics in 2D partitioning of sparse matrices. In a similar vein, we can construct a communication hypergraph $\mathcal{H}_C = (\mathcal{V}_\mathcal{N}, \mathcal{N}_\Pi)$ for a composite hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$—in the form of Fig. 5.2(a) or 5.2(e)—and its partitioning Π. For each column-net $c_i(A)$ and row-net $r_i(M)$ pair, we create a single vertex $v_i \in \mathcal{V}_\mathcal{N}$. For each part P_k, we create two nets a_k and m_k and connect them by inserting a new vertex marked as fixed to the kth part during partitioning $\mathcal{H}_C$. We insert vertex v_i to the pin-list of the net a_k if column-net $c_i(A)$ connects kth part under the given partition Π of $\mathcal{H}$. Similarly, we insert vertex v_i to the pin-list of the net m_k if row-net $r_i(M)$ connects kth part in Π. In a partition Π_C of the communication hypergraph $\mathcal{H}_C$, a vertex v_i in a given part can be used to permute both column-net $c_i(A)$ and row-net $r_i(M)$ to that part. As shown in [97], minimizing the cutsize of Π_C minimizes the total number of messages, and maintaining balance on part weights using a loosely specified vertex weight relates to maintaining balance on the communication volume loads of processors.

5.5.3 Generalizations and related work

The computational structure of the preconditioned iterative methods is similar to the computational structure of a more general class of scientific computations including multi-phase, multi-physics, and multi-mesh simulations.

In multi-phase simulations, there are a number of computational phases separated by global synchronization points. The existence of the global synchronizations necessitates each phase to be load balanced individually. In our model, the multiple weights associated with vertices are used to achieve this goal. The works in [64, 106] also uses multiple weights for the parallelization of multi-phase simulations.

In multi-physics simulations, a variety of materials and processes are analyzed by using different physics procedures. In this type of simulations, computations as well as the memory requirements are not uniform across the mesh [91]. For scalability issues, processor loads should be balanced in terms of these two components. The multi-constraint partitioning framework also addresses this kind of problems [91].

In multi-mesh simulations, a number of grids with probably different discretization schemes and with arbitrary overlaps are used. The existence of overlapping regions/grid points necessitates simultaneous partitioning of the grids [91]. This simultaneous partitioning should balance the computational loads of the processors and minimize the communication cost due to interactions within an individual grid as well as the interactions among the different grids. The vertex amalgamation operation used in our models can be applied to overlapped regions to build the composite hypergraph. With the use of vertex weighting operations, our models thus can be used to address the partitioning problem in the multi-mesh computations. Although the simultaneous partitioning seems to be more adequate for this type of problems, independent partitioning is also possible. In fact, Plimpton et al. [85] reported promising results on using independent partitionings for a two grid system.

In some contact/impact problems there is a priori knowledge about the to be contacting surfaces. Karypis [61] and Plimpton et al. [84] report an implementation [57] which uses this information to decompose the underlying mesh among processors. The implementation uses the graph model and adds edges between the to be contacting surface elements. Partitioning such a graph using two-constraints balances the loads of processors both for the finite element analysis phase and for the contact detection phase, meanwhile by minimizing the edge cut the partitioning algorithm minimizes the communication cost. By modeling the interactions among the to be contacting surface elements with hypergraphs, we can build a composite hypergraph to address this kind of problems. However, the implementation in [57] is reported to suffer from load imbalances and to be limited to a small number of processors [84].

In obtaining partitionings for two or more computation phases interleaved with synchronization points, our models lead to the minimization of the overall sum of the total volume of communication in all phases. For the preconditioned iterative methods, minimizing the overall sum of communication volume will likely minimize the communication cost in one step of these methods. However, in more sophisticated simulations, the magnitude of the interactions in one phase may be different than that of the interactions in another one. In such settings, minimizing the total volume of communication in each phase separately may be advantageous [90]. This problem can be formulated as a multi-objective hypergraph partitioning problem [1, 92] on the composite hypergraphs.

As discussed above, our models can be applied to the multi-phase, multi-physics, and multi-mesh computations but subject to the following limitations. The dependencies must remain the same throughout the computations; our methods cannot be used, for example, in adaptive mesh refinement. The weights assigned to elements, for load balancing issues, should be static and available before the partitioning takes place; hence our methods cannot be used for applications whose computational requirements vary in time [50]. If, however, the computational loads changes gradually in time, then our methods can be used to re-partition the total load at certain time intervals. Another point is that some problems are more suitable to geometric partitioning methods; contact detection without a priori knowledge of the contacting surfaces, for example, should be performed on geometrically close elements [16, 61, 84]. Our methods, in their current forms, will probably be of little help in those problems. To be useful, the models should be enriched with some geometric constructs as is done in [61].

The multiphase mesh partitioning method of Walshaw et al. [73, 106] and multi-constraint/multi-objective graph partitioning methods of Karypis et al. [64, 90] also address the partitioning problem in the scientific computations mentioned above. These two works are built upon undirected graph model. Therefore, the limitations of the graph model exist in these formulations. For example, these two models can obtain only symmetric partitionings; we think that only the requirement of PAP^T–PMP^T, among those given in Table 5.1, can be met using these models. Other than the inherent limitations, there is another restriction in the multiphase mesh partitioning formulation of Walshaw et al. that only a subset of nodes should be active at a given computation phase. Within this respect, our methods and multi-constraint/multi-objective graph partitioning methods of Karypis et al. seem to be superior to multiphase mesh partitioning method in modeling different type of computations, where ours being unique in handling unsymmetric dependencies, in producing unsymmetric partitionings, and in modeling the total communication volume exactly.

5.6 Experiments

We chose the right preconditioned BiCGStab method to evaluate the proposed simultaneous partitioning method. We used a set of unsymmetric sparse matrices which were obtained from University of Florida Sparse Matrix Collection [32]. Approximate inverse preconditioners were obtained using SPAI version 3.0 [45]. Factored approximate inverses were obtained using AINV [13]. These two programs have parameters that affect the quality of the preconditioner matrices. However, we set the parameters in such a way that the number of nonzeros of the approximate inverse or the total number of nonzeros of the factors of the approximate inverse is at most twice and at least half of the number of nonzeros of the coefficient matrix. We adjusted the tolerance parameter eps, number of nonzero entries allowed per step mn, and the number of steps ns in SPAI. In AINV, we adjusted the drop tolerance parameter τ. The properties of the matrices, approximate inverses, and factors of the approximate inverses are given in Table 5.2. In the table, the coefficient matrices are listed with a suffix of A; the approximate inverse matrices are listed with a suffix of M; the factors of the approximate inverse matrices are listed with suffixes of Z and W, where approximate inverse is equivalent to ZW. The composite hypergraphs were partitioned using PaToH [22] with default parameters. The imbalance among processors' loads is kept below 10% in all partitioning instances. Throughout this section, we use "SPAI-matrices" to refer to a pair of a coefficient matrix and its approximate inverse preconditioner. Similarly, we use "AINV-matrices" to refer to a triplet of coefficient matrix and the factors of its approximate inverse preconditioner.

Since the partitioning tool PaToH incorporates randomized algorithms, it was

Table 5.2: Properties of test matrices.

Matrix	number of rows/cols	number of nonzeros total	average row/col	row min	row max	col min	col max
Zhao1-A	33861	166453	4.9	3	6	2	7
big-A	13209	91465	6.9	3	12	3	12
cage11-A	39082	559722	14.3	3	31	3	31
cage12-A	130228	2032536	15.6	5	33	5	33
epb2-A	25228	175027	6.9	3	87	3	87
epb3-A	84617	463625	5.5	3	6	3	7
mark3jac060-A	27449	170695	6.2	2	44	2	47
olafu-A	16146	1015156	62.9	24	89	24	89
stomach-A	213360	3021648	14.2	7	19	6	22
xenon1-A	48600	1181120	24.3	1	27	1	27
SPAI							
Zhao1-M	33861	180988	5.3	1	11	1	16
big-M	13209	109088	8.3	2	22	1	21
cage11-M	39082	424708	10.9	2	51	2	21
cage12-M	130228	1444650	11.1	1	62	2	21
epb2-M	25228	244453	9.7	2	177	2	21
epb3-M	84617	532851	6.3	2	20	2	20
mark3jac060-M	27449	276586	10.1	1	37	1	21
olafu-M	16146	719873	44.6	5	114	4	46
stomach-M	213360	2910283	13.6	2	120	2	46
xenon1-M	48600	878143	18.1	1	35	1	21
AINV; $M = ZW$							
Zhao1-Z	33861	179803	5.3	1	13	1	28
Zhao1-W	33861	57832	1.7	1	5	1	6
big-Z	13209	56302	4.3	1	11	1	13
big-W	13209	56314	4.3	1	13	1	11
cage11-Z	39082	302775	7.7	1	26	1	110
cage11-W	39082	299939	7.7	1	26	1	32
epb2-Z	25228	116161	4.6	1	13	1	22
epb2-W	25228	107620	4.3	1	36	1	19

run 20 times starting from different random seeds for partitioning composite and individual hypergraphs. Averages of the resulting communication patterns of these runs are displayed in the following tables. Although the main objective in the simultaneous partitioning method is the minimization of the total communication volume, the results for the total number of messages, the maximum volume and the maximum number of messages handled by a single processor are also given.

5.6.1 Composite versus individual hypergraph partitioning

In this section, we analyze how the proposed composite hypergraph partitioning models compare with the individual hypergraph partitioning models. We can use the individual hypergraph partitioning models in two different approaches.

The first approach is to obtain independent partitionings on the matrices by partitioning the computational hypergraph models of the coefficient and the preconditioner matrices. This approach requires vector reordering in between the two matrix-vector multiplies. We discuss this approach in §5.6.1.1.

The second approach is to use the same partition for the coefficient and preconditioner matrices. For this purpose, we partition the coefficient matrices by rows or columns and apply the resulting partitions to the preconditioner matrices as well. This approach disregards the sparsity pattern of the preconditioner matrices. However, the sparsity pattern of the approximate inverse preconditioners are related to the sparsity pattern of the coefficient matrices [27, 59]. Therefore, the partitions on the coefficient matrices are likely to induce effective partitions on the preconditioners as well. To justify this reasoning, we show the relation between the sparsity patterns of the coefficient matrices and the approximate inverses in Table 5.3. As seen in the table, the relation between the sparsity patterns of the coefficient and preconditioner matrices varies; almost half of the nonzeros of `Zhao1-M` are covered by the nonzeros of `Zhao1-A`, and only %12 of the nonzeros of `mark3jac060-M` are covered by the nonzeros of `mark3jac060`.

Table 5.3: The relation between the sparsity patterns of the coefficient matrices the approximate inverses. In this table, we use A and M to represent the set of the positions of the nonzeros in the corresponding matrices.

Matrix	number of nonzeros			
	$A \cup M$	$A \setminus M$	$M \setminus A$	$\frac{A \cap M}{A \cup M}$
Zhao1	234205	67752	53217	0.48
big	147632	56167	38544	0.36
cage11	780776	221054	356068	0.26
cage12	2784199	751663	1339549	0.25
epb2	333794	158767	89341	0.26
epb3	773107	309482	240256	0.29
mark3jac060	397706	227011	121120	0.12
olafu	1357370	342214	637497	0.28
stomach	5182305	2160657	2272022	0.14
xenon1	1520936	339816	642793	0.35

Another reason for using the same partitioning for the coefficient and preconditioner matrices is the following. Parallel construction of the approximate inverse preconditioners produces preconditioners in such a way that the initial partitions on the coefficient matrices become partitions on the preconditioner matrices. For example, the left approximate inverse preconditioners can be efficiently constructed rowwise when the coefficient matrix A is partitioned rowwise [28]. The construction yields the same rowwise partition on the approximate inverse M. Equivalently, a right approximate inverse preconditioner can be efficiently constructed columnwise when the coefficient matrix A is partitioned columnwise. We discuss using the same partitioning for the coefficient and preconditioner matrices in §5.6.1.2.

5.6.1.1 Independent partitioning on the matrices

For SPAI-matrices, we choose the partitioning dimensions as columnwise-rowwise (CR) and rowwise-columnwise (RC) for the A and M matrices in the given order.

Tables 5.4 and 5.5 display the average communication patterns of the simultaneous and the individual partitionings for SPAI-matrices. The tables also show the volume of communication required to reorder the vector entries—in an iteration of the BiCGStab method—when the matrices are partitioned individually. Suppose that symmetric partitionings PAP^T and QMQ^T were obtained on the A and M matrices. Then, for each iteration we have to reorder $\hat{p}$ and $\hat{s}$ from Q to P after the matrix-vector multiplies at lines 12 and 17 of the BiCGStab method (see Fig. 5.1), respectively. We also have to reorder v and t from P to Q before the vector update at line 15 and the inner product at line 19, respectively. The volume of communication in the reordering operation is obtained according to the permutation matrices that give the best volume for the individual matrices. The actual total volume of communication in the individual partitioning method can be obtained by adding the volume of reordering operations to the total volumes of the individual partitionings. In all of the partitioning instances, the volume of communication in the reordering operation itself is higher than the volume of communication in the simultaneous partitioning. These high volumes of communication and the associated message start-up overheads prohibit the use of the individual partitioning method. For example, the individual partitioning method incurs higher total communication volume than the proposed simultaneous partitioning method by factors that vary between 2.8 (`cage12`) and 24.6 (`epb3`) with an overall average factor of 8.6 for 32-way CR partitioning. The average factor in 64-way CR partitioning is 6.7. For RC partitioning, the average factors are 5.8 and 4.2 for 32- and 64-way partitionings, respectively.

Table 5.6 displays the averages of the communication patterns of the 32- and 64-way simultaneous and individual partitionings for AINV-matrices. Due to lack of space we give only the experiments in which the partitioning dimensions are chosen as columnwise-rowwise-columnwise (CRC) for the A, Z, and W matrices in the given order. For AINV-matrices, the individual partitioning method requires two additional reordering operations which are necessary for the chains of matrix-vector multiplies at lines 12 and 17 of the BiCGStab method. For AINV-matrices, the minimum ratio of the communication volumes in the individual partitionings (including the reordering cost) to those of the simultaneous partitionings is 2.7 which is obtained for the 64-way partitioning of the `cage11` matrix. The maximum ratio is 14.9 which is obtained for the 32-way

Table 5.4: Communication patterns for 32-way simultaneous and individual partitionings for SPAI-matrices.

	Simultaneous partitioning				Individual partitioning				
	Volume		Message		Volume		Message		Reorder
Matrix	tot	max	tot	max	tot	max	tot	max	Volume
	CR				C/R				
Zhao1-A	9419	453	248.1	13.2	8131	340	217.6	11.4	
Zhao1-M	7927	415	251.8	13.4	7174	334	250.8	13.8	135412
big-A	2455	128	162.1	8.9	2071	91	150.9	7.5	
big-M	2357	133	169.4	9.8	1946	91	155.8	7.8	52828
cage11-A	47979	2473	601.5	26.4	42424	2068	444.1	21.3	
cage11-M	29021	1515	640.8	27.2	24460	1189	495.0	23.3	148924
cage12-A	162030	8313	783.8	29.9	142434	6771	614.1	27.4	
cage12-M	93273	4783	815.0	30.4	76776	3519	660.2	28.8	488032
epb2-A	4846	317	233.8	15.8	4162	243	212.7	15.0	
epb2-M	4943	287	186.8	10.7	3918	188	161.8	9.4	100912
epb3-A	5938	315	168.1	9.2	3705	166	126.5	6.1	
epb3-M	7262	380	169.2	9.1	4478	203	141.9	7.3	317008
mark3jac060-A	13519	631	347.7	17.5	9735	377	266.7	12.1	
mark3jac060-M	14578	697	324.0	16.9	11648	460	298.3	14.2	109508
olafu-A	10390	672	155.2	9.2	8394	444	127.9	7.2	
olafu-M	18197	1180	197.6	11.2	15023	890	152.4	8.4	62312
stomach-A	34872	1864	187.8	10.4	26075	976	178.9	7.6	
stomach-M	41181	2022	193.7	10.7	30306	1221	152.6	7.1	853440
xenon1-A	21833	1085	291.4	14.5	19090	824	242.6	11.9	
xenon1-M	29525	1431	314.1	15.8	23634	1003	262.6	13.3	180032
	RC				R/C				
Zhao1-A	9815	564	245.8	12.8	7801	347	218.1	11.7	
Zhao1-M	10386	568	244.2	12.7	8326	386	234.7	13.2	135444
big-A	3536	214	185.4	10.1	2083	92	152.9	7.7	
big-M	5022	292	189.3	9.9	3521	173	161.9	8.2	52824
cage11-A	59783	3988	787.0	30.6	42539	1993	446.6	21.6	
cage11-M	46953	2263	776.5	29.4	31419	1483	557.5	25.4	150560
cage12-A	192970	10335	923.9	31.0	142734	6222	613.9	26.9	
cage12-M	141307	7303	915.5	31.0	95652	4397	722.0	29.8	511804
epb2-A	6919	516	331.0	19.8	3944	221	207.7	11.4	
epb2-M	7889	515	240.9	14.1	4823	232	173.6	9.9	100912
epb3-A	12771	1248	244.7	16.2	4840	204	143.3	7.5	
epb3-M	13461	1383	242.3	16.1	4720	218	141.4	7.5	337660
mark3jac060-A	14277	760	390.1	19.7	9693	394	327.8	15.8	
mark3jac060-M	15896	756	355.2	18.3	12589	524	311.7	14.9	109796
olafu-A	15169	999	209.7	13.1	8469	451	126.8	7.2	
olafu-M	23002	1325	259.0	15.6	14601	817	153.4	8.8	62648
stomach-A	53375	3853	225.2	13.3	26217	978	177.9	7.7	
stomach-M	62102	3981	230.0	13.7	32674	1329	161.1	7.5	853440
xenon1-A	26536	1398	349.9	19.1	19018	813	242.8	12.1	
xenon1-M	34745	1734	376.1	20.1	23484	983	264.2	12.8	191872

Table 5.5: Communication patterns for 64-way simultaneous and individual partitionings for SPAI-matrices.

	Simultaneous partitioning				Individual partitioning				
	Volume		Message		Volume		Message		Reorder
Matrix	tot	max	tot	max	tot	max	tot	max	Volume
	CR				C/R				
Zhao1-A	13026	327	592.2	17.0	11421	237	529.0	13.1	
Zhao1-M	10809	305	598.0	17.4	9857	234	583.8	16.4	135080
big-A	3666	98	336.5	9.2	3210	69	326.3	8.6	
big-M	3562	102	352.8	9.8	3054	70	339.4	8.8	52824
cage11-A	63779	1732	1585.7	39.5	58177	1463	1173.2	30.9	
cage11-M	38714	1249	1709.0	43.5	32937	835	1262.2	33.5	153784
cage12-A	207077	6111	2229.6	51.1	185531	4711	1626.4	40.4	
cage12-M	119287	3678	2366.2	52.5	98493	2552	1731.2	44.3	504628
epb2-A	6731	259	463.5	22.2	5984	172	439.5	18.6	
epb2-M	7215	211	381.2	12.4	5967	138	343.6	10.8	99904
epb3-A	8167	227	365.7	10.9	5713	130	298.1	7.5	
epb3-M	9759	282	370.2	11.1	6846	164	305.9	8.0	338468
mark3jac060-A	17447	498	945.7	24.3	13331	319	724.1	16.9	
mark3jac060-M	18970	533	923.2	25.4	15567	358	887.5	20.1	109788
olafu-A	16743	569	363.4	11.3	14012	356	294.3	8.2	
olafu-M	29348	1102	518.2	16.1	25137	696	399.6	11.2	63492
stomach-A	47689	1343	424.9	12.5	36800	706	371.3	8.0	
stomach-M	57755	1588	447.1	12.9	44232	966	391.1	8.7	853440
xenon1-A	29644	769	663.5	18.1	26710	593	542.4	15.3	
xenon1-M	40270	1066	744.2	20.8	33597	754	614.0	16.5	194380
	RC				R/C				
Zhao1-A	13734	399	619.0	17.0	10811	238	517.4	13.3	
Zhao1-M	14551	420	612.5	16.4	11756	280	568.7	15.4	135348
big-A	5453	197	410.0	12.8	3215	68	327.9	8.8	
big-M	7610	223	422.6	12.2	5447	140	347.1	9.1	52804
cage11-A	79052	3482	2265.8	56.0	58272	1452	1164.6	31.7	
cage11-M	61304	1627	2203.7	48.6	42512	1057	1470.5	38.8	151840
cage12-A	248253	8193	2959.2	60.6	185191	4217	1617.2	40.9	
cage12-M	180128	4976	2879.7	58.6	122513	3246	1991.8	48.8	510824
epb2-A	10610	515	766.9	29.9	5527	166	430.2	13.7	
epb2-M	11694	437	492.7	17.1	7411	181	353.9	11.5	100912
epb3-A	17911	985	525.7	19.5	7250	155	312.0	7.5	
epb3-M	18512	938	518.4	19.3	7057	172	295.6	7.9	333320
mark3jac060-A	19503	643	1094.3	29.6	12676	285	953.9	24.6	
mark3jac060-M	21096	525	996.0	24.7	16563	408	891.5	20.1	109780
olafu-A	23870	1041	532.2	18.8	13912	370	292.6	8.2	
olafu-M	36427	1071	696.5	21.8	24735	641	399.1	11.3	57372
stomach-A	77080	3239	552.2	18.7	37219	717	370.3	8.0	
stomach-M	89479	3418	574.5	18.8	47965	1028	391.2	8.9	853440
xenon1-A	36044	1049	808.0	24.5	26669	594	545.3	14.8	
xenon1-M	46970	1183	904.5	25.9	33241	729	604.0	15.1	178388

Table 5.6: Communication patterns for 32- and 64-way simultaneous and individual partitionings for AINV-matrices.

	Simultaneous partitioning				Individual partitioning				
	CRC				C/R/C				
	Volume		Message		Volume		Message		Reorder
Matrix	tot	max	tot	max	tot	max	tot	max	Volume
	$K = 32$								
Zhao1-A	11191	515	261.2	13.4	8131	340	217.6	11.4	
Zhao1-Z	9132	476	278.6	14.8	7877	1134	293.9	23.4	
Zhao1-W	1711	108	211.4	11.2	76	11	17.8	2.1	198860
big-A	2443	113	157.5	8.4	2071	91	150.9	7.5	
big-Z	1486	85	150.7	8.4	1217	63	146.7	7.3	
big-W	1496	80	149.1	8.2	1218	60	147.8	7.2	76250
cage11-A	49562	2381	508.1	23.9	42424	2068	444.1	21.3	
cage11-Z	21612	1200	545.5	25.9	16277	1119	354.2	19.6	
cage11-W	20323	1050	537.9	25.1	16127	808	336.9	16.9	220128
epb2-A	7028	393	470.1	27.1	4162	243	212.7	15.0	
epb2-Z	2637	174	174.2	9.8	1395	131	105.8	8.6	
epb2-W	2454	162	168.3	9.1	923	75	116.7	9.6	147538
	$K = 64$								
Zhao1-A	15258	365	635.0	17.9	11421	237	529.0	13.1	
Zhao1-Z	12811	332	698.4	20.7	10808	1630	676.9	38.1	
Zhao1-W	2494	84	464.9	13.7	170	13	43.5	2.6	198614
big-A	3730	91	343.9	10.4	3210	69	326.3	8.6	
big-Z	2262	71	323.2	9.4	1861	50	309.8	8.8	
big-W	2305	68	319.4	9.4	1859	49	309.7	7.8	77630
cage11-A	65430	1828	1276.5	32.5	58177	1463	1173.2	30.9	
cage11-Z	28640	925	1367.2	40.9	22023	956	799.8	26.9	
cage11-W	27397	766	1351.0	36.4	21676	572	743.6	21.6	227770
epb2-A	10313	328	1050.8	39.2	5984	172	439.5	18.6	
epb2-Z	3967	152	372.7	11.8	2058	121	233.9	9.3	
epb2-W	3786	143	349.3	10.3	1431	71	209.4	12.8	150522

partitioning of the `big` matrix. The average of the ratios for 32- and 64-way partitionings are 10.1 and 7.2, respectively. The extremely high communication volumes introduced by the reordering operations again prohibit the application of the individual partitioning method. The minimum and maximum ratios in the 32- and 64-way RCR partitionings are sightly better than those in the CRC case (2.21 and 13.0) implying a slightly better average (8.6 and 6.5 for 32- and 64-way partitionings, respectively) but still not tolerable. Details can be found in [100].

Consider the difference between the total communication volumes of the simultaneous and individual partitionings (without the reordering cost). The increases in the total communication volume values for the simultaneous partitionings remain below 26% of those of the individual partitionings, on the average, for the 32-way CR partitioning instances. The minimum and the maximum of these increases are 13% (`Zhao1`) and 61% (`epb3`). The 64-way CR partitionings give better ratios. The average increase is 20% with the minimum and the maximum being 12% and 43% which are obtained for the same matrices. In fact, for each matrix the 64-way CR partitioning gives smaller percent increase than the 32-way CR partitioning. We investigated the 8- and 16-way CR partitionings as well (see [100]) and observed that for each matrix in our data set the larger the number of parts, the smaller the percent increases. The same relation holds for 8-, 16-, 32-, and 64-way RC partitioning case, except for the 32- and 64-way partitionings of the `epb2` and `mark3jac060` matrices. It also holds for most of the CRC and RCR partitioning of AINV-matrices. Details can be found in [100]. The reason behind this may be the following. The cutsize function almost always increases monotonically with the increasing K. In other words, the flexibility of finding better partitions reduces with the increasing K. At the limit, where $K = \mathcal{V}$ and all the nets are in cut, the cutsize of a composite hypergraph will be equivalent to the sum of the cutsizes of the individual hypergraphs (i.e., $nnz(A) + nnz(M) - 2m$) that forms it. Therefore, the difference between the total communication volumes has to converge to zero.

5.6.1.2 Using the same partitioning

Partitioning a coefficient matrix and then applying the resulting partition to the preconditioner matrix results in columnwise-columnwise (CC) or rowwise-rowwise (RR) partitioning on the A and M matrices. Recall that for CC and RR partitioning schemes, there is a communication phase in between the two matrix-vector multiplies. Since the A and M matrices have comparable number of nonzeros (see Table 5.2), processors' loads for the two matrix-vector multiplies should be balanced separately, i.e., a two-constraint formulation is necessary.

The individual row-net hypergraph model of A can be used to obtain a CC partitioning on A and M. In order to obtain load balance for the two multiplies, the vertices of A are assigned two weights which correspond to the number of nonzeros in the respective columns of A and M matrices. That is, v_i has weights $\langle |c_i(A)|, |c_i(M)| \rangle$. Similarly, the column-net hypergraph model of A, with two weights on the vertices, can be used to obtain an RR partitioning on A and M.

The composite hypergraph model for the CC partitioning scheme is built by creating row-net hypergraph models of A and M, applying pin addition operation between net $r_i(A)$ and vertex $c_i(M)$ and also between net $r_i(M)$ and vertex $c_i(A)$ for each i, and applying vertex weighting operation in such a way that the vertices coming from A have weights $\langle |c_i(A)|, 0 \rangle$ and the vertices coming from M have weights $\langle 0, |c_i(M)| \rangle$. The composite hypergraph model for the RR partitioning scheme is built similarly by interchanging the roles of rows and columns.

Tables 5.7 and 5.8 display the averages of the communication patterns of the 32- and 64-way partitioning of the SPAI-matrices with the composite and individual hypergraph models. The right most columns in these tables show the improvements achieved by the composite hypergraph partitioning as the percentage of the total communication volumes found by partitioning the individual hypergraph of A. The minimum percent improvements are obtained for the `Zhao1` matrices in all cases. The maximum percent improvements are obtained for the `mark3jac060` matrices in all cases. As seen in Table 5.3, the `Zhao1` matrices have the highest number of common nonzeros, and the `mark3jac060` matrices have the least number of common nonzeros. Although, the `stomach` matrices have %14 common nonzeros (second minimum) the improvements achieved for these matrices are almost half of the those obtained for the `mark3jac060` matrices. The average of the improvements is %20 in Tables 5.7 and 5.8 both for CC and RR partitioning choices.

We have also experimented with the 32- and 64-way, CC and RR partitionings using single constraint formulation. In the single constraint formulation, the weight of a vertex v_i is set to the sum of the number of nonzeros in the ith columns of A and M for the CC partitioning. Both the composite hypergraph formulation and the individual hypergraph formulation were able to obtain balance on the total loads of the processors. Both formulations could not obtain balance

Table 5.7: Communication patterns for 32-way CC and RR composite and individual hypergraph partitionings for SPAI-matrices.

	Individual partitioning				Simultaneous partitioning				
	Volume		Message		Volume		Message		Percent
Matrix	tot	max	tot	max	tot	max	tot	max	Gain
	CC								
Zhao1-A	9228	385	205.0	11.0	9256	392	217.6	11.2	7
Zhao1-M	10847	489	207.8	11.0	9331	420	217.4	11.2	
big-A	2691	128	177.1	9.4	2519	136	161.8	8.4	19
big-M	5358	256	188.3	10.0	4016	189	169.7	8.9	
cage11-A	49634	2138	554.0	23.8	48096	2147	545.5	24.6	19
cage11-M	55307	2482	663.1	27.8	36739	1875	534.7	24.4	
cage12-A	164456	7199	805.3	29.9	161042	7224	736.4	28.6	18
cage12-M	172098	7725	885.2	30.9	114446	6103	718.6	29.8	
epb2-A	6410	317	382.4	19.1	6259	313	341.4	18.1	15
epb2-M	8620	410	220.7	11.3	6587	368	215.4	12.4	
epb3-A	8169	555	199.3	11.8	7851	519	190.5	11.1	25
epb3-M	14290	850	202.5	11.8	8992	577	193.8	11.4	
mark3jac060-A	11155	471	310.8	16.6	13146	518	318.1	15.1	31
mark3jac060-M	29590	1241	303.2	16.8	15158	644	290.8	14.6	
olafu-A	12455	670	174.3	10.2	10275	593	130.7	7.2	28
olafu-M	24120	1307	240.2	13.8	15905	930	162.2	9.8	
stomach-A	34714	1596	221.8	11.7	40865	2618	213.5	11.4	16
stomach-M	72997	3376	233.7	12.1	49987	3099	216.2	11.5	
xenon1-A	21809	933	265.8	14.1	19571	829	252.8	12.4	22
xenon1-M	35264	1513	306.7	15.8	25116	1144	269.6	13.8	
	RR								
Zhao1-A	8829	372	201.1	10.0	8697	381	224.1	11.9	6
Zhao1-M	8787	383	203.4	10.1	7871	356	224.6	12.2	
big-A	2682	126	178.4	9.7	2484	127	165.4	8.4	20
big-M	3641	170	191.1	10.1	2543	131	169.1	8.8	
cage11-A	50521	2179	571.8	25.1	46893	2621	506.9	23.6	19
cage11-M	45057	1935	684.1	28.0	30468	1390	531.1	24.2	
cage12-A	166189	7185	790.0	29.8	156384	8709	696.9	30.1	18
cage12-M	142088	6160	875.0	30.9	96906	4172	704.6	28.6	
epb2-A	6186	371	365.1	20.2	5817	470	297.9	17.4	17
epb2-M	8009	409	223.9	12.2	5939	341	209.6	11.4	
epb3-A	8967	678	220.0	12.8	8376	593	187.7	11.2	21
epb3-M	12786	836	224.5	13.2	8788	592	193.9	11.2	
mark3jac060-A	12851	526	483.6	24.3	11793	501	344.6	16.4	36
mark3jac060-M	26559	1158	507.2	26.1	13569	566	323.0	15.8	
olafu-A	12762	719	172.5	10.6	9846	598	132.2	8.2	29
olafu-M	24535	1239	237.9	13.9	16657	888	160.4	8.7	
stomach-A	36049	1718	217.2	11.2	40160	2652	215.1	11.2	13
stomach-M	62622	2857	229.5	11.9	45415	3101	218.8	11.6	
xenon1-A	20993	904	249.5	12.4	19683	907	249.3	12.8	17
xenon1-M	33878	1427	288.9	14.8	25768	1108	265.9	13.8	

Table 5.8: Communication patterns for 64-way CC and RR composite and individual hypergraph partitionings for SPAI-matrices.

	Individual partitioning				Simultaneous partitioning				
	Volume		Message		Volume		Message		Percent
Matrix	tot	max	tot	max	tot	max	tot	max	Gain
	CC								
Zhao1-A	12982	271	543.7	13.8	12892	283	558.2	14.6	8
Zhao1-M	15406	359	550.6	13.8	13116	311	560.6	14.5	
big-A	4089	101	378.0	11.3	3849	109	332.8	9.3	18
big-M	8082	204	412.9	12.5	6114	154	358.9	10.3	
cage11-A	67198	1610	1511.1	36.9	64928	1641	1456.4	36.9	18
cage11-M	72710	1803	1886.3	43.7	49361	1362	1410.2	37.2	
cage12-A	213360	5079	2249.8	49.0	208209	5054	2004.5	44.9	18
cage12-M	218227	5396	2659.7	54.9	146726	4357	1904.1	48.6	
epb2-A	9820	268	907.0	27.4	9357	263	782.1	27.9	16
epb2-M	12913	335	491.7	15.8	9766	314	457.6	15.0	
epb3-A	11764	483	508.9	17.0	11293	414	425.0	13.1	26
epb3-M	21010	761	521.2	18.1	13063	443	427.1	13.8	
mark3jac060-A	15676	367	900.8	24.3	18090	380	911.1	22.8	34
mark3jac060-M	42608	954	1009.5	28.2	20183	506	894.6	22.9	
olafu-A	19802	562	402.3	13.0	16733	476	321.9	8.8	26
olafu-M	38318	1102	653.0	20.2	26470	731	431.6	13.4	
stomach-A	50980	1239	488.4	13.6	57777	1916	488.6	13.4	19
stomach-M	107817	2664	536.4	15.2	71060	2310	501.3	13.8	
xenon1-A	30510	671	608.3	17.1	27683	601	567.9	14.9	21
xenon1-M	49568	1111	753.9	20.7	35520	836	624.6	16.8	
	RR								
Zhao1-A	12502	269	541.0	13.9	12131	284	557.6	14.8	8
Zhao1-M	12523	277	550.4	14.2	10896	251	561.4	14.7	
big-A	4068	98	377.6	11.6	3783	104	347.1	10.3	20
big-M	5548	138	414.2	12.6	3904	112	359.5	10.8	
cage11-A	68374	1607	1536.0	37.6	63612	1877	1324.9	35.8	19
cage11-M	60385	1384	1932.8	44.7	41030	974	1376.5	34.2	
cage12-A	216501	4888	2248.7	48.3	203197	6290	1857.8	49.2	18
cage12-M	183548	4205	2670.7	54.1	125836	2979	1887.8	43.1	
epb2-A	9611	367	873.5	30.4	8545	402	640.8	25.4	19
epb2-M	11649	308	484.2	16.1	8590	257	431.9	14.3	
epb3-A	12441	519	468.8	15.2	11607	434	396.2	12.7	22
epb3-M	18036	637	480.6	15.5	12201	420	410.1	12.8	
mark3jac060-A	18107	426	1396.2	40.0	15891	399	1016.0	26.2	36
mark3jac060-M	34929	899	1483.6	46.8	18005	420	960.5	25.2	
olafu-A	20273	585	410.3	12.8	16173	478	325.9	10.2	26
olafu-M	39026	982	642.5	20.0	27911	724	435.4	12.1	
stomach-A	52582	1301	484.9	12.8	58230	2007	486.5	13.1	13
stomach-M	90605	2103	532.9	14.0	66561	2253	502.3	13.2	
xenon1-A	29597	660	590.9	16.4	27615	648	559.5	15.1	17
xenon1-M	47909	1047	729.2	20.1	36437	797	620.5	16.1	

on the loads of the processors for the individual matrix-vector multiplies, since these formulations ignore the fact that there is a local synchronization between the two multiply operations. The composite hypergraph partitioning approach again obtained better solutions than the individual hypergraph partitioning. The best and worst improvements are again obtained for the `mark3jac060` and `Zhao1` matrices. The average of the improvements is %17. See [100] for the details.

5.6.2 Effects of partitioning dimensions on the simultaneous partitioning

Comparing the lower and upper halves of the Tables 5.4 and 5.5, we see that CR partitioning scheme yields better total communication volume than the RC scheme. The ratio of the average total communication volume in the CR partitioning to that in the RC partitioning is around 0.74 for the data given in the Tables 5.4 and 5.5. This ratio remains the same for the 8- and 16-way partitionings given in the Table 5.9. The standard deviation of these ratios is around 0.13 for each $K = 8, 16, 32, 64$. Note that the matrices in our data set do not have dense rows or dense columns. Therefore, it is expected that the rowwise and columnwise partitionings of the matrices will result in comparable results. This theoretical expectation is verified by the data given in the total communication volume column under the individual partitioning header in Tables 5.4 and 5.5. In the light of this observation, we can deduce that the performance difference between the CR and RC partitioning schemes is mainly due to the two-constraint formulation in the RC scheme. This degradation in the multi-constraint formulation is in concordance with the previously reported results [64, 106]. The degradation in our case stems from two facts. First, the additional balance constraints shrink the search space. Second, the heuristics in PaToH are not very well tailored toward handling the multiple vertex weights.

5.6.3 Parallelization results

It is important to see whether the theoretical improvements obtained by the proposed simultaneous partitioning method hold in practice. For this purpose, we have implemented a parallel program for the BiCGStab method. The program uses LAM/MPI 6.5.6 [18] message passing library. The tests were carried out on a Beowulf class [94] PC cluster with 24 nodes. Each node has a 400MHz Pentium-II processor and 128 MB memory. The interconnection network is comprised of a 3COM Superstack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node. The system runs Linux kernel 2.4.20 and the Debian GNU/Linux 3.0 distribution.

We are not concerned with the numerics of the preconditioners and the BiCGStab method. Therefore, for each matrix we let the BiCGStab run for 100 iterations and measure the average running time of a single iteration. In order to guarantee 100 iterations, we set ρ and ω of the BiCGStab method (see Fig. 5.1) to 1.0 after computing their actual values. The speed up values corresponding to these running times are given in Table 5.9 under the column *Sp./up*. Note that we shortened the matrix name `mark3jac060` to `mark3_060` to fit the table into the page. The given speed up values are the averages of 20 runs corresponding to different partitionings. In order to show how the improvements obtained by the proposed method relate to parallel running times, we give the average communication patterns of the partitionings in same table as well.

As seen from Table 5.9, the CR partitioning gives better speedup values than the RC partitioning for all matrices. On the average, CR obtains speedup values of 6.3 and 9.9 for 8- and 16-way partitionings, respectively, where the highest speedups are 7.3 (`epb3`) and 14.1 (`stomach`). Meanwhile, RC obtains speedups of 5.8 and 8.4, on the average, for 8-way and 16-way partitionings, respectively, where the the highest speedups are 7.2 (`epb3`) and 12.7 (`stomach`). The worst speedups for 8-way partitioning are obtained for the `cage11` matrix by both of the partitioning schemes. The worst speedups for 16-way partitioning are obtained for the `big` and `cage11` matrices by the CR and RC schemes, respectively. As seen from Table 5.9, the `cage11` matrix-pair has inferior communication pattern than all but the `cage12` matrix-pair in terms of the total and maximum number of messages metrics. Therefore, we were already expecting to have the worst speedups with the `cage` matrices. The `big` matrix has the smallest number of nonzeros. This low granularity of computations may be the reason behind having the worst speedup with 16-way CR partitioning of the `big` matrix. The same reasoning may also explain why we obtain better speedups in `cage12` than those in `cage11`.

We have also experimented with the CRC and RCR partitioning schemes for AINV-matrices (see [100]). The speedup values are not as good as those given in Table 5.9 as expected, because of the three load balance constraints (see Fig. 5.2(e)) and more communication phases. The best speedups for 8- and 16-way CRC partitionings are 6.7 and 8.9, respectively. The best speedups for 8- and 16-way RCR partitionings are 6.4 and 8.9, respectively.

5.6.4 Partitioning timings

Lastly, we comment on the additional partitioning overhead introduced by simultaneous partitioning instead of individual partitionings. Let the sum of the times elapsed in individual partitionings (of the SPAI- and AINV-matrices) be 1.0. Then, the average running times of the simultaneous partitioning of the SPAI-matrices with the CR and RC schemes are 1.4 and 1.2, respectively, for all $K = 8, 16, 32$, and 64. The average running times of the simultaneous partitioning of the AINV-matrices with both the CRC and RCR schemes are close to 1.4. These increases are acceptable because the simultaneous partitioning method obtains much smaller total communication volume than the individual partitioning method combined with the reordering cost. Timings for the CR and RC partitionings for SPAI-matrices are given in Tables 5.10 and 5.11. Timings for the CRC and RCR partitioning schemes of AINV-matrices are given in Tables 5.12 and 5.13.

Table 5.9: Communication patterns for 8- and 16-way simultaneous partitionings for SPAI-matrices and the respective speed up values.

	8-way					16-way				
	Volume		Message		Sp.	Volume		Message		Sp.
Matrix	tot	max	tot	max	up	tot	max	tot	max	up
					CR					
Zhao1-A	4098	746	32.2	5.7	6.2	6444	586	96.7	9.3	8.7
Zhao1-M	3514	694	32.2	5.6		5478	551	97.0	9.3	
big-A	1032	201	31.4	5.7	5.7	1581	156	73.2	7.5	7.3
big-M	989	191	31.9	5.6		1527	150	75.3	7.5	
cage11-A	24424	4144	54.6	7.0	5.5	34835	3314	201.2	14.8	8.1
cage11-M	14663	2439	55.1	7.0		21010	1917	208.9	15.0	
cage12-A	87542	14306	56.0	7.0	5.9	122878	11925	230.3	15.0	9.4
cage12-M	50962	7839	56.0	7.0		71066	6136	233.1	15.0	
epb2-A	2326	429	39.0	6.4	6.4	3357	371	102.8	9.6	8.6
epb2-M	2242	438	35.0	6.5		3335	335	84.7	8.4	
epb3-A	2354	442	23.9	4.3	7.3	3971	393	66.0	6.5	12.4
epb3-M	3003	536	23.9	4.3		5023	496	66.3	6.5	
mark3_060-A	5249	960	35.2	6.3	5.8	9370	786	115.0	11.3	8.7
mark3_060-M	6323	1182	32.2	6.0		10287	964	105.3	11.0	
olafu-A	3908	960	25.8	5.0	6.7	6489	781	66.2	6.8	10.6
olafu-M	6749	1449	28.0	5.4		11258	1285	77.8	7.8	
stomach-A	14614	2815	21.1	4.0	7.1	24436	2351	67.2	7.0	14.1
stomach-M	16193	3206	21.4	4.0		28014	2652	67.8	7.1	
xenon1-A	10848	2037	36.2	6.5	6.7	15998	1496	113.2	11.3	11.2
xenon1-M	14437	2523	37.7	6.7		21459	2032	117.8	11.8	
					RC					
Zhao1-A	4146	905	33.2	5.8	5.9	6728	734	95.5	9.3	8.1
Zhao1-M	4362	771	33.1	5.6		7141	716	95.5	9.3	
big-A	1467	320	33.9	5.8	5.2	2408	280	84.7	8.8	6.4
big-M	2084	433	34.0	5.8		3326	366	85.9	8.4	
cage11-A	30373	6054	55.9	7.0	4.2	43335	4599	227.0	15.0	5.5
cage11-M	24447	4302	55.9	7.0		34339	3151	227.4	15.0	
cage12-A	107187	17621	56.0	7.0	4.4	147504	12516	239.8	15.0	6.2
cage12-M	80949	15047	56.0	7.0		109472	10885	239.7	15.0	
epb2-A	3074	646	46.7	6.8	6.1	4555	560	132.1	12.9	8.6
epb2-M	3789	814	43.8	6.5		5388	633	109.5	10.6	
epb3-A	6347	1907	39.8	6.8	7.2	8490	1244	108.3	11.7	11.7
epb3-M	6766	1937	39.5	6.8		8991	1480	108.0	11.8	
mark3_060-A	5568	981	40.7	6.7	5.7	9792	906	137.4	13.0	7.4
mark3_060-M	6417	1213	38.6	6.8		11099	1027	126.3	12.2	
olafu-A	5605	1088	30.5	5.8	6.0	9325	1012	86.0	9.1	8.6
olafu-M	9075	1947	33.5	6.2		14636	1636	99.8	10.2	
stomach-A	21139	4354	27.4	5.3	7.1	35856	4255	81.5	8.8	12.7
stomach-M	24538	5221	27.6	5.7		41254	4660	82.8	8.9	
xenon1-A	12654	2322	39.6	6.9	6.1	18800	1797	126.2	12.2	9.2
xenon1-M	16486	2974	40.8	7.0		24477	2286	131.6	12.9	

Table 5.10: Average CR partitioning times for the SPAI-matrices in seconds.

K	Matrix	Individual partitioning		Simultaneous partitioning	Ratio
		A	M		
8	Zhao	2.28	2.00	6.76	1.6
	big	0.69	0.80	2.56	1.7
	cage11	7.23	5.19	15.64	1.3
	cage12	34.35	22.78	65.08	1.1
	epb2	1.24	1.73	4.41	1.5
	epb3	3.58	4.54	12.67	1.6
	mark3jac060	1.88	2.21	5.64	1.4
	olafu	4.81	5.11	12.19	1.2
	stomach	22.98	25.36	63.52	1.3
	xenon1	6.49	8.54	17.06	1.1
16	Zhao	2.86	2.57	8.66	1.6
	big	0.92	1.03	3.20	1.6
	cage11	9.23	6.50	19.86	1.3
	cage12	43.88	28.90	83.61	1.1
	epb2	1.63	2.16	5.72	1.5
	epb3	4.81	5.99	16.70	1.5
	mark3jac060	2.40	2.85	7.38	1.4
	olafu	6.32	6.73	15.87	1.2
	stomach	30.66	33.38	84.00	1.3
	xenon1	8.75	9.76	23.33	1.3
32	Zhao	3.70	3.08	10.17	1.5
	big	1.07	1.23	3.77	1.6
	cage11	10.93	7.57	23.66	1.3
	cage12	52.27	33.82	100.56	1.2
	epb2	2.04	2.71	6.91	1.5
	epb3	5.93	7.23	20.54	1.6
	mark3jac060	2.89	3.58	8.85	1.4
	olafu	7.71	8.54	19.35	1.2
	stomach	37.86	40.91	104.06	1.3
	xenon1	10.58	11.72	28.50	1.3
64	Zhao	4.02	3.56	11.58	1.5
	big	1.33	1.52	4.36	1.5
	cage11	12.61	8.74	27.02	1.3
	cage12	58.87	43.86	116.01	1.1
	epb2	2.43	3.18	8.05	1.4
	epb3	6.85	8.64	23.95	1.5
	mark3jac060	3.29	4.17	10.22	1.4
	olafu	9.33	10.28	22.70	1.2
	stomach	46.33	48.32	123.63	1.3
	xenon1	12.45	13.80	33.21	1.3

Table 5.11: Average RC partitioning times for the SPAI-matrices in seconds.

K	Matrix	Individual partitioning		Simultaneous partitioning	Ratio
		A	M		
8	Zhao	2.24	2.34	5.90	1.3
	big	0.71	0.95	2.30	1.4
	cage11	7.27	5.99	14.89	1.1
	cage12	34.70	26.32	62.06	1.0
	epb2	1.25	1.73	4.04	1.4
	epb3	3.98	4.59	11.54	1.3
	mark3jac060	1.79	2.39	4.90	1.2
	olafu	4.85	5.30	11.54	1.1
	stomach	22.99	25.21	57.59	1.2
	xenon1	6.69	7.34	16.07	1.1
16	Zhao	2.86	3.04	7.51	1.3
	big	0.88	1.14	2.93	1.4
	cage11	9.08	7.46	19.05	1.2
	cage12	43.84	32.48	79.21	1.0
	epb2	1.65	2.32	5.23	1.3
	epb3	5.24	6.06	15.17	1.3
	mark3jac060	2.28	3.04	6.40	1.2
	olafu	6.33	7.01	14.93	1.1
	stomach	30.55	32.97	75.71	1.2
	xenon1	8.64	9.75	21.38	1.2
32	Zhao	3.30	3.62	8.88	1.3
	big	1.09	1.46	3.50	1.4
	cage11	10.83	8.92	22.76	1.2
	cage12	51.80	38.05	95.57	1.1
	epb2	1.99	2.81	6.25	1.3
	epb3	6.45	7.31	18.59	1.4
	mark3jac060	2.74	3.86	7.71	1.2
	olafu	7.77	8.68	18.19	1.1
	stomach	38.10	40.81	93.57	1.2
	xenon1	10.69	11.68	26.44	1.2
64	Zhao	3.77	4.16	10.27	1.3
	big	1.34	1.69	4.13	1.4
	cage11	12.59	10.20	26.17	1.1
	cage12	59.33	43.34	110.08	1.1
	epb2	2.39	3.31	7.33	1.3
	epb3	7.55	8.69	21.73	1.3
	mark3jac060	3.25	4.45	9.03	1.2
	olafu	9.32	10.46	21.43	1.1
	stomach	44.92	48.42	117.00	1.3
	xenon1	12.48	13.80	30.87	1.2

Table 5.12: Average CRC partitioning times for the AINV-matrices in seconds.

K	Matrix	Individual partitioning			Simultaneous partitioning	Ratio
		A	Z	W		
8	Zhao	2.28	1.71	0.60	6.44	1.4
	big	0.69	0.50	0.47	2.37	1.4
	cage11	7.23	3.52	3.51	16.61	1.2
	epb2	1.24	1.01	0.79	4.41	1.5
16	Zhao	2.86	2.28	0.77	8.09	1.4
	big	0.92	0.61	0.60	2.99	1.4
	cage11	9.23	4.48	4.40	21.22	1.2
	epb2	1.63	1.20	1.01	5.63	1.5
32	Zhao	3.70	2.75	0.90	9.57	1.3
	big	1.07	0.74	0.79	3.64	1.4
	cage11	10.93	5.31	5.35	25.16	1.2
	epb2	2.04	1.51	1.21	6.79	1.4
64	Zhao	4.02	3.28	1.01	11.08	1.3
	big	1.33	0.84	0.89	4.27	1.4
	cage11	12.61	6.29	6.21	29.25	1.2
	epb2	2.43	1.68	1.46	7.98	1.4

Table 5.13: Average RCR partitioning times for the AINV-matrices in seconds.

K	Matrix	Individual partitioning			Simultaneous partitioning	Ratio
		A	Z	W		
8	Zhao	2.24	2.30	0.59	7.40	1.4
	big	0.71	0.54	0.46	2.308	1.3
	cage11	7.27	4.78	4.68	19.17	1.1
	epb2	1.25	1.01	0.84	4.59	1.5
16	Zhao	2.86	2.85	0.73	9.34	1.5
	big	0.88	0.62	0.62	2.94	1.4
	cage11	9.08	5.80	5.86	24.32	1.2
	epb2	1.65	1.31	1.06	5.91	1.5
32	Zhao	3.30	3.53	0.83	11.19	1.5
	big	1.09	0.75	0.76	3.53	1.4
	cage11	10.83	6.99	6.90	29.16	1.2
	epb2	1.99	1.58	1.30	7.16	1.5
64	Zhao	3.77	4.10	1.01	12.93	1.5
	big	1.34	0.89	0.91	4.17	1.3
	cage11	12.59	8.01	7.89	33.36	1.2
	epb2	2.39	1.82	1.54	8.44	1.5

Chapter 6

Message ordering

We consider a certain class of parallel program segments in which the order of messages sent affects the completion time. We give characterization of these parallel program segments and propose a solution to minimize the completion time. With a sample parallel program, we experimentally evaluate the effect of the solution on a PC cluster.

6.1 Introduction

We consider a certain class of parallel program segments with the following characteristics. First, there is a small-to-medium grain computation between two communication phases which are referred to as pre- and post-communication phases. Second, local computations cannot start before the pre-communication phase ends, and the post-communication phase cannot start before the computation ends. Third, the communication in both phases is irregular and sparse. That is, the communications are performed using point-to-point send and receive operations, where the sparsity refers to small number of messages having small sizes. These traits appear, for example, in the sparse-matrix vector multiply $y \leftarrow Ax$, where matrix A is partitioned on the nonzero basis and also in the sparse matrix-chain-vector multiply $y \leftarrow ABx$, where matrix A is partitioned along columns and matrix B is partitioned conformably along rows. In both examples, the x-vector entries are communicated just before the computation and the y-vector entries are communicated just after the computation.

There has been a vast amount of research in partitioning sparse matrices to effectively parallelize scientific computations by achieving computational load balance and by minimizing the communication overhead [21, 23, 24, 51, 52]. As noted in [51], most of the existing methods consider minimization of the total message volume. Depending on the machine architecture and problem characteristics, communication overhead due to message latency may be a bottleneck as well [35]. Furthermore, the maximum message volume and latency handled by a single processor also have crucial impact on the parallel performance as shown in [97, 99] and Chapters 3 and 4. However, optimizing these metrics is not sufficient to minimize the total completion time of the subject class of parallel programs. Since the phases do not overlap, the receiving time of a processor, and hence the issuing time of the corresponding send operation play an important role in the total completion time.

There may be different solutions to the above problem. One may consider balancing the number of messages per processor both in terms of sends and receives. This strategy would then has to partition the computations with the objectives of achieving computational load balance, minimizing total volume of messages, minimizing total number of messages, and also balancing the number of messages sent/received on the per processor basis. However, combining these objectives into a single function to be minimized would challenge the current state of the art. For this reason, we take these problems apart from each other and decompose the overall problem into stages, each of which involving a certain objective. We first use standard models to minimize the total volume of messages and maintain the computational load balance across processors using effective methods, such as hypergraph partitioning [21]. Then, we minimize the total number of messages and maintain a loose balance on the communication volume loads of processors, and in the meantime we address the minimization of the maximum number of messages sent by a single processor [97, 99]. After this stage, the communication pattern is determined. In this chapter, we suggest to append one more stage in which the send operations of processors are ordered to address the minimization of the total completion time.

6.2 Message ordering problem and a solution

We make the following assumptions. The computational load imbalance is negligible. All processors begin the pre-communication phase at the same time because of the possible global synchronization points and balanced computations that exist in the other parts of the parallel program. The parallel system has a high latency overhead so that the message transfer time is dominated by the start-up cost due to small message volumes. By the same reasoning, the receive operation is assumed to incur negligible cost to the receiving processor. For the sake of simplicity, the send operations are assumed to take unit time. Under these assumptions, once a send is initiated by a processor at time t_i, the sending processor can continue with some other operation at time t_{i+1}, and the receiving processor receives the message at time t_{i+1}. This assumption extends to concurrent messages destined for the same processor. The rationale behind these assumptions is that, the start-up costs for all messages destined for a certain processor truly overlap with each other.

Let *send-lists* $S_1(p)$ and $S_2(p)$ denote the set of messages, distinguished by the ranks of the receiving processors, to be sent by processor P_p in the pre- and post-communication phases, respectively. For example, $\ell \in S_1(p)$ denotes the fact that processor P_ℓ will receive a message from P_p in the pre-communication phase. For $\ell \in S_1(p)$, we use $s_1(p, \ell)$ to denote the completion time of the message from P_p to P_ℓ, i.e., P_p issued the send at time $s_1(p, \ell) - 1$, and P_ℓ received the message at time $s_1(p, \ell)$. We use $s_2(p, \ell)$ for the same purpose for the post-communication phase. Let W be the amount of computation performed by each processor. Let

$$r_1(p) = \max_{j:p \in S_1(j)} \{s_1(j, p)\} \tag{6.1}$$

denote the point in time at which processor P_p receives its latest message in the pre-communication phase. Then, P_p will enter the computation phase at time

$$c_1(p) = \max\{|S_1(p)|, r_1(p)\}, \tag{6.2}$$

i.e., after sending all of its messages and receiving all messages destined for it in the pre-communication phase. Let

$$r_2(p) = \max_{j:p\in S_2(j)}\{s_2(j,p)\} \tag{6.3}$$

denote the point in time at which processor P_p receives its latest message in the post-communication phase. Then, processor P_p will reach completion at time

$$c_p = \max\{c_1(p) + W + |S_2(p)|, r_2(p)\}, \tag{6.4}$$

i.e., after completing its computational task as well as all send operations in the post-communication phase and after receiving all post-communication messages destined for it. Using the above notation, our objective is

$$minimize\{\max_p\{c_p\}\}, \tag{6.5}$$

i.e., to minimize the maximum completion time. The maximum completion time induced by a message order is called the bottleneck value, and the processor that defines it is called the bottleneck processor. Note that the objective function depends on the time points at which the messages are delivered.

In order to clarify the notations and assumptions, consider a six-processor system as shown in Fig. 6.1(a). In the figure, the processors are synchronized at time t_0. The computational load of each processor is of length five-units and shown as a gray rectangle. The send operation from processor P_k to P_ℓ is labeled with $s_{k\ell}$ on the right-hand side of the time-line for processor P_k. The corresponding receive operation is shown on the left-hand side of the time-line for processor P_ℓ. For example, processor P_1 issues a send to P_3 at time t_0 and completes the send at time t_1 which also denotes the delivery time to P_3. Also note that P_3 receives a message from P_5 at the same time. In the figure, $r_1(1) = c_1(1) = t_5$, $r_2(1) = t_{10}$ and $c_1 = t_{15}$. The bottleneck processor is P_1 with the bottleneck value $t_b = t_{15}$.

Reconsider the same system where the messages are sent according to the order as shown in Fig. 6.1(b). In this setting, P_1 is also a bottleneck processor with value $t_b = t_{11}$.

Note that if a processor P_p never stays idle then it will reach completion at time $|S_1(p)| + W + |S_2(p)|$. The optimum bottleneck value cannot be less than the maximum of these values. Therefore, the order given in Fig. 6.1(b) is the best possible. Let P_q and P_r be the maximally loaded processors in the pre- and post-communication phases respectively, i.e., $|S_1(q)| \geq |S_1(p)|$ and $|S_2(r)| \geq |S_2(p)|$ for all p. Then, the bottleneck value cannot be larger than $|S_1(q)| + W + |S_2(r)|$. The setting in Fig. 6.1(a) attains this worst possible bottleneck value.

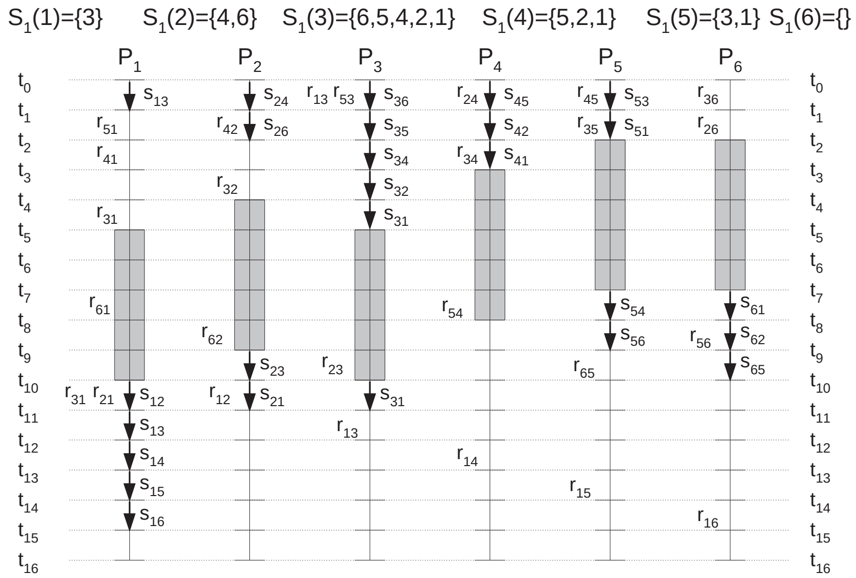
Observe that in a given message order, the bottleneck occurs at a processor with an outgoing message. Meaning that, for any bottleneck processor that receives a message at time t_b, there is a processor which finishes a send operation at time t_b. Therefore, for a processor P_p to be a bottleneck processor we require

$$c'_p = c_1(p) + W + |S_2(p)| \tag{6.6}$$

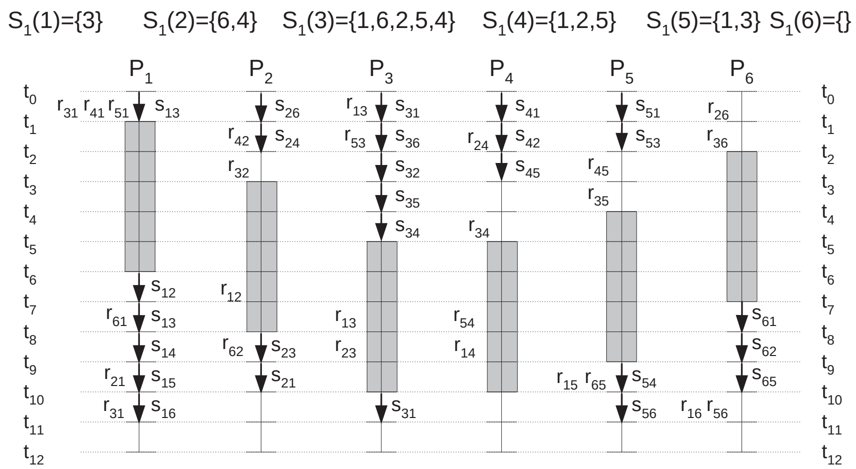
as a bottleneck value. Hence, our objective reduces to

$$minimize\{\max_p\{c'_p\}\}. \tag{6.7}$$

Also observe that the bottleneck processor and value remains as is, for any order of the post-communication messages. Therefore, our problem reduces to ordering the messages in the pre-communication phase. From these observations



(a) A sample message order which produces worst completion time.



(b) A sample message order which produces best completion time.

Figure 6.1: Worst and best order of the messages.

we reach the intuitive idea of assigning the maximally loaded processor in the post-communication phase to the first position in each send-list. This will make the processor with maximum $|S_2(\cdot)|$ to enter the computation phase as soon as possible. Extending this to the remaining processors we develop the following algorithm. First, each processor P_p determines its key-value $key(p) = |S_2(p)|$. Second, each processor obtains the key-values of all other processors with an all-to-all communication on the key-values. Third, each processor P_p sorts its send-list $S_1(p)$ in descending order of the key-values of the receiving processors. These sorted send-lists determine the message order in the pre-communication phase, where the order in the post-communication phase is arbitrary.

Theorem 6.1 *The above algorithm obtains the optimal solution that minimizes the maximum completion time.*

Proof. We take an optimal solution and then modify it to have each send-list sorted in descending order of key-values.

Consider an optimal solution. Let processor P_b be the bottleneck processor finishing its sends at time t_b. For each send-list in the pre-communication phase, we perform the following operations.

For any P_ℓ with $key_b \leq key_\ell$ where P_b and P_ℓ are in the same send-list $S_1(p)$, if $s_1(p,\ell) \leq s_1(p,b)$, then we are done, if not swap $s_1(p,\ell)$ and $s_1(p,b)$. Let $t_s = s_1(p,\ell)$ before the swap operation. Then, we have $t_s + W + key_\ell \leq t_b$ before the swap. After the swap we will have $t_s + W + key_b$ and $t_h + W + key_\ell$ for some $t_h < t_s$, for processors P_b and P_ℓ. These two values are less than t_b.

For any P_j with $key_j \leq key_b$ where P_j and P_b are in the same send-list $S_1(q)$, if $s_1(q,b) \leq s_1(q,j)$, then we are done, if not swap $s_1(q,b)$ and $s_1(q,j)$. Let $t_s = s_1(q,b)$ before the swap operation. Then, we have $t_s + W + key_b \leq t_b$. After the swap operation we will have $t_s + W + key_j$ and $t_h + W + key_b$ for some $t_h < t_s$ for processors P_j and P_b, respectively. Clearly, these two values are less than or equal to t_b.

For any P_u and P_v that are different from P_b with $key_u \leq key_v$ in a send-list

$S_1(r)$, if $s_1(r,v) \leq s_1(r,u)$, then we are done, if not swap $s_1(r,u)$ and $s_1(r,v)$. Let $t_s = s_1(r,v)$ before the swap operation. Then, we have $t_s + W + key_v \leq t_b$. After the swap operation we will have $t_s + W + key_u$ and $t_h + W + key_v$ for some $t_h < t_s$, for P_u and P_v respectively. These two values are less than or equal to t_b. Therefore, for each optimal solution we have an equivalent solution in which all send-lists in the pre-communication phase are sorted in decreasing order of the key values. Since the sorted order is unique with respect to the key values, the above algorithm is correct. □

6.3 Experiments

In order to see whether the findings in this chapter help in practice we have implemented a simple parallel program which is shown in Fig 6.2. In this figure, each processor first posts its non-blocking receives and then sends its messages in the order as they appear in the send-lists. In order to simplify the effects of the message volume on the message transfer time, we set the same volume for each message. We have used LAM [18] implementation of MPI and `mpirun` command without `-lamd` option. The parallel program were run on a Beowulf class [94] PC cluster with 24 nodes. Each node has a 400MHz Pentium-II processor and 128MB memory. The interconnection network is comprised of a 3COM SuperStack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node. The system runs Linux kernel 2.4.14 and Debian GNU/Linux 3.0 distribution.

We extracted the communication patterns of some row-column-parallel sparse matrix-vector multiply operations on 24 processors. Table 6.1 lists minimum and maximum number of send operations per processor under columns *min* and *max*. Total number of messages is given under the column *tot*.

For each test case, we have run the parallel program of Fig. 6.2 with small message lengths of 8, 64, 512, and 1024-bytes to justify the practicality of the assumptions made in this work. We have experimented with the best and worst

```
MPI_Barrier(MPI_COMM_WORLD);
startTime = MPI_Wtime();
for(iter = 0; iter < MAXITER; iter++){
  communication(preSendList, preSendCount, preRecvList,
                preRecvCount, sendBuf, recvBuf, iter);
  computation(sendBuf, recvBuf);
  communication(postSendList, postSendCount, postRecvList,
               postRecvCount, sendBuf, recvBuf, iter + 1);
  MPI_Barrier(MPI_COMM_WORLD);
}
totTime = 1000.0*MPI_Wtime() - 1000.0*startTime;
```

(a) A parallel program segment.

```
void computation(MSSGTYPE *sendBuf, MSSGTYPE *recvBuf){
 int i,j;
 for(i = 0; i < numProcs; i++){
   int indi = mssgSizes * i;
   for(j = 0; j < mssgSizes; j++)
      sendBuf[indi+j]=(sendBuf[indi+j] +
                       recvBuf[indi+j])/(MSSGTYPE)2;
 }
}
```

(b) Local computation performed at each processor.

```
void communication(int *sList, int sCnt, int *rList,
    int rCnt, MSSGTYPE *sBuf, MSSGTYPE *rBuf, int tag){
 int i;  MPI_Request reqs[rCnt]; MPI_Status stats[rCnt];
 for(i = 0 ; i < rCnt; i++){
   int p = rList[i], ind = p*mssgSizes;
   MPI_Irecv(&rBuf[ind], mssgSizes, bMPITYPESTR, p,
                       tag, MPI_COMM_WORLD,&reqs[i]);
 }
 for(i = 0; i < sCnt; i++){
   int p = sList[i], ind = myId * mssgSizes;
   MPI_Send(&sBuf[ind], mssgSizes,bMPITYPESTR, p,
                             tag, MPI_COMM_WORLD);
 }
 if(rCnt > 0) MPI_Waitall(rCnt, reqs, stats);
}
```

(c) Implementation of pre- and post-communication phases.

Figure 6.2: A simple parallel program

Table 6.1: Communication patterns and parallel running times on 24 processors.

Data	Communication pattern			Mssg order	Completion time				
					unit	milliseconds			
					Max	Message length (bytes)			
	min	max	tot		$\{c'_p\}$	8	64	512	1024
1-PRE	5	21	290	best	38	4.3	4.4	5.5	7.2
1-POST	6	22	358	worst	42	4.8	5.0	6.2	7.8
2-PRE	3	23	313	best	39	4.9	5.0	6.0	7.3
2-POST	11	22	370	worst	45	5.3	5.4	6.7	7.8
3-PRE	10	23	490	best	45	6.3	6.4	7.8	9.7
3-POST	15	23	504	worst	46	6.6	6.6	8.2	10.1
4-PRE	6	22	312	best	41	4.5	4.6	5.9	7.3
4-POST	10	20	356	worst	42	5.3	5.6	6.8	8.2
5-PRE	5	23	228	best	36	4.0	4.1	4.9	5.9
5-POST	7	13	228	worst	36	4.4	4.6	5.6	6.6
6-PRE	1	23	212	best	35	4.1	4.1	5.1	6.0
6-POST	4	17	236	worst	40	4.5	4.6	5.8	6.7
7-PRE	3	20	226	best	29	3.7	3.7	4.5	5.3
7-POST	7	17	253	worst	37	3.9	3.9	5.0	5.9
8-PRE	2	23	267	best	43	4.7	4.7	6.1	7.6
8-POST	4	22	278	worst	45	5.7	5.9	7.0	8.1
9-PRE	3	16	167	best	35	3.7	4.0	4.8	5.6
9-POST	4	20	273	worst	36	4.3	4.3	5.3	6.0
10-PRE	2	23	300	best	46	4.7	4.7	6.3	8.0
10-POST	10	23	316	worst	46	5.6	5.7	7.1	8.3
					W (Computation time):	0.00	0.01	0.06	0.11

orders. The best message orders are generated according to the algorithm proposed in § 6.2. The worst message orders are obtained by sorting the send-lists in increasing order of the key-values of the receiving processors. In all cases, we used the same message order in the post-communication phase. The running are presented in milliseconds in Table 6.1. We give the best among 20 runs (see [44] for choosing best in order to obtain reproducible results). In the table, we also give $\max_p\{c'_p\}$ for worst and best orders with $W = 0$. In all cases, the best order always gives better completion time than the worst order. In theory, however, we did not expect improvements for the 5th and 10th cases, in which the two orders give the same bottleneck value. This unexpected outcome may be resulting from the internals of the process that handles the communication requests. We are going to investigate this issue.

Chapter 7

SpMxVLib: A library for parallel matrix vector multiplies

We provide parallel matrix-vector multiply routines for 1D and 2D partitioned square and rectangular sparse matrices. We clearly give pseudocodes that perform necessary initializations for parallel execution. We show how to maximize the overlap between communication and computation through the proper usage of compressed sparse rows and column storage formats of the sparse matrices.

7.1 Introduction

Parallel sparse matrix vector multiplies (SpMxV) of the form $y \leftarrow Ax$ resides in the kernel of many scientific computations. One-dimensional (1D) [20, 21, 68, 53, 62, 99] and two-dimensional (2D) [23, 24, 105] partitioning methods are proposed to balance the computational loads of the processors while minimizing the communication overhead. In this chapter, we describe software that perform parallel SpMxV operations under 1D and 2D partitionings. Our aim is to ease the development of iterative methods. We give coding of the BiCGSTAB method as an example.

As noted in [95], software packages that implement only the parallel SpMxV operations are not common for several reasons. First, matrix-vector multiply is a simple operation; developers write their own routine. Second, there are different sparse matrix storage formats that fits different applications; it is difficult to design softwares that apply to all areas. Recently published sparse BLAS standard [36] even does not specify a data structure for storing sparse matrices. Rather, it allows complete freedom for sparse BLAS library developers to optimize their own libraries [37]. However, there are numerous software packages (see the sparse iterative solvers having parallel mode in Dongarra's survey [33]) that include utilities for performing distributed SpMxV operations; see for example PETSc [4], Aztec [60, 95], and PSPARSLIB [87].

Common features of existing software utilities for SpMxV operations are as follows. Most of the packages target 1D partitioned matrices, where y and x vectors have the same processor assignment as that of the rows or the columns of the matrix. This symmetric partitioning on the input and output vectors restricts the packages to square matrices. Some packages enable the user of the library to plug the necessary communication subroutines which are called between the partial executions of the SpMxV routines in a reverse communication [34] loop.

The characteristics of our software are as follows. Its SpMxV routines apply to 1D and 2D partitioned matrices of any shape. It can handle symmetric and unsymmetric partitionings on the input and output vectors. Our software uses point-to-point communication operations internally to exploit sparsity during communications, i.e., there does not exist any redundancy in the communication. To our knowledge, there does not exist any package that uses point-to-point communication when the matrices have 2D partitions. Also, we are not aware of any SpMxV libraries targeting rectangular matrices. Our package include entry-level matrix construction process as prescribed in Sparse BLAS standard [36]. There are software utilities to set-up communication data structures using the partitioning indicators. The software exploits compressed sparse column (CSC) and compressed sparse row (CSR) formats to achieve maximum communication and computation overlap.

In a parallel SpMxV implementation based on 1D partitioning, matrix, input vector, and output vector are partitioned among the processors using two partitioning indicators. One of the indicators describes both a partition on the matrix and a conformal partition on the input or output vector. The second indicator describes a partition on the remaining vector. In a parallel SpMxV implementation based on 2D partitioning, there are three partitioning indicators: on the output vector y, on the input vector x, and on the nonzeros of A. In some applications, the partitioning on the output vector is required to be the same as the partitioning on the input vector to avoid communication of vector entries during vector operations. In such cases, the two partitioning indicators on the input and output vectors coincide.

We have discussed the SpMxV operations under 1D and 2D partitioning of the sparse matrices in §2.1 and §2.2 respectively. It is worth noting that the pseudocodes given for multiplication routines imply a possibility of overlapping communication and computation (the third step in the row-parallel and column-parallel algorithms). We suggest reader review the Sections 2.1 and 2.2 on parallel SpMxV operations. Section 7.2 describes two sparse matrix storage formats and how to implement sequential SpMxV operations using these storage formats. In §7.3, we discuss necessary steps to realize efficient implementation of the SpMxV routines. We clearly give pseudocodes that set up communication and list issues that should be resolved to design parallel SpMxV routines along with our decisions. In §7.4, we give the interface of the library and its usability through actual implementations.

7.2 CSR and CSC storage formats

The most popular storage formats for the sparse matrices are the compressed sparse row (CSR) and compressed sparse column (CSC) formats [86]. In these formats, an $m \times n$ matrix A having z nonzeros is stored with three arrays. The first array is of size z and stores the nonzero entries of the matrix A row by row or column by column in the CSR and CSC formats, respectively. The second array is again of size z. In the CSR and CSC formats, this array stores, respectively, the column indices and the row indices of the nonzeros. The third array is of size $m+1$ and $n+1$ in the CSR and CSC formats, respectively. In the CSR format, the third array contains pointers to the beginning of each row in the first two arrays. In the CSC format, the third array contains pointers to the beginning of each column in the first two arrays.

Consider a 5×5 matrix

$$A = \begin{pmatrix} 1.1 & 0.0 & 0.0 & 1.4 & 0.0 \\ 2.1 & 2.2 & 0.0 & 2.4 & 0.0 \\ 3.1 & 0.0 & 3.3 & 3.4 & 3.5 \\ 0.0 & 0.0 & 4.3 & 4.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 5.5 \end{pmatrix}. \tag{7.1}$$

In the CSR format, the matrix A given above is stored as follows:

AA	1.1	1.4	2.1	2.2	2.4	3.1	3.3	3.4	3.5	4.3	4.4	5.5
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Since the matrix A given in Eq. 7.1 has 12 nonzeros, the array AA is of size 12 and stores the nonzeros row by row. The array JA is of size 12 and stores the column indices of the nonzeros again row by row. The array IA of size 6 holds pointers to the beginning of each row in arrays AA and JA. In particular, the values of the nonzeros in row i can be accessed by $\mathrm{AA}[j]$ for all j in $\mathrm{IA}[i] \leq j < \mathrm{IA}[i{+}1]$. The algorithm SpMxV-CSR given in Figure 7.1 shows the SpMxV operation using the CSR storage format.

SpMxV-CSR(AA, JA, IA, m, n, x, y)
for $i = 1$ to m **do**
  $tmp \leftarrow 0$
  $js \leftarrow$ IA[i]
  $je \leftarrow$ IA[i+1]
  for $j = js$ to $je - 1$ **do**
    $tmp \leftarrow tmp +$ AA$[j] \times x[JA[j]]$
  $y[i] \leftarrow tmp$

SpMxV-CSC(AA, IA, JA, m, n, x, y)
for $i = 1$ to m **do** $y[i] \leftarrow 0$
for $j = 1$ to n **do**
  $tx \leftarrow x[j]$
  $is \leftarrow$ JA[j]
  $ie \leftarrow$ JA[j+1]
  for $i = is$ to $ie - 1$ **do**
    $y[IA[i]] \leftarrow y[IA[i]]$ + AA$[i] \times tx$

Figure 7.1: SpMxV using the CSR and CSC storage formats.

In the CSC format, the matrix A given in Eq. 7.1 is stored as follows:

AA	1.1	2.1	3.1	2.2	3.3	4.3	1.4	2.4	3.4	4.4	3.5	5.5
IA	1	2	3	2	3	4	1	2	3	4	3	5
JA	1	4	5	7	11	13						

The array AA is again of size 12 and stores the nonzeros. The array IA of size 12 stores the row indices of the nonzeros. The array JA of size 6 holds pointers to the beginning of each column in arrays AA and IA. In particular, the values of the nonzeros in column j can be accessed by AA$[i]$ for all i in JA$[j] \leq i <$ JA$[j+1]$. The algorithm SpMxV-CSC given in Figure 7.1 shows the SpMxV operation using the CSC storage format.

7.3 Implementation details

In order to implement the above algorithms, one has to follow some initialization steps:

1. *Provide partitioning indicators on x and y vectors.*

In our implementation a central processor reads these partitioning indicators from different files and broadcast them to the other processors. We chose to provide each processor with partitioning indicators as a whole, i.e., each processor gets two arrays of size m and n one for the output vector and one for the input vector, respectively. Note that processors usually need only a small portion of these partition arrays. The rationale behind our choice is to enable the library handle arbitrary partitionings. That is, a processor can hold an x-vector entry and thus expand it even if it has not got a single nonzero in the corresponding column of A. Similarly, a processor can be set to be responsible for folding on a y-vector entry even if it does not generate partial result for that entry. We refer reader to our previous work [99] to get taste of such unusual partitionings. Note that these indicators are usually available; however it is possible to efficiently construct them as discussed by Pınar [81] and Tunimaro et.al. [95]. If the matrix partitioning is 1D, then one of the partitioning indicators is used to partition the matrix as well.

2. *Provide matrix nonzeros and x-vector entries to the processors.*

A central processor reads the matrix and vector entries and distribute them according to the partitions on the matrix and the vector. If the matrix partitioning is 2D, then the central processor reads partitioning indicator on the nonzeros of A from a file. In distributing the matrix, the central processor sends all the matrix entries of a processor in a single message.

3. *Determine the communication pattern.*

This is a complicated task that takes more time than SpMxV operation. We show the pseudocode, which is executed by each processor, for setting-up communication pattern for 2D case in Fig. 7.2. By removing lines pertaining to the y vector, the communication set-up procedure for 1D rowwise-partitioned matrices can be obtained. Similarly, the communication set-up procedure for 1D columnwise partitioned matrices can be obtained by removing the lines pertaining to the x vector. As seen in the figure, a certain processor sweeps (lines 1–9) its nonzeros to mark global indices of x-vector entries that it needs and global indices of y-vector entries on which it generates partial results. Note that after that sweep, processors know which x-vector entries are to be received from which processor, and which y-vector entries are to be sent to which processor. Here, a processor increments the counters corresponding to its rank to compute its local matrix's row and column dimensions. Two additional sweeps over row indices (lines 10–12) and column indices (lines 13–15) are necessary to handle arbitrary input and output vector partitionings. After the all-to-all communication (line 16), processors know the number of x-vector entries to be sent and the number of y-vector partial results to be received on per processor basis. After allocating necessary space, each processor become ready to exchange the indices of the vector entries to be communicated later in SpMxV routines. In lines 18–23, processors build the lists that hold global indices of the vector entries. In the remaining of the method, processors exchange those global index lists. After executing the depicted steps, each processor obtain the information on the vector entries' indices to be sent and to be received. Besides, each processor obtain the row and column dimensions for the sparse matrix in its memory.

4. *Determine local indices.*

For row-parallel algorithm, it is customary to renumber the x-vector entries that are accessed by processors in such a way that entries those belong to the same processor have contiguous indices; see [87, 95]. Analogously, for column-parallel algorithm, the y-vector entries that are to be sent to the same processor are renumbered contiguously. Combining these, it is preferable to renumber the x-vector entries to be received from the same processor contiguously and y-vector entries to be sent to the same processor contiguously in 2D case. In previous works [87, 95], developers renumbered the local vector entries starting from 0, and then continue on the external vector entries. We choose to renumber the vector entries according to the rank of the processors responsible on the corresponding vector entry. For example, processor P_k gives label to the external vector entries belonging to some other processor P_j where $j < k$, then gives labels to the local vector entries and then continues with labeling the external vector entries belonging to some other processor P_ℓ where $k < \ell$. Note that processor P_k can give labels to external vector entries belonging to a processor P_ℓ in any order; $x[i]$ can get label that is less than the label of $x[j]$ even processor P_ℓ labels $x[j]$

before $x[i]$. Since processors communicate global indices in the algorithm given in Fig. 7.2 this does not cause any problem.

5. *Set local indices for vector entries to be sent and to be received and also for matrix entries.*

This is a straightforward task that is done locally by each processor. Each processor sweeps the local data structures holding the global indices of local matrix, `xSendList`, `xRecvList`, `ySendList`, and `yRecvList`.

6. *Assemble the local sparse matrix.*

The local matrix is assembled using the labels determined in Step 4. In [87, 95], developers store the local matrix in CSR format for 1D rowwise-partitioned matrices. Considering their labeling procedure, this conceptually results in splitting the matrix into two, that is, $Aloc$ and $Acpl$. Here $Aloc$ contains nonzeros a_{ij} where $x[j]$ belongs to the associated processor, and $Acpl$ contains nonzeros a_{ic} where $x[c]$ belongs to some other processor. Remember that the mentioned works address symmetric partitioning on x and y vectors, hence $Aloc$ is a square matrix. In [87], developers mention that the matrices $Aloc$ and $Acpl$ can be stored in any format.

In our implementation, we explicitly split a processor's matrix into two sparse matrices $Aloc$ and $Acpl$ for row-parallel algorithm. Here $Aloc$ contains all nonzeros a_{ij} where $x[j]$ is local to the processor even if $y[i]$ belongs to some other processor and $Acpl$ contains all nonzeros a_{ie} where $x[e]$ belongs to some other processor. We store $Aloc$ and $Acpl$ in CSC format. Our aim is to maximize communication and computation overlap without incurring any extra operation. In [87], developers perform the first two steps of the row-parallel algorithm given in §2.1.1 by overlapping communication in the first step with the computation in the second one. After receiving all external x-vector entries, they continue with multiplication using $Acpl$ instead of the third step of the multiply algorithms given in §2.1.1 and §2.1.2. With our approach, we again obtain the same overlap in the first two steps and also achieve communication and computation overlap in the third step as well, i.e., we implement the third step of row-parallel algorithm as given in §2.1.1 and §2.1.2. When a processor receives a message in the third step containing some external x-vector entries, it can continue multiplying before waiting all external x-vector entries to arrive through exploiting the CSC format. Note that, using CSC format instead of CSR here is essential. In this format, we have explicit and immediate access to the row indices that has nonzeros in a given column. Hence, given an $x[j]$ one can update those $y[i]$'s where there is a nonzero a_{ij} sequentially without any search. Similarly, for column-parallel algorithm, we store *Aloc* and *Acpl* in CSR format to maximize the communication and computation overlap. In using CSR format here, our gain is the overlap between the messages a processor receives and the associated gathering of partial sums in step 3 of the column-parallel algorithm given in §2.1.2. In row-column parallel algorithm, we benefit both of the overlaps by using the same constructs in row and column-parallel algorithms.

7.4 Examples using the library

In Fig. 7.3, we give the listing of the interface to the library and a call to external BiCGSTAB solver we have developed using the SpMxV routines of the library. We used LAM implementation [18] of message passing interface (MPI). In Fig. 7.3, `buMatrix` data structure is used the store the sparse matrices, either in CSR or CSR format. The data structure also has fields to hold number of rows, columns, and nonzeros and to distinguish the storage formats. Each local matrix will be of this type (`loc and cpl` in the figure). The `parMatrix` structure is used to store distributed matrices for SpMxV operation. It has `loc` and `cpl` fields to store *Aloc* and *Acpl* as discussed in Section 7.3. The `parMatrix` structure also has fields to describe and implement communications. The communication handle `in` is used in communications regarding the input vectors of the SpMxV operation. The handle `out` is used in communications regarding the output vectors of SpMxV operations. We carry those communication handles along with matrices, however, they are used with vectors that appear in a SpMxV operation with the associated matrix. The field `scheme` designates the partitioning scheme on the matrix, which

```
SetupComm2D(A, xpartvec, ypartvec)
begin
    for each nonzero a_ij in A do #i and j are global indices
        if i is not marked then
            mark i
            increase ySendCount to processor p=ypartvec[i]
            put p into ySendList
        if j is not marked then
            mark j
            increase xRecvCount from processor p=xpartvec[j]
            put p into xRecvList
    for i=1..M do
        if i is not marked and myId=ypartvec[i] then
            mark i; increase ySendCount[myId]
    for j=1..N do
        if j is not marked and myId=xpartvec[j] then
            mark j; increase xRecvCount[myId]
    AlltoAll communication #send xRecvCounts, receive into xSendCounts;
                           #send ySendCounts, receive into yRecvCounts
    #allocate space for indices to be sent and to be received
    for each column j do
        if j is marked then
            put j into xIndexRecv list for processor p=xpartvec[j]
    for each row i do
        if i is marked then
            put i into yIndexSend list for processor p=ypartvec[i]
    for each processor in xRecvList do
        send xIndexRecv list to processor p
    for each processor in xSend list do
        receive into xIndexSend list for processor p
    for each processor in ySendList list do
        send yIndexSend list to processor p
    for each processor in yRecvList list do
        receive into yIndexRecv list for processor p
end
```

Figure 7.2: Setting up communication for 2D partition.

is used to decide on the SpMxV subroutine to call.

In Fig. 7.3, `initParLib` initializes the library. Note that this call creates a communication world under the current communication world (in the figure, the parent communication world is MPI's default `MPI_COMM_WORLD`). We hide the world that library's communication exist from the user, however, there are necessary subroutines which returns library's communication world handle. Such a distinct communication world is necessary in order to distinguish messages that are performed inside and outside the library (see Chapter 5 in [93]) to avoid message conflicts. The routine `readMatrixCoordinates` fills coordinate format storage area through communication. In `setup2D`, the initialization steps discussed in Section 7.3 are executed. It also assembles the matrices *Aloc* and *Acpl* from the coordinate format. The vector x is created with size

```
A->loc->n - A->in->recv->all[numProcs].
```

This is the size of the local x-vector entries which is mostly available explicitly without above computation. Note that matrices `loc` and `cpl` have column dimension `A->loc->n`. We choose to decouple the size of the local vectors from the local matrices' dimensions to free the user from parallel programming details. Similarly, vector `b` generated at the end of `mxv` routine holds only the entries of b that are folded in this processor. Finally, we delete the library's communication world by a call `quitParLib`. After this call, any attempts to call library's facilities will fail with a proper message.

We have developed BiCGSTAB [5, 104] to test the usability of the developed SpMxV library and give the code in Fig. 7.4. Once we have designed the SpMxV routine with proper interface, development of iterative methods becomes an easy task. One has to deal with vector operations only. We have provided a few linear vector operations such as `dotv`, `normv`, and `v_plus_cw` as well. These operations perform dot product of two vectors, compute the norm of a vector, and compute "scalar c w plus v" as in SAXPY of BLAS1, however, we choose to have different resulting vector. With these routines, the code looks like its pseudocode listing given in §5.3.

```
int main(int argc, char *argv[]) {
 int myId, numProcs, i,partScheme;
 int *rowIndices, *colIndices;  double *val;
 buMatrix *mtrx, *loc, *cpl;
 parMatrix *A;
 comm *in, comm *out;
 buVector *x, *b;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
 MPI_Comm_rank(MPI_COMM_WORLD, &myId);
 mtrx  = (buMatrix*) malloc(sizeof(buMatrix));
 loc = (buMatrix *) malloc(sizeof(buMatrix));
 cpl = (buMatrix *) malloc(sizeof(buMatrix));
 in = allocComm();
 out = allocComm();
 initParLib(MPI_COMM_WORLD);
 partScheme = PART_2D;
 readMatrixCoordinates(&rowIndices, &colIndices, &val,
     &(mtrx->nnz), &(mtrx->gm), &(mtrx->gn), &(mtrx->outPart),
     &(mtrx->inPart), argv[1], MPI_COMM_WORLD);
 setup2D(rowIndices, colIndices, val, mtrx->nnz,
         mtrx, loc, cpl, in, out, MPI_COMM_WORLD);
 A = (parMatrix *) malloc(sizeof(parMatrix));
 A->loc = loc;  A->cpl = cpl; A->in = in; A->out = out;
 A->scheme = partScheme;
  x = allocVector(A->loc->n - A->in->recv->all[numProcs]);
 for(i  = 0 ; i < x->sz; i++)
     x->val[i] = 1;
 b = (buVector *)malloc(sizeof(buVector));
 b->sz = 0;
 mxv(A, x, b, MPI_COMM_WORLD); /*compute b = A.1*/
 for( i  = 0 ; i < x->sz; i++) /*reset x to zero*/
     x->val[i] = 0.0;

 bicgstab( A, x, b, 200, 1.0e-12, MPI_COMM_WORLD);

 freeMatrix(mtrx); freeMatrix(loc); freeMatrix(cpl); free(A);
 freeVector(x); freeVector(b); freeComm(out);  freeComm(in);
 quitParLib(MPI_COMM_WORLD);
 MPI_Finalize();
}
```

Figure 7.3: A simple C program that uses library with 2D partitioning.

```
void bicgstab(parMatrix *A, buVector *x, buVector *b,
        int maxIter, double tol, MPI_Comm parentComm){
 buVector *rhat, *r, *p, *v, *w, *z;
 double c, old_rho, rho, alpha, old_omega, omega, beta;
 double res, res0;
 int k, myId, inSz, outSz;
 c = -1.0; old_rho = 1.0; alpha = 1.0; old_omega = 1.0;
 z = (buVector *)malloc(sizeof(buVector));
 z->sz = 0;
 mxv(A, x, z, parentComm);
 v_plus_cw(b, z, c, r);
 inSz = x->sz;  outSz = z->sz;
 p = allocVector(inSz); v = allocVector(inSz);
 r = allocVector(outSz); rhat = allocVector(outSz);
 w = allocVector(inSz + outSz); /*a local, temporary vector*/
 vcopy_vv(r, rhat);
 res0 = normv(b); k = 0;
 do {/*main BiCGSTAB loop*/
     k ++;
     rho = dotv(rhat, r);
     beta = (rho / old_rho) * (alpha / old_omega);
/*compute new p */
     v_plus_cw(p, v, -old_omega, z);   /*z=p -old_omega . v*/
     v_plus_cw(r, z, beta, p);         /*p = r - beta . z*/
/*compute new v, r, and alpha*/
     mxv(A, p, v, parentComm);
     alpha = rho/dotv(rhat, v);
     v_plus_cw(r, v, -alpha, r);
     if(normv(r)/res0 < tol){v_plus_cw(x, p, alpha, x); break;}
/*compute new omega*/
     mxv(A, r, z, parentCOMM);
     omega = dotv_div_dotv(z, r, z, z);  /* <z.r>/<z.z> */
/*compute new x and new r*/
     v_plus_cw(x, p, alpha, w);
     v_plus_cw(w, r, omega, x);
     v_plus_cw(r, z, -omega, r);
     res = normv(r);
     old_rho = rho;
     old_omega = omega;
     } while ( (res/res0 > tol) && (k < maxIter) );
 freeVectors(p, v, w, z, r, rhat);
}
```

Figure 7.4: A simple C program that uses library to develop BiCGSTAB.

7.5 Experiments

We have conducted experiments on a few sparse matrices. Properties of these matrices are listed in Table 7.1. In the table, m denotes the number of rows, n denotes the number of columns and z denotes the number of nonzeros of the matrices. The matrices are obtained from University of Florida Sparse Matrix Collection [32], Matrix Market [15], and [48].

The matrices are partitioned among 24 processors using PaToH software [22] to obtain rowwise, columnwise, fine-grain on nonzero basis, and checkerboard partitionings [21, 23, 24] to test our 1D and 2D parallel algorithms. We report the timings in Table 7.2 in milliseconds. Timings are obtained using `MPI_Wtime()` function. The columns having label R list the time consumed while reading the matrix, the vectors, and the partitioning indicators and providing each processor with the necessary data. The columns having labels S list the time consumed during setting up the communication and local matrix data structures. The columns having labels M list the time for an SpMxV operation.

Except for the matrix `bcsstk25`, the row-column-parallel algorithm based on checkerboard partitioning is the best among algorithm-partitioning combinations. The algorithm based on fine-grain partitioning is the worst except for matrix `pig-very`. In order to investigate these results, we give the communication pattern for parallel SpMxV computations. In Table 7.3, we give the communication pattern for row-parallel and column-parallel algorithms. In Table 7.4, we give the communication pattern for row-column-parallel algorithm based on fine-grain partitioning. In Table 7.5 we give the communication pattern for row-column-parallel algorithm based on checkerboard partitioning. For the checkerboard partitioning, we assumed a processor mesh of size 6×4. In these tables, communication patterns are specified by giving the total number of messages, the maximum number of messages per processor, the total volume of messages, and the maximum volume of messages per processor. These metrics refer to the *send* operations. Note that PaToH minimizes the total volume metric; in fine-grain partitioning case it minimizes the sum of the volumes in fold and expand steps;

Table 7.1: Properties of test matrices.

Matrix	m	n	z
memplus	17758	17758	126150
bcsstk25	15439	15439	252241
onetone2	36057	36057	254595
pig-very	174193	105882	463303
lhr34	35152	35152	799064

Table 7.2: Parallel times on 24 processors. R reading time (msecs.), S setup time (msecs.), M SpMxV time (msecs.).

	1D partitioning						2D partitioning					
	Row-Parallel			Column-Parallel			Fine Grain			Checkerboard		
Matrix	R	S	M	R	S	M	R	S	M	R	S	M
memplus	1220	25	3.25	1260	27	3.14	1890	42	4.95	1880	35	1.97
bcsstk25	1950	53	1.53	2000	47	1.37	3340	72	1.68	3360	38	1.66
onetone2	2550	70	2.24	2600	39	1.97	3880	58	3.77	3880	58	2.11
pig-very	6900	1060	5.63	6980	139	6.72	9530	187	6.03	8950	165	5.46
lhr34	6590	54	6.02	6310	56	4.56	10850	82	6.23	10620	89	5.96

in checkerboard partitioning case it minimizes the total volumes in fold and expand phases separately. Note that in all cases except the `pig-very` matrix, the total number of messages are doubled in fine-grain partitionings. Hence, even in the case of `memplus` in which total volume in fine-grain partitioning shrinks to 1/3 of other partitionings, the row-column-parallel algorithm based on fine-grain partitioning takes more time than the other SpMxV options. Note also that the checkerboard partitioning produces the smallest total number of messages in all cases. Combined with the advantage of bounding the maximum number of messages per processor, the checkerboard partitioning delivers the fastest SpMxV, where there are significant differences on the metrics pertaining to the number of messages.

Table 7.3: Communication pattern for parallel SpMxV based on 1D partitionings.

Matrix	Row-Parallel				Column-Parallel			
	Msg		Volume		Msg		Volume	
	Tot	Max	Tot	Max	Tot	Max	Tot	Max
memplus	522	23	12016	1070	509	23	10754	1689
bcsstk25	59	4	6855	377	62	5	6702	403
onetone2	132	7	5959	556	103	8	7890	835
pig-very	511	23	10196	1573	496	23	24172	3686
lhr34	233	15	24184	1482	239	13	24967	1323

Table 7.4: Communication pattern for parallel SpMxV based on 2D fine-gain partitionings.

Matrix	Expand				Fold			
	Msg		Volume		Msg		Volume	
	Tot	Max	Tot	Max	Tot	Max	Tot	Max
memplus	488	23	2407	347	472	23	2298	127
bcsstk25	47	4	1227	92	56	4	6094	362
onetone2	173	20	2783	349	132	10	3653	260
pig-very	525	23	12781	1553	61	5	169	19
lhr34	167	11	4185	449	234	13	22631	1533

Table 7.5: Communication pattern for parallel SpMxV based on 2D checkerboard partitionings.

Matrix	Expand				Fold			
	Msg		Volume		Msg		Volume	
	Tot	Max	Tot	Max	Tot	Max	Tot	Max
memplus	118	5	5998	365	72	3	6005	556
bcsstk25	10	1	1679	230	48	3	5717	450
onetone2	35	4	1332	266	65	3	7360	647
pig-very	116	5	6200	714	72	3	17497	957
lhr34	60	5	11261	792	72	3	20131	1324

Chapter 8

Conclusions

8.1 Summary

In Chapter 3, we proposed a two-phase approach that encapsulates multiple communication-cost metrics in one-dimensional partitioning of structurally unsymmetric square and rectangular sparse matrices. The objective of the first phase was to minimize the total message volume and maintain computational-load balance within the framework of the existing 1D matrix partitioning methods. For the second phase, communication-hypergraph models were proposed. Then, the problem of minimizing the total message latency while maintaining the balance on message-volume loads of processors was formulated as a hypergraph partitioning problem on communication hypergraphs. Several methods were proposed for partitioning communication hypergraphs. One of these methods was tailored to encapsulate the minimization of the maximum message count per processor. We tested the performance of the proposed models and the associated partitioning methods on a wide range of large unsymmetric square and rectangular sparse matrices. In these experiments, the proposed two-phase approach achieved substantial improvements in terms of the communication-cost performance metrics. We also implemented parallel matrix-vector and matrix-matrix-transpose-vector multiplies using MPI to see whether the theoretical improvements achieved in the given performance metrics hold in practice. Experiments on a PC cluster showed that the proposed approach can achieve substantial improvements in parallel run times.

In Chapter 4, we extended the two-phase approach of Chapter 3 to the 2D partitioning of matrices. We proposed communication-hypergraph models for the 2D partitioned matrices. Different from the 1D case, we developed models to obtain symmetric and unsymmetric partitioning on the input and output vectors. We tested the performance of the proposed models on practical implementations.

In Chapter 5, we demonstrated that hypergraph models are able to capture the application of multiple matrices. In particular, we developed models that allow simultaneous partitioning of a matrix and an approximate inverse preconditioner or the factors of an approximate inverse preconditioner. These points were raised by Hendrickson and Kolda [52]. We defined four operations to combine the previously proposed hypergraph models into a composite hypergraph. We showed how a partition on the composite hypergraph defines partitions on two or more matrices simultaneously. Further investigations on the proposed four operations shed light on the hypergraph models for 1D partitioning of sparse matrices. In particular, we described the creation of hypergraph models for 1D partitioning by starting from an intuitive hypergraph model and then applying the proposed operations. The computational structure of the preconditioned iterative methods abounds in scientific computing applications. We discussed the applicability of the proposed models in certain scientific computations. We showed the efficiency of the proposed composite hypergraph models through in-depth experimentation.

In Chapter 6, we addressed the problem of minimizing the completion time of a certain class of parallel program segments in which there is a small-to-medium grain computation between two irregular communication phases. We showed that the order in which the messages are sent affects the completion time and showed how to order the messages optimally. Experimental results on a PC cluster verified the existence of the specified problem and the validity of the proposed solution.

In Chapter 7, we presented a library developed for parallelizing sparse matrix vector multiply operations which includes algorithms for 1D and 2D partitioned matrices. The matrices can be square and rectangular. The library can handle vector distributions that are different than the matrix distribution. In our implementation of the SpMxV routines, processors perform scalar multiplications as soon as the associated data are available, e.g., the routines overlap communication and computation to the most possible extent.

8.2 Future work

Parallel matrix-vector multiply, $y \leftarrow Ax$, is one of the basic parallel reduction algorithms. Here, the x-vector entries are the input, and the y-vector entries are the output of the reduction operation. The matrix A corresponds to the mapping from the inputs to the outputs. Çatalyürek and Aykanat [24] briefly lists several practical problems that involve this correspondence. One concrete example is [26] which uses hypergraph models to decompose the computations. We think that the works presented in Chapters 3, 4, and 5 are applicable in distributed dataset applications. We will follow the literature on distributed dataset applications to identify new problems.

In Chapter 4, a sophisticated hypergraph partitioning tool that can handle fixed vertices in the context of multi-constraint partitioning was needed. Since the existing tools do not handle this type of partitioning, we are considering to develop such a method.

The experiments in Chapter 6 were conducted on hypothetical programs. In order to build a sound experimental framework for the methods proposed in there, we are trying to set up experiments to observe the findings of this chapter in parallel sparse matrix-vector multiplies. A generalization of the problem given in Chapter 6 addresses parallel programs that have multiple computation phases interleaved with communications. These kind of programs include multi-physics and multi-mesh simulations. We do not know the computational complexity of this general message ordering problem. We are going to investigate this problem in the near future.

The SpMxV library presented in Chapter 7 requires improvements to be set publicly available. The most important improvement needed is to couple the library with matrix partitioning tools such as PaToH [22] to simplify the parallel code development process. Another improvement needed is to implement a reference model for the vectors, matrices, and communicators using integers to enable inter-operability of the library with Fortran codes and to hide the complexity of data structures.

Another research direction, not as immediate as those given above, is to develop sparse matrix partitioning methods for heterogeneous computing systems. Heterogeneity comes into scene in two dimensions: heterogeneity in computing powers and heterogeneity in network access capabilities of the processors. We find handling the heterogeneity in computing powers to be easier than handling the heterogeneity in network access capabilities. Within this respect, we have done a work on task assignment in heterogeneous computing systems with homogeneous interconnection network [103]. The work in [103] addresses partitioning computational domains that are represented as undirected graphs, e.g., the dependencies between the tasks are binary and bidirectional. We are considering to extend our work on partitioning computational domains represented as graphs to partitioning computational domains represented as hypergraphs in order to address partitioning of the sparse matrices for heterogeneous computing systems.

Bibliography

[1] C. Ababei, N. Selvakkumaran, K. Bazargan, and G. Karypis. Multi-objective circuit partitioning for cutsize and path-based delay minimization. In *Proc. ICCAD 2002*, San Jose, CA, November 2002.

[2] C. J. Alpert, A. E. Caldwell, A. B. Kahng, and I. L. Markov. Hypergraph partitioning with fixed vertices. *IEEE Transactions on Computer-Aided Design*, 19(2):267–272, 2000.

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

[4] B. Balay, W. Gropp, L. C. McInnes, and B. Smith. PETSc 2.0 users manual. Technical report, Argonne National Laboratories, 1996.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. A. Van Der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[6] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.

[7] M. Benzi, J. K. Cullum, and M. Tuma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 22:1318–1332, 2000.

[8] M. Benzi, J. C. Haws, and M. Tuma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM Journal on Scientific Computing*, 22:1333–1353, 2000.

[9] M. Benzi, C. D. Meyer, and M. Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996.

[10] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, May 1998.

[11] M. Benzi and M. Tuma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 30:305–340, 1999.

[12] M. Benzi and M. Tuma. A parallel solver for large-scale Markov chains. *Applied Numerical Mathematics*, 41:135–153, 2002.

[13] M. Benzi and M. Tuma. Private communication, 2003.

[14] L. Bergamaschi, G. Pini, and F. Sartoretto. Approximate inverse preconditioning in the solution of sparse eigenproblems. *Numerical Linear Algebra and its Applications*, 7:99–116, 2000.

[15] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra. Matrix market: a web resource for test matrix collections. In R. Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, London, 1997.

[16] K. Brown, S. Attaway, S. J. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering*, 184:373–390, 2000.

[17] T. N. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, Philadelphia, 1993.

[18] G. Burns, R. Daoud, and J. Vaigl. LAM: an open cluster environment for MPI. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[19] U. V. Çatalyürek. *Hypergraph models for sparse matrix partitioning and reordering*. PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999.

[20] U. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplications. *Lecture Notes in Computer Science*, 1117:75–86, 1996.

[21] U. 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, 1999.

[22] U. V. Çatalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Computer Engineering Department, Bilkent University, 1999.

[23] U. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), 8th International Workshop on Solving Irregularly structured Problems in Parallel (Irregular 2001)*, April 2001.

[24] U. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of Scientific Computing 2001 (SC2001)*, pages 10–16, Denver, Colorado, November 2001.

[25] T. F. Chan, E. Chow, Y. Saad, and M. C. Yeung. Preserving symmetry in preconditioned krylov subspace methods. *SIAM Journal on Scientific Computing*, 20(2):568–581, 1998.

[26] C. Chang, T. Kurc, A. Sussman, U. Çatalyürek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of Tenth SIAM Conference on Parallel Processing for Scientific Computing (CDROM)*, Portsmouth, Virginia, USA, March 2001. SIAM.

[27] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[28] E. Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High Perf. Comput. Apps.*, 15:56–74, 2001.

[29] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998.

[30] J. K. Cullum, K. Johnson, and M. Tuma. Effects of problem decomposition on the convergence behavior of parallel numerical algorithms. *Numerical Linear Algebra with Applications*, 10:445–465, 2003.

[31] A. Daşdan and C. Aykanat. Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(2):169–178, 1997.

[32] T. Davis. University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[33] J. Dongarra. Freely available software for linear algebra on the web. URL http://www.netlib.org/utk/people/JackDongarra/la-sw.html, July 2001.

[34] J. Dongarra, V. Eijkhout, and A. Kalhan. Reverse communication interface for linear algebra templates for iterative methods. Technical Report UT-CS-95-291, University of Tennessee at Knoxville, 1995.

[35] J. J. Dongarra and T. H. Dunigan. Message-passing performance of various computers. *Concurrency—Practice and Experience*, 9(10):915–926, 1997.

[36] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Software*, 28(2):239–267, June 2002.

[37] I. S. Duff and C. Vömel. Algorithm 818: A reference model implementation of the sparse BLAS in Fortran 95. *ACM Trans. Math. Software*, 28(2):268–283, June 2002.

[38] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[39] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, March 1993.

[40] R. W. Freund and N. M. Nachtigal. QMR: A quasi-minimal residual algorithm for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[41] R. W. Freund and N. M. Nachtigal. A new krylov-subspace method for symmetric indefinite linear systems. Technical Report ORNL/TM-12754, Oak Ridge National Labs., May 1994.

[42] G. H. Golub and C. F. V. Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[43] N. I. M. Gould and J. A. Scott. Sparse approximate-inverse preconditioners using norm minimization techniques. *SIAM Journal on Scientific Computing*, 19(2):605–625, March 1998.

[44] W. Gropp and E. Lusk. Reproducible measurements of mpi performance characteristics. Technical Report ANL/MCS-P755-0699, Argonne National Laboratory, June 1999.

[45] M. J. Grote. SPAI: SParse Approximate Inverse Preconditioner. http://www.sam.math.ethz.ch/~grote/spai/.

[46] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[47] A. Gupta. Watson graph partitioning package. Technical Report RC 20453, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1996.

[48] M. Hegland. Description and use of animal breeding data for large least squares problems. Technical Report TR-PA-93-50, CERFACS, Toulouse, France, 1993.

[49] B. Hendrickson. Graph partitioning and parallel solvers: has the emperor no clothes? *Lecture Notes in Computer Science*, 1457:218–225, 1998.

[50] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Comp. Meth. Applied Mechanics and Engineering*, 184:485–500, 2000.

[51] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.

[52] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[53] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical report, Sandia National Laboratories, 1993.

[54] B. Hendrickson and R. Leland. The chaco users's guide, version 2.0. Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, NM, 1995.

[55] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. 1995 ACM/IEEE Conference, High Performance Networking and Computing, Supercomputing '95*, New York, 1995.

[56] B. Hendrickson, R. Leland, and S. Plimpton. An efficient parallel algorithm for matrix-vector multiplication. *Int. J. High Speed Computing*, 7(1):73–88, 1995.

[57] C. Hoover, A. DeGroot, J. Maltby, and R. Procassini. Paradyn: Dyna3d for massively parallel computers. Presentation at Tri-Laboratory Engineering Conference on Computational Modeling, 1995.

[58] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[59] T. Huckle. Approximate sparsity patterns for the inverse of a matrix and preconditioning. Technical Report 342/12/98 A, Technische Universität München, 1998.

[60] S. A. Hutchinson, J. N. Shadid, and R. S. Tunimaro. The Aztec user's guide - version 1.0. Technical report, Sandia National Laboratories, Albuquerque, NM, 1995.

[61] G. Karypis. Multi-constraint mesh partitioning for contact/impact computations. Technical Report 03-022, Department of Computer Science and Engineering/Army HPC Research Center, University of Minnesota, 2003.

[62] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[63] G. Karypis and V. Kumar. *MeTiS: A software package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0.* University of Minnesota, Department of Computer Science / Army HPC Research Center, Minneapolis, MN 55455, September 1998.

[64] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, May 1998.

[65] G. Karypis and 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 55455, November 1998.

[66] G. Karypis, V. Kumar, R. Aggarwal, and S. Shekhar. *hMeTiS a hypergraph partitioning package version 1.0.1.* University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[67] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb. 1970.

[68] T. G. Kolda. Partitioning sparse rectangular matrices for parallel processing. *Lecture Notes in Computer Science*, 1457:68–79, 1998.

[69] L. Y. Kolotilina and A. Y. Yeremin. Factorized sparse approximate inverse preconditionings. I: Theory. *SIAM J. Matrix Analysis and Applications*, 14:45–58, 1993.

[70] L. Y. Kolotilina and A. Y. Yeremin. Factorized sparse approximate inverse preconditioning II: Solution of 3D FE systems on massively parallel computers. *International Journal of High Speed Computing*, 7(2):191–216, 1995.

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

[72] J. G. Lewis, D. G. Payne, and R. A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High Performance Computing Conference*, 1994.

[73] K. McManus, M. Cross, C. Walshaw, S. Johnson, and P. Leggett. A scalable strategy for the parallelization of multiphysics unstructured mesh-iterative codes on distributed-memory systems. *The International Journal of High Performance Computing Applications*, 14(2):137–174, 2000.

[74] G. Meurant. *Computer Solution of Large Linear Systems.* Elsevier, Amsterdam, The Netherlands, 1999.

[75] J. G. Nagy, K. Palmer, and L. Perrone. Iterative methods for image deblurring: A Matlab object oriented approach. *Numerical Algorithms*, 36:73–93, 2004.

[76] A. T. Ogielski and W. Aielo. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Numerical Analysis*, 1993.

[77] H. Özaktaş. *Algorithms for linear and convex feasibility problems: A brief study of iterative projection, localization and subgradient methods.* PhD thesis, Bilkent University, 1998.

[78] H. Özaktas, M. Akgül, and M. Ç. Pınar. A parallel surrogate constraint approach to the linear feasibility problem. *Lecture Notes in Computer Science*, 1184:565–574, 1996.

[79] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford University, 1998.

[80] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:43–71, 1982.

[81] A. Pınar. *Combinatorial Algorithms in Scientific Computing*. PhD thesis, Department of Computer Science at University of Illinois at Urbana-Champaign, June 2001.

[82] A. Pınar, U. V. Çatalyürek, C. Aykanat, and M. Ç. Pınar. Decomposing linear programs for parallel solution. *Lecture Notes in Computer Science*, 1041:473–482, 1996.

[83] A. Pınar and B. Hendrickson. Graph partitioning for complex objectives. In *Proceedings of Irregular 2001*, April 2001.

[84] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. *J. Parallel Distrib. Comput.*, 50:104–122, 1998.

[85] S. Plimpton, B. Hendrickson, and J. Stewart. A parallel rendezvous algorithm for interpolation between multiple grids. In *Proc. Supercomputing'98*, Orlando, FL, November 1998.

[86] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., Boston, 1996.

[87] Y. Saad and A. V. Malevsky. P-SPARSLIB: A portable library of distributed memory sparse iterative solvers. Technical Report R95-71, Centre de Recherche en Calcul Applique, Jun 1995.

[88] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, Jan 1989.

[89] L. A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Transactions on Computers*, 42(12):1500–1504, December 1993.

[90] K. Schloegel, G. Karypis, and V. Kumar. A new algorithm for multi-objective graph partitioning. Technical Report 99-033, Department of Computer Science and Engineering, University of Minnesota, September 1999.

[91] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors, *CRPC Parallel Computing Handbook*, chapter 18, pages 491–541. Morgan Kaufmann, San Francisco, CA, 2002.

[92] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proc. ICCAD 2003*, San Jose, CA, November 2003.

[93] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1st edition, 1996.

[94] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranaweke, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[95] R. S. Tunimaro, J. N. Shadid, and S. A. Hutchinson. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, 10(3):229–247, March 1998.

[96] E. Turna. Parallel algorithms for the solution of large sparse inequality systems on distributed memory architectures. Master's thesis, Bilkent University, 1998.

[97] B. Uçar and C. Aykanat. Minimizing communication cost in fine-grain partitioning of sparse matrices. *Lecture Notes in Computer Science*, 2869:926–933, 2003.

[98] B. Uçar and C. Aykanat. ParMxvLib: A parallel library for sparse-matrix vector multiplies. In *Proc. 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI2003)*, pages 393–398, Orlando, Florida, USA, July 2003.

[99] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1827–1859, 2004.

[100] B. Uçar and C. Aykanat. Experiments on hypergraph models for parallelizing preconditioned iterative methods. Technical Report BU-CE-0413, Department of Computer Engineering, Bilkent University, 2004.

[101] B. Uçar and C. Aykanat. A message ordering problem in parallel programs. *Lecture Notes in Computer Science*, 3241:131–138, 2004.

[102] B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM Journal on Scientific Computing*, submitted, 2005.

[103] B. Uçar, C. Aykanat, K. Kaya, and M. Ikinci. Task assignment in heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, accepted, 2005.

[104] H. A. Van Der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[105] 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.

[106] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. *Applied Mathematical Modeling*, 25:123–140, 2000.

[107] K. Yang and K. G. Murty. New iterative methods for linear inequalities. *Journal of Optimization Theory and Applications*, 72:163–185, 1992.