

PARALLEL DIRECT AND HYBRID METHODS BASED ON ROW BLOCK PARTITIONING FOR SOLVING SPARSE LINEAR SYSTEMS

A DISSERTATION SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

By
Fahreddin Şükrü Torun
August 2017

Parallel direct and hybrid methods based on row block partitioning for
solving sparse linear systems

By Fahreddin Şükrü Torun

August 2017

We certify that we have read this dissertation and that in our opinion it is fully
adequate, in scope and in quality, as a dissertation for the degree of Doctor of
Philosophy.

Cevdet Aykanat(Advisor)

Murat Manguoğlu(Co-Advisor)

Engin Demir

Muhammet Mustafa Özdal

Tayfun Küçükyılmaz

Muhammed Abdullah Bülbül

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

In reference to ACM copyrighted material which is used with permission in this thesis, the ACM does not endorse any of Bilkent University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing ACM copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please request permissions from permissions@acm.org.

Copyright Information

F. Sukru Torun, Murat Manguoglu, and Cevdet Aykanat. 2017. "Parallel Minimum Norm Solution of Sparse Block Diagonal Column Overlapped Underdetermined Systems". ACM Transactions on Mathematical Software (TOMS) 43, 4, Article 31 (January 2017). DOI: <https://doi.org/10.1145/3004280>. ©2017 Association for Computing Machinery, Inc. Reprinted by permission.

ABSTRACT

PARALLEL DIRECT AND HYBRID METHODS BASED ON ROW BLOCK PARTITIONING FOR SOLVING SPARSE LINEAR SYSTEMS

Fahreddin Şükrü Torun

Ph.D. in Computer Engineering

Advisor: Cevdet Aykanat

Co-Advisor: Murat Manguoğlu

August 2017

Solving system of linear equations is a kernel operation in many scientific and industrial applications. These applications usually give rise to linear systems in which the coefficient matrix is very large and sparse. The need for solving these large and sparse systems within a reasonable time necessitates efficient and effective parallel solution methods.

In this thesis, three novel approaches are proposed for reducing the parallel solution time of linear systems. First, a new parallel algorithm, ParBaMiN, is proposed in order to find the minimum 2-norm solution of underdetermined linear systems, where the coefficient matrix is in the form of column overlapping block diagonal. The conducted experiments demonstrate the scalability of ParBaMiN on both shared and distributed memory architectures. Secondly, a new graph theoretical partitioning method is introduced in order to reduce the number of iterations in block Cimmino algorithm. Experimental results validate the effectiveness of the proposed partitioning method in terms of reducing the required number of iterations. Finally, we propose a new parallel hybrid method, BCDcols, which further reduces the number of iterations of block Cimmino algorithm for matrices with dense columns. BCDcols combines the block Cimmino iterative algorithm and a dense direct method for solving the system. Experimental results show that BCDcols significantly improves the convergence rate of block Cimmino method and hence reduces the parallel solution time.

Keywords: Sparse matrix, solution of sparse linear systems, parallel algorithms, direct methods, iterative methods, hybrid methods, underdetermined linear system, graph, graph partitioning, Schur complement.

ÖZET

SEYREK DOĞRUSAL SİSTEMLERİ ÇÖZMEK İÇİN SATIR BLOK BÖLÜMLEMEYE DAYALI PARALEL DİREKT VE HİBRİT METOTLAR

Fahreddin Şükrü Torun

Bilgisayar Mühendisliği, Doktora

Tez Danışmanı: Cevdet Aykanat

Eş Tez Danışmanı: Murat Manguoğlu

Ağustos 2017

Doğrusal sistemleri çözmek, bir çok bilimsel ve endüstriyel uygulamalarda çekirdek bir işlemdir. Bu uygulamalar genellikle büyük ve seyrek katsayı matrisi içeren doğrusal sistemler meydana getirmektedirler. Bu büyük ve seyrek matrisleri makul bir zamanda çözme ihtiyacı, etkin ve verimli paralel çözüm metodlarını gerekli kılmaktadır.

Bu tezde, doğrusal sistemlerin paralel çözüm zamanlarını hızlandıran üç yeni ve orjinal yaklaşım sunulmaktadır. İlk olarak, katsayı matrisi sütun kesişen blok köşegen biçiminde olan eksik-tanımlı (underdetermined) doğrusal sistemlerin en küçük 2-norm çözümünü bulan yeni bir paralel algoritma olan ParBaMiN sunulmaktadır. Paylaşılan bellek (shared memory) ve dağıtık bellek (distributed memory) mimarilerinde yapılan deneyler ParBaMiN'in ölçeklenebilirliğini göstermektedir. İkinci olarak, blok Cimmino algoritmasının iterasyon sayısını düşürmek için yeni bir çizge-kuramsal bölümlleme metodu önerilmektedir. Deney sonuçları, gerekli iterasyon sayısının azalımı bakımından önerilen metodun etkinliğini doğrulamaktadır. Son olarak, yoğun (dense) sütunları olan matrisler için blok Cimmino algoritmasının iterasyon sayısını daha da azaltan yeni bir paralel hibrit metot olan BCDcols'u sunmaktayız. BCDcols, sistemin çözümü için blok Cimmino iteratif algoritması ile bir yoğun direkt çözüm metodunu birleştirmektedir. Deney sonuçları, BCDcols'un blok Cimmino algoritmasının yakınsama hızını önemli ölçüde iyileştirdiğini göstermektedir ve böylelikle çözüme ulaşmak için gereken paralel zamana azaltmaktadır.

Anahtar sözcükler: Seyrek matris, seyrek doğrusal sistemlerin çözümü, paralel algoritmalar, direkt metotlar, yinelemeli metotlar, hibrit metotlar, eksik-tanımlı doğrusal sistem, çizge, çizge bölümlleme, Schur komplement.

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Cevdet Aykanat for his continuous support and guidance throughout my PhD. study. I am also very grateful to my co-advisor Assoc. Prof. Murat Manguoğlu for his invaluable helps and motivation throughout this study.

I would also like to thank my jury members, Asst. Prof. M. Mustafa Özdal, Asst. Prof. Engin Demir, Asst. Prof. Tayfun Küçükyılmaz and Asst. Prof. Muhammed Abdullah Bülbül for their insightful comments and suggestions.

I would like to thank my friends Kadir Akbudak, Oğuz Selvitopi, Seher Acer and Muhsin Can Orhan for their feedback and cooperation.

I am grateful to TÜBİTAK for financially supporting me under the national scholarship program BİDEB-2211.

I wish to express my deep sense of gratitude to my family for their unconditional support and patience.

Last but not the least, I thank and dedicate this dissertation to my beloved wife Tuğba, who helped me get through this tough period in the most positive way with her endless love and support, and my lovely daughter Büşra who has joined us recently.

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Organization of the Thesis	6
2	Background and Definitions	8
2.1	Types of System of Linear Equations	8
2.2	Sparse Matrix	10
2.3	Eigenvalues and Condition Number	11
2.4	Solution Methods	12
2.4.1	Direct Methods	12
2.4.2	Iterative Methods	15
2.4.3	Hybrid Methods	16
2.5	Graph and Graph Partitioning	17
3	ParBaMiN: Parallel Balance Scheme Algorithm for Finding Minimum Norm Solution of Underdetermined Systems	18
3.1	Introduction	18
3.2	The Proposed Algorithm: ParBaMiN	20
3.2.1	Formulation	20
3.2.2	Parallelization	24
3.3	Experimental Results	28
3.3.1	Datasets	28
3.3.2	Experimental Framework	33
3.3.3	Scalability Results	35
3.3.4	Accuracy of the Numerical Results	42

3.4	Forward and Backward Errors	44
3.5	Conclusion	47
4	A Novel Graph Theoretical Row-Block Partitioning Method for Block Cimmino Algorithm	48
4.1	Introduction	48
4.1.1	The Conjugate Gradient Acceleration of Block Cimmino Algorithm	51
4.1.2	The effect of partitioning	54
4.2	A Novel Row-Block Partitioning method for Block Cimmino Algorithm	56
4.2.1	Row Inner-product Graph Model	56
4.2.2	Row-block partitioning via partitioning the row inner-product graph	59
4.2.3	Implementation Details	64
4.3	Numerical Experiments	67
4.3.1	Experimental Framework	67
4.3.2	Effect of Row-block Partitioning on the Eigenvalue Spectrum	69
4.3.3	Dataset for Performance Analysis	71
4.3.4	Convergence and Parallel Performance	71
4.3.5	Preprocessing Overhead	74
4.3.6	Parallel Scalability	78
4.3.7	Parallel GP method for reducing the preprocessing overhead	81
4.4	Conclusion	83
5	BCDcols: A Parallel Solution Scheme Based on Block Cimmino Algorithm for Matrices with Dense Columns	84
5.1	Motivation and Problem Statement	85
5.2	BCDcols	87
5.2.1	Solution Process	89
5.3	Parallelization and Implementation Details	91
5.4	Numerical Experiments	95
5.4.1	Dataset	95
5.4.2	Experimental Framework	95

5.4.3	Results and Discussion	97
5.5	Conclusion	100
6	Conclusion	101
A	Code	114
A.1	ParBaMiN	114
A.2	BCDcols	117



List of Figures

1.1	Nonzero pattern of grow15 which has 15 diagonal blocks each of which consists of 20 rows and 63 columns, except the first one that has 43 columns.	4
1.2	4-way row-block partitioning of the coefficient matrix	5
1.3	Schur complement approach for block Cimmino algorithm	6
3.1	Two sample matrices with 8 sparse diagonal blocks of size 100×200	31
3.2	Nonzero pattern of grow15	33
3.3	Average speedup curves for realistic matrices (averages over 5 different overlap sizes)	37
3.4	Average speedup curves for synthetic matrices (averages over 5 different overlap sizes)	37
3.5	Speedup curves of real-world matrices	40
3.6	Variation of average speedup curves with varying overlap sizes . .	41
3.7	Relative residual norms of sequential algorithm and parallel Par-BaMiN for the datasets	43
3.8	Sequential and parallel $\ x\ _2$ values for the datasets	44
4.1	Comparison of classical Cimmino algorithm and Conjugate Gradient accelerated Cimmino algorithm	53
4.2	Row inner-product graph model	58
4.3	$C = AA^T$ matrix for the sample matrix in Figure 4.2a	59
4.4	(a) Uniform 3-way row partition of A and (b) 3-way vertex partition of $\mathcal{G}_{\text{RIP}}(A)$ induced by (a)	61
4.5	(a) “Good” 3-way vertex partition of $\mathcal{G}_{\text{RIP}}(A)$ and (b) 3-way row-block partition of A induced by (a)	62

4.6	3×3 checkboard partition of matrix $C = AA^T$ induced by 3-way (a) uniform (Figure 4.4a) and (b) $\mathcal{G}_{\text{RIP}}$ (Figure 4.5b) block-row partitions of matrix A	63
4.7	Eigenvalue spectrum of H for row-block partitionings on A	64
4.8	Nonzero patterns of A , $C = AA^T$, $\tilde{A}$ and $\tilde{C} = \tilde{A}\tilde{A}^T$	66
4.9	The eigenvalue spectrum of H for sherman3 , LeGresley_4908 . .	70
4.10	Speedup curves for CG time on the distributed memory archi- tecture compared to the best sequential running time (given in Table4.10)	79
5.1	Sparsity pattern of init_adder1_6 , dense columns are in blue color. 86	
5.2	Eigenvalue spectrum of H matrix (with the smallest and largest eigenvalues)	86
5.3	Matrix partitioning of $\mathcal{A}$ for BCDcols	89
5.4	Row-block partitioning of submatrices A and B	92
5.5	Row-block partitioning of the global system	92

List of Tables

3.1	Properties of realistic test matrix instances.	30
3.2	Properties of synthetic test matrices.	32
3.3	Properties of real-world matrix instances.	33
3.4	Settings for parallel SuiteSparseQR. T: Number of TBB threads, B: Number of BLAS threads.	34
3.5	Forward and backward errors using three different algorithms for solving problems with different condition numbers.	45
4.1	IP(Π^u)	63
4.2	IP(Π^g)	63
4.3	Properties of the test matrices which are sorted in descending order according to the number of nonzeros. (n : number of rows/columns, nnz : number of nonzeros)	72
4.4	Number of CG Iterations, best result for each matrix is shown in bold	73
4.5	Parallel CG time, best result for each matrix is shown in bold . .	75
4.6	Preprocessing times in seconds.	76
4.7	Total execution times (including preprocessing) in seconds. . . .	77
4.8	Properties of test matrices for parallel scalability and the sequential solutions times using the MUMPS direct solver in seconds. Mem. means the direct solver ran out of memory due to the fill-in. . . .	79
4.9	The number of CG iterations and parallel CG running times (in seconds).	80
4.10	Dissection of parallel running times (in seconds) and the number of CG iterations.	82

5.1	Properties of test matrices.	96
5.2	Number of iterations for block Cimmino algorithm and BCDcols. (NC: Not Converged)	98
5.3	Parallel solution time in seconds for block Cimmino algorithm and BCDcols. (NC: Not Converged)	99



Chapter 1

Introduction

Solving system of linear equations is a kernel operation in many scientific and industrial applications such as physics [1, 2], meteorology [3], economics [4], structural analysis [5], bio-informatics [6] and data-analysis [7]. Generally the solution time of linear systems dominates the total execution time of these applications.

The aforementioned applications usually give rise to linear systems in which the coefficient matrix is very large and sparse. A matrix is said to be sparse if most of its values are zero. In order to exploit these high number of zero elements and increase the efficiency, these large and sparse matrices should be treated intelligently.

There are mainly two kind of algorithms for solving system of linear equations: direct and iterative methods. Direct methods solve the linear system by certain number of operations and find the exact solution without effects of round-off error in numerical operations. On the other hand, iterative methods find an approximate solution which is improved by a sequence of iterations.

There are some strengths and weaknesses of direct and iterative methods for solution of linear systems. Direct methods are more robust and more reliable than iterative methods, however they may need much more memory and computational resources especially for large and sparse systems. Iterative methods

require more predictable and less amount of memory but may suffer from slow convergence rate unless applying a preconditioner to the coefficient matrix. Preconditioning may improve the numerical properties of the iteration matrix and this leads to requiring less number of iterations in the solver. However, finding an appropriate preconditioner is not always possible and preconditioners are usually problem dependent. Hybrid methods are proposed as an alternative approach which combines direct and iterative methods. Hybrid methods principally aim to take advantage of the robustness of the direct methods and lower memory usage of the iterative methods.

Since today's systems of linear equations are very large, sequential solution methods for solving these systems are unfeasible and hence they may take huge amounts time, even days or months. Parallel solution methods are primarily proposed in order to reduce execution time of the solution methods through using multiple computational units. Although parallel solution methods have big advantages, most of the time it is difficult to design and develop an efficient parallel algorithm which usually requires intelligent techniques. In this thesis, we propose novel and intelligent techniques for reducing parallel solution of linear systems.

We study the solution of the system of linear equations of the form

$$Ax = b, \tag{1.1}$$

where A is a sparse matrix of size $m \times n$ whereas x and b are column vectors of size m and n , respectively. In order to reduce the time for parallel solution of linear systems, we propose three novel contributions all of which exploit row-block decomposition of the coefficient matrix A . In row-block partitioning, matrix A is partitioned into k row blocks, where $k < m$. Column vector b is also partitioned conformably such that

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}. \tag{1.2}$$

Depending on the algorithm design, the column vector x can also be partitioned

conformably. In the proposed parallel algorithms, each processor is responsible for a set of A_i row blocks in compliance with the number of processors in the environment.

In the next section, we will explain motivations behind our contributions. We principally aim to reduce the parallel solving time of large sparse systems.

1.1 Motivation

In this section, we state three different problems and motivation behind the solution of these problems. All of our works focus on improving the parallel solving time of linear system by using intelligent algorithms and techniques.

There are various applications that give rise to underdetermined coefficient matrices to be in column overlapping block diagonal or banded forms such as Spline Interpolation [8], Seismic Tomography [9] and Staircase Linear Programming [10, 11]. Staircase-structured linear programming problems may arise from multi-stage or time-phased systems such as structural design optimization [12] and economic planning over time [13]. One example of staircase-structured linear programming problem is **grow15** [14] which models input-output analysis of the U.S. economy [15] with 15 time periods. Figure 1.1 shows the nonzero pattern of the coefficient matrix of **grow15**. The general state-of-the-art solution tools for underdetermined linear systems may miss out the properties of the coefficient matrix for efficient parallel solution. For similar systems to **grow15**, we need to use an alternative algorithm in order to exploit the sparsity pattern of the coefficient matrix. General solution tools usually permute the coefficient matrix in the preprocessing step in order to reduce the computation complexity. However, permuting the coefficient matrix generally destroys this special nonzero structure. Thus, their parallel efficiency is very poor for this type of matrices. For this reason, we propose a new parallel algorithm, ParBaMiN, for solving sparse block diagonal column overlapped underdetermined systems. ParBaMiN efficiently utilizes the structure of block diagonal with column overlap in the coefficient matrix.

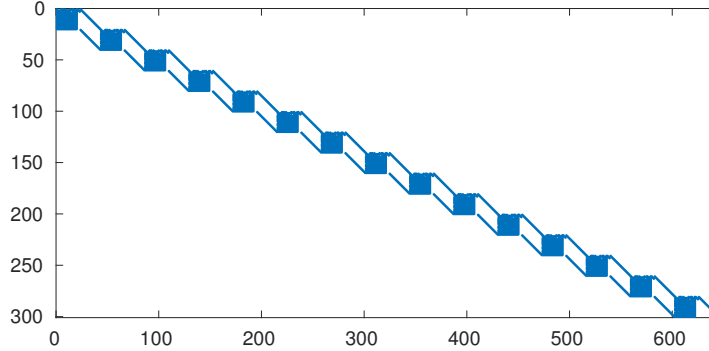


Figure 1.1: Nonzero pattern of **grow15** which has 15 diagonal blocks each of which consists of 20 rows and 63 columns, except the first one that has 43 columns.

It achieves more than 10 times faster parallel execution time than the parallel state-of-the-art algorithm. Moreover, ParBaMiN can work on distributed memory architectures, in which many computing nodes are interconnected with a network, whereas the baseline algorithm can run only on shared memory architectures in which there is one computing node that may contain multiple processing units.

Our other motivation is to reduce parallel solution time of a row-block projection algorithm. Row-block projection algorithms may suffer from slow convergence rate for solving systems. More specifically, we focus on block Cimmino algorithm as a row-block projection algorithm that obtains the solution by summing up the row-block projections in an iterative manner. Convergence rate of the block Cimmino method highly depends on the row-block partitioning of the coefficient matrix. We observe that the existing algorithms that aim at finding good row-block partitions do not improve the convergence of the method sufficiently. Furthermore, their objectives are not directly correspond to reduction in the required number of iterations. We propose a novel graph theoretical partitioning algorithm whose objective directly corresponds to minimizing the summation of the row-inner-product values between row blocks of the coefficient matrix. The proposed partitioning method significantly reduces the required number of iterations for the solution.

In block Cimmino algorithm, the coefficient matrix is partitioned into a number of row blocks. Figure 1.2a shows an example 4-way partition of the coefficient matrix. In each iteration of the block Cimmino, the underdetermined systems, in

which the coefficient matrices are the row blocks, are solved to find the minimum 2-norm solution. In order to solve these systems, one can use ParBaMiN, if the row blocks are in the column overlapping block diagonal form. That is, the row blocks A_i for $i = 1, \dots, 4$ in Figure 1.2b can be solved efficiently by using ParBaMiN.

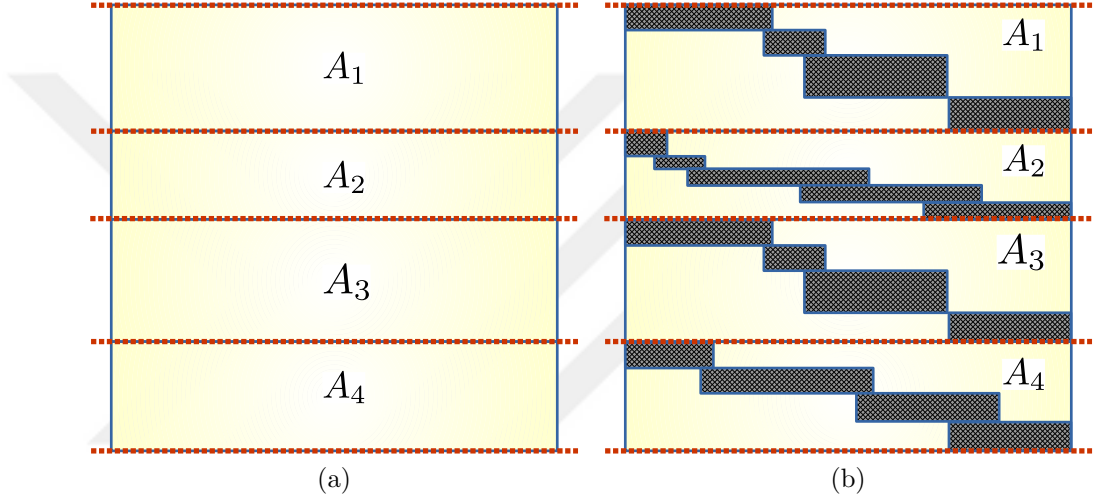


Figure 1.2: 4-way row-block partitioning of the coefficient matrix

The final motivation is to further reduce the parallel solution time of the linear systems in which the coefficient matrix has dense columns. We observe that the solution of the row-block projection methods for the linear systems whose coefficient matrices include dense columns suffers from very slow convergence rate even after applying a novel partitioning method. We propose a new algorithm, BCDcols, to accelerate the solution of these types of problems. The dense columns are treated separately so that they are exempted from the solution process of the row-block projection method. BCDcols adopts well-known Schur complement approach. In BCDcols, the original coefficient matrix is partitioned into 2×2 matrix blocks

$$\begin{bmatrix} A & B \\ C^T & D \end{bmatrix}, \quad (1.3)$$

where A is a square matrix whose some dense columns are moved to B submatrix. An example illustration is shown in Figure 1.3a. We use block Cimmino algorithm to solve the systems in which A is involved. The proposed scheme significantly

reduces the required number of iterations for the systems in which the coefficient matrix has dense columns.

Note that, since we use block Cimmino algorithm to solve the systems in which A is involved, we can also apply ParBaMiN to solve row blocks having the structure similar to A_i for $i = 1, \dots, 4$ in Figure 1.3b.

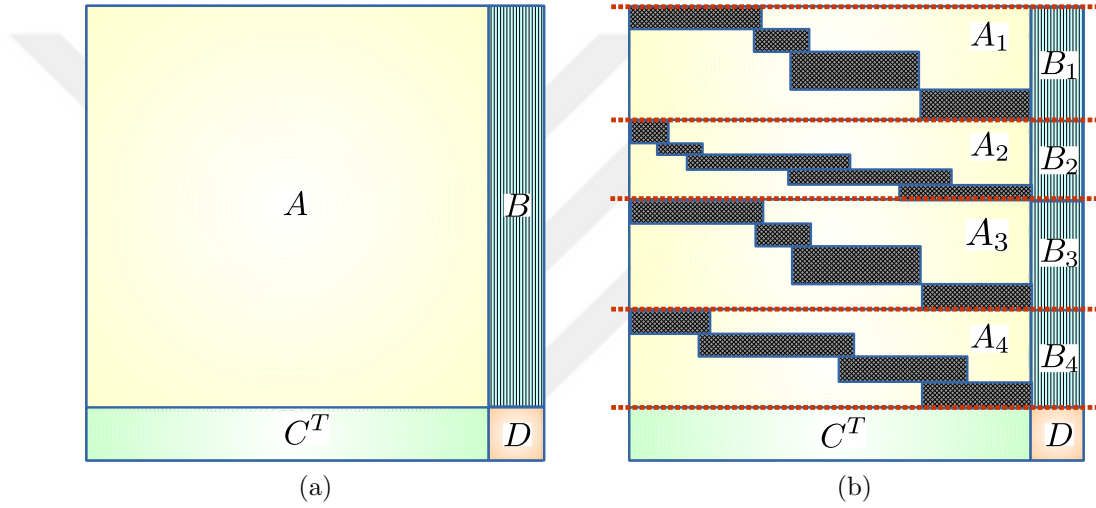


Figure 1.3: Schur complement approach for block Cimmino algorithm

1.2 Organization of the Thesis

Rest of this dissertation is organized as follows. In Chapter 2, we give brief background information which helps the reader to understand some concepts which are used throughout this dissertation.

In Chapter 3, we propose our first contribution, ParBaMiN, which is a parallel direct method that finds the minimum 2-norm solution of sparse column overlapped block diagonal underdetermined system of linear equations. We experimentally compare and confirm the error bound of ParBaMiN against the QR factorization-based techniques by using true single-precision arithmetic. We demonstrate numerical effectiveness as well as parallel scalability of ParBaMiN on both shared and distributed memory architectures for solving various types of

problems.

In Chapter 4, we introduce our second contribution which is a graph based row-block partitioning method for parallel block Cimmino algorithm. We present a graph model and partitioning problem on this model in order to make row blocks more orthogonal to each other. We demonstrate experimental results which confirm the validity of the proposed method through significant reduction in the required number of iterations for convergence.

In Chapter 5, we propose a new solution method, BCDcols, which is based on block Cimmino algorithm for solution of sparse matrices with dense columns. We demonstrate experimental results which show significant reduction in the required the number of iterations that leads to reduction in the parallel solution time of the system.

Finally, in Chapter 6, this dissertation is ended with conclusion.

Chapter 2

Background and Definitions

In this chapter, we give brief background information which helps the reader to understand some concepts which are used throughout this dissertation.

2.1 Types of System of Linear Equations

In the thesis, we consider the system of linear equations of the form

$$Ax = b, \tag{2.1}$$

where A is a $m \times n$ sparse matrix, whereas x and b are column vectors of size m and n , respectively. A matrix is said to be sparse if most of its values are zero. We assume linear systems are consistent, nonsingular and full rank [16]. For square linear systems which have the same number of equations and unknowns, if the coefficient matrix A is nonsingular then there is a unique solution which satisfies the system. For example, consider a square linear system

$$\begin{aligned} x_1 + 2x_2 &= 4 \\ 2x_1 + x_2 &= 5. \end{aligned} \tag{2.2}$$

The matrix representation of the system of linear equations in Equation(2.2) is shown as:

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}. \quad (2.3)$$

There is one unique solution which is $x_1 = 2$ and $x_2 = 1$.

If a system of linear equations has more equations than unknowns, it is called overdetermined ($m > n$) system. For example the following system is overdetermined and there is no solution for the system.

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 1 \end{bmatrix}. \quad (2.4)$$

If a system of linear equations has less equations than unknowns, it is called underdetermined ($m < n$) system. Underdetermined linear systems have infinitively many solutions. For example, the following sample system is underdetermined

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}. \quad (2.5)$$

One solution of this system is $x_1 = 2$, $x_2 = 1$ and $x_3 = 0$, where the 2-norm of this solution is 2.23. 2-norm of a vector is obtained by this formula

$$x = \sqrt{\sum_{i=1}^n x_i^2}. \quad (2.6)$$

However this solution is not the solution having the minimum 2-norm. The minimum 2-norm solution is $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$ and the 2-norm of this solution is 1.73. Section 2.4.1.1 explains the methods for finding the minimum 2-norm solution of the underdetermined systems.

2.2 Sparse Matrix

Since a sparse matrix contains more zero values than nonzero values, there are some data structures for efficient storage of sparse matrices. Storage complexity of sparse matrices can be reduced to $O(nnz)$, where nnz is the number of nonzero in the matrix. Coordinate or triplet format is the simplest data format to store sparse matrices. In triplet format, there is 3 unsorted lists whose sizes are nnz . Although triplet format is easy to construct, generally it is very difficult and inefficient to use in algorithms.

One intelligent way of storing sparse matrices is using compressed schemes. There are two well-known compressed schemes, which are compressed sparse row (CSR) and compressed sparse column (CSC). These schemes need three lists for storing the matrix; size of two of them is nnz and the other is m or n . In CSC scheme, two of those lists store row indices (**rowind**) and nonzero values (**val**) whereas the third list (**colptr**) stores the pointer for each column according to respective starting point of that column.

Sample sparse matrix and its CSC representation is

$$A = \begin{bmatrix} 4.1 & 0 & 3.9 & 0 & 6.6 \\ 0 & 2.0 & 0 & 0 & 0 \\ 9.5 & 0 & 5.3 & 0 & 0 \\ 0 & 2.6 & 0 & 1.6 & 0 \\ 1.9 & 0 & 3.1 & 0 & 5.7 \end{bmatrix}$$
$$\text{colptr} = \begin{bmatrix} 1 & 4 & 6 & 9 & 10 & 12 \end{bmatrix}$$
$$\text{rowind} = \begin{bmatrix} 1 & 3 & 5 & 2 & 4 & 1 & 3 & 5 & 4 & 1 & 5 \end{bmatrix}$$
$$\text{val} = \begin{bmatrix} 4.1 & 9.5 & 1.9 & 2.0 & 2.6 & 3.9 & 5.3 & 3.1 & 1.6 & 6.6 & 5.7 \end{bmatrix}.$$

As for CSR scheme, the sparse matrix is stored row by row and has the same storage complexity with CSC. CSR scheme for a matrix is equivalent to the CSC scheme for the transpose of the same matrix.

Data storage scheme should be determined accordingly to the algorithm design. In CSC scheme, accessing any columns is very fast, but accessing a specific row is very costly. In contrast, in CSR scheme, accessing a specific column is very expensive but it is very straightforward to access any row.

2.3 Eigenvalues and Condition Number

In order to explain eigenvalues, we first explain eigenvectors. In linear algebra, eigenvectors are vectors whose directions are not changed when multiplied with a matrix. That is, an eigenvector x is in the same direction with Ax . *Eigenvalue* is a scalar that satisfies the equation

$$Ax = \lambda x, \quad (2.7)$$

where λ is an eigenvalue and x is an eigenvector associated with λ .

Condition number of a matrix gives an upper bound of the amount of change with a small change in the input. Condition number can be any number in the interval $[1, \infty]$. One way to compute the condition number of matrix A is

$$\text{cond}(A) = \frac{\lambda_{\max}}{\lambda_{\min}}, \quad (2.8)$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the minimum and maximum eigenvalues of A .

A matrix A is called *singular* if its condition number is infinite. A singular matrix cannot be inverted. That is, if matrix A is singular, then the system $Ax = b$ can not be solved.

A matrix A is called *ill-conditioned* if the condition number of A is large. In the presence of rounding error, some solution methods which are less robust may not solve ill-conditioned systems.

2.4 Solution Methods

In this subsection, we briefly express some well-known solution methods for the square and underdetermined system of linear equations. Solution methods for overdetermined system of linear equations are out of scope of this dissertation.

2.4.1 Direct Methods

Direct methods are preferred mostly for their robustness. They can solve large problems and they are generally fast. However, these good points bring some expenses, such as very high and unpredictable storage requirements for large and sparse systems. Below some direct methods for square and underdetermined systems are explained briefly.

In direct methods [17, 18], the coefficient matrix is factorized into product of matrices, some examples are LU , Cholesky, WZ, QR factorizations and Singular value decomposition (SVD). In LU factorization, the square coefficient matrix is decomposed into the product of two matrices L and U . L is the lower triangular matrix and U is the upper triangular matrix. Then the system $Ax = b$ can be written as

$$LUx = b. \quad (2.9)$$

The solution consists of two steps. First the system $Ly = b$ is solved by forward substitution, then the system $Ux = y$ is solved by backward substitution. Since L and U matrices are triangular matrices, these two steps are straightforward and take less time than the factorization step. Cholesky factorization LL^T can be applicable when the coefficient matrix is square, symmetric and positive definite [16]. QR factorization and SVD can be applicable for the rectangular coefficient matrices. In QR factorization, matrix A is decomposed by the product of Q and R matrices, where Q is an orthogonal matrix [16] and R is an upper triangular matrix. Since Q is orthogonal, we have $Q^T Q = I$, where I is the identity matrix. Then the solution of the system $Ax = b$ is obtained by QR factorization

as

$$\begin{aligned} QRx &= b \\ Rx &= Q^T b \\ x &= R^{-1} Q^T b. \end{aligned} \tag{2.10}$$

x is the unique solution for square systems, but for underdetermined system x is one of the solution in infinitely many solutions. Finding the unique minimum 2-norm solution for underdetermined systems will be explained in the next subsection.

Another advantages of direct methods is that, after factorizing successfully the coefficient matrix into the factors (e.g. L and U in LU factorization), we may use these factors to solve different right-hand side vectors with consuming relatively small amount of arithmetic operations.

Efficient solution methods for sparse systems involve much more complicated algorithms. The most important issue to be handled is fill-in [19] of a matrix for sparse matrix factorization. When factorizing the matrix, some zero values may change to nonzero values. For some sparse matrices there could be a large number of changes from zero to nonzero. During factorization, fill-ins increase the density of the matrix and also the computational and memory cost of the algorithm. To prevent fill-ins, some rows and columns reordering algorithms [20, 21, 22, 23] are proposed.

There are several tools which support solution of sparse system by using direct methods. For sparse LU factorization, some examples are SuperLU [24], UMFPACK [25], PARDISO [26] and MUMPS [27]. For sparse QR factorization, some examples are SuiteSparseQR [28], HSL_MA49 [29], SPOOLES [30] and qr_mumps [31].

2.4.1.1 Direct Methods for Underdetermined Systems

As mentioned earlier, underdetermined system of linear equations has infinitely many solutions. Out of infinite solution, the minimum 2-norm solution is needed

by many engineering applications. In order to find minimum 2-norm solution for an underdetermined system of linear equations, we can use a direct method by using QR factorization. In this method, the QR factorization of the transpose of the coefficient matrix A

$$A^T = Q \begin{pmatrix} R \\ 0 \end{pmatrix} \quad (2.11)$$

is computed. Then, the minimum 2-norm solution of an underdetermined system can be computed as,

$$x = Q \begin{pmatrix} R^{-T}b \\ 0 \end{pmatrix}. \quad (2.12)$$

This method is referred as QR method.

Another method for computing minimum 2-norm solution is seminormal equations method (SNE) [32, 33]

$$x = A^T(R^T R)^{-1}b. \quad (2.13)$$

SNE method is cheaper than QR method since SNE method does not need Q matrix. However SNE method is less reliable when solving ill-conditioned problems.

Other factorizations such as LQ or SVD can also be used for obtaining the minimum 2-norm solution of an underdetermined linear least squares problems. Another approach is to use the normal equations to obtain the minimum norm solution,

$$x = A^T(AA^T)^{-1}b \quad (2.14)$$

which requires solution of a linear system. The solution of the linear system can be obtained directly via the Cholesky factorization or an iterative method. Although normal and seminormal equation approaches could save some storage and computational costs, they have the potential of introducing numerical difficulties that can be disastrous in some cases when the problem is ill-conditioned.

One alternative method is augmented system approach [34]. In the augmented system approach, minimum 2-norm solution is obtained by solving the linear

system

$$Ax = d, \quad (2.15)$$

where $d = b - Ax$. Then, the augmented system

$$\begin{pmatrix} I & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ d \end{pmatrix} \quad (2.16)$$

is solved. Hence, the minimum 2-norm solution x is obtained via the solution of the augmented system. This method is proven more stable than normal equations method [34].

2.4.2 Iterative Methods

Iterative methods are usually preferred for their low storage requirement and enabling solution of very large problems. Disadvantages of iterative methods are the number of iterations can be high and generally require a good preconditioner. There are 2 main category for iterative methods: stationary iterative methods and Krylov subspace methods. Some examples for stationary methods [5] are Jacobi, Gauss-Seidel, Successive Overrelaxation methods. Some examples for Krylov subspace methods [5] are Conjugate Gradient, Minimum Residual and BiConjugate Gradient methods.

In the thesis, we use Conjugate Gradient (CG) method which is the most popular iterative method [5] for solving a sparse linear system in which the coefficient matrix is square, symmetric and positive definite. A matrix is called positive definite if it provides $x^T Ax > 0$ for every x vectors, where $x \neq 0$. Algorithm 1 shows

the pseudo-code of CG algorithm.

ALGORITHM 1: Conjugate Gradient algorithm

```

1 Choose  $x_0$ 
2  $r_0 = b - Ax_0$ 
3  $p_0 = r_0$ 
4 while  $t=0,1,2,\dots$ , until convergence do
5    $\alpha_t = (r_t^T r_t) / ((Ap_t)^T p_t)$ 
6    $x_{t+1} = x_t + \alpha_t p_t$ 
7    $r_{t+1} = r_t - \alpha_t Ap_t$ 
8    $\beta_t = (r_{t+1}^T r_{t+1}) / (r_t^T r_t)$ 
9    $p_{t+1} = r_{t+1} + \beta_t p_t$ 
10 end

```

There are some libraries for solving sparse systems by using iterative methods. Some examples are PETSc [35], cuSPARSE [36] and SPARSKIT [37].

2.4.3 Hybrid Methods

Hybrid methods are emerged recently and they use combination of direct and iterative methods. The aim of hybrid methods is to take advantage of robustness of direct methods and less memory usage of iterative methods.

There are some types of hybrid methods. One type of hybrid method is factorization of nearby problem [38] which is easier to compute than the original matrix, and use this factorization as a preconditioner for the iterative solver.

Another type of hybrid method is that using a direct method on subproblems in an iterative method. That is, we first partition the matrix into row blocks then use a direct method for solution of the row blocks in an iterative solver. Some example of this type of hybrid methods are [39, 40]. In this thesis, we will focus on the block Cimmino iterative algorithm [41] which is a hybrid method where in each iteration underdetermined row blocks is solved by a direct method.

2.5 Graph and Graph Partitioning

In this section, we briefly explain the notion of graph and graph partitioning. The objective and the constraint of the graph partitioning problem, which is used in the thesis, is also expressed.

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph which contains a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. The edge connecting vertices v_i and v_j is denoted by $(v_i, v_j) \in \mathcal{E}$. A weight $w(v_i)$ on each vertex $v_i \in \mathcal{V}$ and a cost $cost(v_i, v_j)$ on each edge $(v_i, v_j) \in \mathcal{E}$ are assigned.

K-way vertex partition of $\mathcal{G}$ is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_K\}$, where vertex parts $\mathcal{V}_i$ are mutually disjoint. An edge (v_i, v_j) is called cut edge if the vertices v_i and v_j reside in different vertex parts. $\mathcal{E}_{\text{cut}}$ denotes the set of cut edges in the K-way vertex partition Π . W_k denotes the sum of weights of vertices in part $\mathcal{V}_k$. That is,

$$W_k = \sum_{v_i \in \mathcal{V}_k} w(v_i). \quad (2.17)$$

The constraint of the graph partitioning problem is to maintain balance on vertex part weight. That is,

$$W_k \leq W_{\text{avg}}(1 + \epsilon) \text{ for } k = 1, 2, \dots, K, \quad (2.18)$$

where ϵ is parameter that indicates the maximum imbalance ratio on part weights and

$$W_{\text{avg}} = \sum_{v_i \in \mathcal{V}} w(v_i) / K. \quad (2.19)$$

The objective of the graph partitioning is to minimize the sum of the edge costs in $\mathcal{E}_{\text{cut}}$, namely the cutsize, which is defined as

$$\text{cutsize}(\Pi) = \sum_{(v_i, v_j) \in \mathcal{E}_{\text{cut}}} cost(v_i, v_j). \quad (2.20)$$

Chapter 3

ParBaMiN: Parallel Balance Scheme Algorithm for Finding Minimum Norm Solution of Underdetermined Systems

3.1 Introduction

Underdetermined systems of equations [42, 43] in which the minimum norm solution needs to be computed arise in many application areas such as geophysics [44, 45], signal and image processing [46, 47] and biomedical engineering [48, 49]. An underdetermined system of equations, $Ax = b$ where A is an $m \times n$ matrix with $m < n$ and $\text{rank}(A) = m$ has infinitely many solutions. In this chapter, we will focus on the minimum 2-norm solution of underdetermined linear least squares problems. This solution can be obtained either by direct or iterative methods. We describe the well-known solution methods for obtaining minimum 2-norm solution of an underdetermined systems in Section 2.4.1.1.

see [50] for the original work

A considerable effort has been spent on developing efficient parallel and sequential implementations of general sparse QR algorithms such as SuiteSparseQR [51, 28], HSL MA49 [29], SPOOLES [30] and `qr_mumps` [31].

The Balance Scheme [52, 53, 54] is a parallel algorithm that was designed to solve ill-conditioned, banded, linear system of equations that are sparse within the band. The rows of the linear system are partitioned into k blocks where k is the number of processes. This partitioning gives rise to k linear least squares equations where each has infinitely many solutions. Since the coefficient matrix is banded, however, each block row has some columns that overlap with the neighboring blocks. The overlapping between the blocks means that parts of the solutions of the linear least squares problems can not be independent and the unique solution is obtained enforcing a constraint on the equality of the solution in the overlapping parts of the solution vector, giving rise to a smaller independent reduced system of equations which is solved either directly or iteratively.

In this chapter, we show that the Balance Scheme can be extended to obtain the minimum 2-norm solution of an underdetermined system of equations. The algorithm designed for underdetermined systems in which the coefficient matrix is in a generalized banded form with column overlapping block diagonals that are sparse within the block.

There are a number of applications that give rise to coefficient matrices that are in column overlapping block diagonal or banded forms such as Spline Interpolation [8], Seismic Tomography [9] and Linear Programming [10, 11]. We show that the proposed algorithm, ParBaMiN, finds the minimum 2-norm solution of the linear least squares problems. Furthermore, [55] has shown error bounds of the QR method and SNE for finding the minimum norm solution by simulating single precision arithmetic in MATLAB [56]. We confirm these bounds by using true single precision arithmetic in MATLAB and show that ParBaMiN is comparable to the QR method in MATLAB. Finally, since there are no sparse parallel QR factorization routines available for banded or a more generalized column overlapping block row matrices either on distributed or shared memory architectures, we compare the parallel scalability of ParBaMiN against a well known general

parallel sparse QR factorization algorithm on two different parallel computing platforms. Since we use the same sparse QR factorization algorithm for the block diagonals, ParBaMiN can also be considered to be an extension of the general sparse QR factorization to distributed memory computing platforms for obtaining the minimum 2-norm solution of underdetermined linear least squares problems.

Rest of this chapter is organized as follows. Section 3.2 presents the formulation of ParBaMiN and its parallelization. In Section 3.3, we compare ParBaMiN against a state-of-the-art algorithm in terms of parallel scalability and numerical accuracy on shared and distributed memory architectures. Finally, we confirm the error bounds for the underdetermined systems given in [55] using true single precision and show the benchmarks of the QR method, SNE and ParBaMiN in Section 3.4.

3.2 The Proposed Algorithm: ParBaMiN

In the first subsection, we first define the problem clearly and then explain the theory behind ParBaMiN. In the second subsection, we discuss the details of the parallel algorithm and the implementation based on the theoretical findings given in the first subsection.

3.2.1 Formulation

An underdetermined system of equations of the form

$$Ax = b \tag{3.1}$$

is given, where A is a sparse $m \times n$ ($m < n$) matrix. We assume A has full row rank. Our objective is to find the unique solution x such that $\|x\|_2$ is minimized.

We assume that the coefficient matrix A can be partitioned row-wise into k

diagonal blocks with column overlap as follows:

$$\begin{pmatrix} A_1 & B_1 & & & & \\ & C_2 & A_2 & B_2 & & \\ & & C_3 & A_3 & B_3 & \\ & & & \ddots & \ddots & \ddots \\ & & & & C_{k-1} & A_{k-1} & B_{k-1} \\ & & & & & C_k & A_k \end{pmatrix} \begin{pmatrix} x_1 \\ \xi_1 \\ x_2 \\ \xi_2 \\ \vdots \\ x_{k-1} \\ \xi_{k-1} \\ x_k \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{k-1} \\ b_k \end{pmatrix}. \quad (3.2)$$

In Equation (3.2), diagonal blocks are defined as

$$\begin{aligned} E_1 &= (A_1, B_1) \\ E_i &= (C_i, A_i, B_i), \text{ for } i = 2, \dots, k-1 \\ E_k &= (C_k, A_k). \end{aligned} \quad (3.3)$$

Here B_i and C_{i+1} denote the column overlapping sub-matrices of the successive i^{th} and $(i+1)^{th}$ diagonal blocks. Let t_i denote the size of the column overlap between the i^{th} and $(i+1)^{th}$ diagonal blocks for $i = 1, \dots, k-1$. In Equation (3.3), $A_i \in \mathbb{R}^{m_i \times n_i}$ for $i = 1, \dots, k$, $B_i \in \mathbb{R}^{m_i \times t_i}$ for $i = 1, \dots, k-1$ and $C_i \in \mathbb{R}^{m_i \times t_{i-1}}$ for $i = 2, \dots, k$. Since $A \in \mathbb{R}^{m \times n}$, we have

$$\begin{aligned} m &= \sum_{i=1}^k m_i, \\ n &= \sum_{i=1}^k n_i + t, \end{aligned} \quad (3.4)$$

where $t = \sum_{i=1}^{k-1} t_i$ denotes the total column overlap size. Note that $E_1 \in \mathbb{R}^{m_1 \times \tilde{n}_1}$, $E_i \in \mathbb{R}^{m_i \times \tilde{n}_i}$ for $i = 2, \dots, k-1$, and $E_k \in \mathbb{R}^{m_k \times \tilde{n}_k}$, where $\tilde{n}_1 = (n_1 + t_1)$, $\tilde{n}_i = (t_{i-1} + n_i + t_i)$ for $i = 2, \dots, k-1$, and $\tilde{n}_k = (t_{k-1} + n_k)$.

As seen in Equation (3.2), the right hand side vector b is partitioned conformably with the row block partition of the coefficient matrix. Hence, b_i is a subvector of size m_i for $i = 1, \dots, k$. As also seen in Equation (3.2), the solution vector x is partitioned conformably with the column block partition induced by

the block diagonal form of the coefficient matrix. Thus x_i is a subvector of size n_i for $i = 1, \dots, k$ and ξ_i is a subvector of size t_i for $i = 1, \dots, k - 1$.

Underdetermined linear least squares problem (3.2) gives rise to k smaller underdetermined linear least squares problems of the form

$$E_i z_i = b_i, \quad \text{for } i = 1, 2, \dots, k, \quad (3.5)$$

where

$$\begin{aligned} z_1 &= (x_1^T, \xi_1^T)^T \\ z_i &= (\hat{\xi}_{i-1}^T, x_i^T, \xi_i^T)^T, \quad \text{for } i = 1, 2, \dots, k - 1 \\ z_k &= (\hat{\xi}_{k-1}^T, x_k^T)^T. \end{aligned} \quad (3.6)$$

The general solution of the system is

$$z_i = p_i + Q_i y_i, \quad \text{for } i = 1, 2, \dots, k, \quad (3.7)$$

where p_i is a particular solution which can be computed independently, Q_i is a basis for $\mathcal{N}(E_i)$, and y_i is arbitrary. One can obtain Q_i via the QR factorization

$$E_i^T = (\hat{Q}_i, Q_i) \begin{pmatrix} R_i \\ 0 \end{pmatrix}, \quad (3.8)$$

where $\hat{Q}_i$ and Q_i have dimensions of $\tilde{n}_i \times m_i$ and $\tilde{n}_i \times (\tilde{n}_i - m_i)$ for $i = 1, \dots, k$, respectively.

One way of obtaining p_i is via the QR factorization computed above,

$$p_i = \hat{Q}_i (R_i^{-T} b_i). \quad (3.9)$$

Note that for x to be one of the solutions of the original system, $\hat{\xi}_i$ must be equal to ξ_i for $i = 1, \dots, k - 1$, which implies

$$\begin{aligned} p_1^{(low)} + (0, I)Q_1 y_1 &= p_2^{(up)} + (I, 0)Q_2 y_2 \\ p_2^{(low)} + (0, I)Q_2 y_2 &= p_3^{(up)} + (I, 0)Q_3 y_3 \\ &\dots = \dots \\ p_{(k-2)}^{(low)} + (0, I)Q_{k-2} y_{k-2} &= p_{(k-1)}^{(up)} + (I, 0)Q_{k-1} y_{k-1} \\ p_{(k-1)}^{(low)} + (0, I)Q_{k-1} y_{k-1} &= p_k^{(up)} + (I, 0)Q_k y_k, \end{aligned} \quad (3.10)$$

where $p_i^{(up)}$, $p_i^{(low)}$ and $p_i^{(rem)}$ refer to the upper, lower and the remaining parts of the vectors p_i , respectively, as shown below,

$$p_1 = \begin{pmatrix} p_1^{(rem)} \\ p_1^{(low)} \end{pmatrix}, p_i = \begin{pmatrix} p_i^{(up)} \\ p_i^{(rem)} \\ p_i^{(low)} \end{pmatrix}, p_k = \begin{pmatrix} p_k^{(up)} \\ p_k^{(rem)} \end{pmatrix}. \quad (3.11)$$

Note that vectors $p_i^{(low)}$ and $p_{i+1}^{(up)}$ both have size t_i . $(I, 0)$ and $(0, I)$ refer to matrices of size $t_i \times (\tilde{n}_i - m_i)$ where I is the identity matrix of size $t_i \times t_i$.

Equation (3.10) gives rise to a reduced system of equations,

$$My = \hat{p} \quad (3.12)$$

where

$$M = \begin{pmatrix} (0, I)Q_1 & (-I, 0)Q_2 & & & \\ & (0, I)Q_2 & (-I, 0)Q_3 & & \\ & & \vdots & \vdots & \\ & & & (0, I)Q_{k-2} & (-I, 0)Q_{k-1} \\ & & & & (0, I)Q_{k-1} & (-I, 0)Q_k \end{pmatrix} \quad (3.13)$$

and

$$\hat{p} = \begin{pmatrix} p_2^{(up)} - p_1^{(low)} \\ p_3^{(up)} - p_2^{(low)} \\ \dots \\ p_{k-1}^{(up)} - p_{k-2}^{(low)} \\ p_k^{(up)} - p_{k-1}^{(low)} \end{pmatrix}. \quad (3.14)$$

Note that M has a dimension of $t \times (n - m + t)$. After solving system (3.12) for y , we can retrieve the solution z_i via Equation (3.7).

We provide the following two lemmas and a theorem in order to show that ParBaMiN finds the minimum 2-norm solution.

Lemma 3.2.1 *Let $v = (v_1^T, v_2^T, \dots, v_k^T)^T$ be a vector of size N and partitioned into k subvectors. Then, $\|v\|_2$ is minimized iff each $\|v_i\|_2$ is minimized.*

Proof By definition of 2-norm, $\|v\|_2^2 = \sum_N |v_i|^2 = \|v_1\|_2^2 + \|v_2\|_2^2 + \dots + \|v_k\|_2^2$. Therefore, $\|v\|_2$ is minimized iff each $\|v_i\|_2$ is minimized. ■

Lemma 3.2.2 *Assume that the minimum norm solution for the undertermined systems of equations in Equation (3.10) is obtained. Then, $\|z_i\|_2$ in Equation (3.7) is also minimized.*

Proof By Lemma 3.2.1, if the minimum norm solution y to the reduced system is obtained then $\|y_i\|_2$ is also minimized. Given $z_i = p_i + Q_i y_i$ for $i = 1, 2, \dots, k$,

$$\begin{aligned} \|z_i\|_2^2 &= \|p_i + Q_i y_i\|_2^2 \\ &= (p_i + Q_i y_i)^T (p_i + Q_i y_i) \\ &= \|p_i\|_2^2 + 2y_i^T Q_i^T p_i + \|y_i\|_2^2 \quad (Q_i^T p_i = 0 \text{ since } Q_i \text{ is orthogonal to } p_i) \\ &= \|p_i\|_2^2 + \|y_i\|_2^2. \end{aligned}$$

Since $\|y_i\|_2^2$ is minimized and p_i is a fixed particular solution, $\|z_i\|_2$ is also minimized. ■

Theorem 3.2.3 *If the minimum norm solution is obtained for the reduced system in ParBaMiN, then the algorithm obtains the minimum norm solution x of the underdetermined systems of equations.*

Proof If the minimum norm solution is obtained for the reduced system, then $\|z_i\|_2$ is minimized for each i by Lemma 3.2.2. Therefore $\|x\|_2$ is also minimized by Lemma 3.2.1 since z_i subvectors constitute the solution vector x . ■

3.2.2 Parallelization

In this subsection, we provide the details of the parallel algorithm and its implementation based on the theoretical findings given in the previous subsection. The solution process of ParBaMiN can be summarized as follows:

- (a) Obtain a particular solution p_i according to Equation (3.9)
- (b) Solve the reduced system $My = \hat{p}$ according to Equation (3.12)
- (c) Retrieve the solution subvector z_i according to Equation (3.7).

Stage (a) involves the solution of k independent linear least squares problems which can be done in parallel without any communication. In Stage (b), we solve a smaller underdetermined linear system which can be done either sequentially or in parallel. We note that the solution of the reduced system can be computed either directly by forming it explicitly or iteratively without forming it explicitly, both requiring some communication. The retrieval of the solution, namely Stage (c), is done again in parallel without any communication. If needed, an optional last step involves gathering the global solution vector in one of the processors which requires some communication.

The pseudocode of ParBaMiN for processor i is given in Algorithm 2. The coefficient matrix and the right hand side vector of the given underdetermined linear system are distributed among processors by assigning each diagonal block E_i and the respective subvector b_i to processor i for $i = 1, 2, \dots, k$. A sample MATLAB code for ParBaMiN is included in Appendix A.1.

At line 1, each processor i concurrently and independently applies QR factorization on the coefficient matrix E_i^T of the local underdetermined least square problem $E_i z = b_i$. Sparse QR factorization of SuiteSparseQR [28] package is used in local sparse QR factorization operations. For performance and storage efficiency, the orthogonal matrices generated by the local QR factorizations are stored in Householder form at line 1. At line 2, each processor i concurrently computes particular solution p_i according to Equation (3.9). In the “if-then-else” statement between lines 3 and 12, the reduced system $My = \hat{p}$ is gathered in the master processor. For this purpose, all processors except the first and the last one send subvectors $p_i^{(low)}$ and $p_i^{(up)}$ and submatrices $(0, I)Q_i$ and $(-I, 0)Q_i$ to the master processor. The last processor only needs to send subvector $p_k^{(low)}$ and submatrix $(-I, 0)Q_k$ to the master processor. Each processor i obtains local $(0, I)(\hat{Q}_i, Q_i)$ and/or $(-I, 0)(\hat{Q}_i, Q_i)$ matrices by applying $(0, I)$ and $(-I, 0)$ to

ALGORITHM 2: Parallel algorithm for k -processor system (pseudocode for processor i).

Input: Block matrix E_i , RHS subvector b_i
Output: Solution vector z_i

```

1 Apply QR factorization  $E_i^T = (\hat{Q}_i, Q_i) \begin{pmatrix} R_i \\ 0 \end{pmatrix}$ 
2 Obtain local particular solution  $p_i = \hat{Q}_i(R_i^{-T}b_i)$ 
3 if  $1 < i < k$  then ▷ processor  $i$  neither master nor the last processor
4   | Send  $p_i^{(low)}$ ,  $p_i^{(up)}$ ,  $(0, I)Q_i$  and  $(-I, 0)Q_i$  to master processor (processor 1)
5 end
6 else if  $i = k$  then ▷ processor is the last processor
7   | Send  $p_k^{(low)}$  and  $(-I, 0)Q_k$  to master processor
8 end
9 else ▷ master processor
10  | Receive  $p_i$  subvectors and  $Q_i$  submatrices
11 end
12 endif
13 if  $i = 1$  then ▷ master Processor
14   | Construct coefficient matrix  $M$  and RHS vector  $\hat{p}$  for reduced system (see Eqs (3.13), (3.14))
15   | Find minimum 2-norm solution  $y$  of the reduced system  $My = \hat{p}$ 
16   | Scatter solution vector  $y$  among processors so that processor  $i$  receives subvector  $y_i$ 
17 end
18 endif
19 Compute the solution subvector  $z_i = p_i + Q_i y_i$  (see Equation 3.7)

```

$(\hat{Q}_i, Q_i)$ in the Householder form for $i = 2, \dots, k$ and $i = 1, \dots, k - 1$, respectively. Note that after forming the upper and lower parts of Q_i and $\hat{Q}_i$ we discard the parts of the $\hat{Q}_i$ since we do not need it in the algorithm.

In the “if” statement between lines 13–17, the master processor first constructs the coefficient matrix M and the right hand side vector $\hat{p}$ of the reduced system from the received submatrices and subvectors. It then finds the minimum 2-norm solution of the underdetermined linear system $My = \hat{p}$ via SuiteSparseQR_min2norm [57] procedure with default parameters. Finally, the master processor scatters the solution vector y among processors through collective communication operation *MPI_Scatterv* provided by the MPI library [58]. At line 19, each processor i concurrently computes the solution subvector z_i by using y_i according to Equation (3.7).

We should note here that the gather operation on the master processor is presented through point-to-point communications as shown in the “if-then-else” statement between lines 3–12, for the sake of clarity of the presentation. In our implementation, this gather operation is performed using the collective communication operation *MPI_Gatherv* provided by the MPI library.

Solving the reduced system is the only sequential part in our parallel algorithm due to the limitation of the current software implementation. We have experimented with multithreaded version of SuiteSparseQR for solving the reduced system $My = \hat{p}$, however it does not attain speedup on the solution time of the reduced system. We also have experimented with ScaLAPACK [59] subroutine PDGELS for the parallel solution of the reduced system, however we did not obtain speedup on more than 2 cores.

3.3 Experimental Results

3.3.1 Datasets

Matrix collections such as UF Sparse Matrix Collection [60] and Matrix Market [61] do not include many sparse matrices in a similar form of the problem defined in Section 3.2. For evaluating the performance of ParBaMiN, we generated two datasets which are referred here as realistic and synthetic datasets. Construction of realistic and synthetic datasets are described in Section 3.3.1.1 and 3.3.1.2, respectively. We also include a dataset obtained from a real-world application as described in Section 3.3.1.3. In all experiments, the right hand side vectors of the underdetermined systems are set to a vector whose elements are all ones.

3.3.1.1 Realistic Dataset

We have created our realistic dataset by linking together 64 copies of several real rectangular matrices from UF Sparse Matrix Collection in order to construct underdetermined linear systems of the form given in Equation (3.2). UF Sparse Matrix Collection has 37 rectangular matrices in least squares, computer graphics/vision and power network problems. To illustrate the performance of parallel algorithms, we have selected full rank matrices having more than 1,000 columns and rows. Due to the memory limitation, the matrices having more than 50,000 columns or rows and having more than 500,000 nonzeros are excluded. After applying these criteria, six matrices (shown in Table 3.1) remain for generating our realistic dataset.

In the construction of a coefficient matrix, if the shape of the underlying real matrix is tall and skinny then the transpose of the matrix is used in order to construct an underdetermined linear system. Each of the 64 diagonal blocks (i.e., E_i blocks) of a coefficient matrix is obtained by slightly perturbing nonzero values of the underlying real matrix in a random manner. That is, diagonal blocks differ slightly only in nonzero values, whereas they have the same sparsity

pattern. For each nonzero, the perturbation amount is selected randomly in the range $[-\alpha \times \eta, \alpha \times \eta]$, where $\eta = \|A\|_F$ and $\alpha = 0.10$ in order not to deviate too much from the real matrix. The original condition number of the diagonal block matrices are shown in Table 3.1.

As discussed in Section 3.2.2, sizes of the column overlaps between successive blocks define the size of the reduced system M , which is the bottleneck of our parallel algorithm. In order to observe the effects of overlap sizes on the performance of ParBaMiN, we generated 5 coefficient matrices with different amounts of column overlaps of $t_i = 5, 10, 20, 50$ and 100 for each underlying real matrix. Thus, the realistic dataset contains 30 underdetermined coefficient matrices. The properties of these coefficient matrices are shown in Table 3.1. Note that each of the 5 coefficient matrices obtained from the same underlying real matrix have the same number of rows and nonzeros whereas they have slightly different number of columns because of different column overlap sizes.

3.3.1.2 Synthetic Dataset

We have constructed a synthetic dataset of 15 underdetermined systems of the form given in Equation (3.2) from sparse uniformly distributed random diagonal blocks. In the constructed coefficient matrices, each diagonal block E_i has more columns than rows. We produce problems that are as large as possible and can fit into the memory when solving the minimum norm solution on our test platforms. In the synthetic dataset, the coefficient matrices have different dimensions, nonzero densities and overlap sizes. The properties of the synthetic underdetermined systems are shown in the Table 3.2.

Each synthetic test matrix contains 64 diagonal blocks and they are constructed by using average block diagonal size of $m_i \times \tilde{n}_i = 4000 \times 5000$. Both row and column dimensions of diagonal blocks are perturbed by a maximum amount of 5% on the above given averages. Each diagonal block E_i has a condition number of approximately 10^2 and a nonzero density of each diagonal block is set so that each row has about 10, 20 and 30 nonzeros by using *sprand* procedure in

Table 3.1: Properties of realistic test matrix instances.

Matrix	Diagonal Block	Coefficient Matrix				Seq. Soln. Time
ID	(<i>Cond.Num.</i>)	t_i	m	n	nnz	S.M. D.M.
1		5		1,887,237		14.29 10.36
2		10		1,886,922		14.35 10.35
3	graphics	20	756,608	1,886,292	7,549,056	14.40 10.41
4	($1.59e+8$)	50		1,884,402		14.43 10.46
5		100		1,881,252		14.47 10.49
6		5		1,820,613		30.83 20.52
7		10		1,820,298		31.44 20.79
8	kemelmacher	20	620,352	1,819,668	6,456,000	31.87 20.94
9	($2.38e+4$)	50		1,817,778		31.94 21.08
10		100		1,814,628		32.46 21.38
11		5		1,709,893		8.18 5.74
12		10		1,709,578		8.12 5.79
13	psse0	20	705,792	1,708,948	6,555,648	8.22 5.81
14	($1.07e+6$)	50		1,707,058		8.43 5.96
15		100		1,703,908		8.88 6.32
16		5		916,037		6.09 4.98
17		10		915,722		6.29 5.09
18	psse1	20	705,792	915,092	3,672,064	6.31 5.25
19	($1.12e+6$)	50		913,202		6.84 5.54
20		100		910,052		7.60 6.23
21		5		1,832,261		8.83 6.15
22		10		1,831,946		8.79 6.48
23	psse2	20	705,792	1,831,316	7,376,768	9.01 6.31
24	($1.03e+6$)	50		1,829,426		9.49 6.69
25		100		1,826,276		10.37 7.25
26		5		677,765		6.32 4.88
27		10		677,450		6.62 4.87
28	gemat1	20	315,456	676,820	2,981,824	6.66 4.92
29	($1.17e+8$)	50		674,930		6.88 5.16
30		100		671,780		7.31 5.47

t_i : overlap size, m : # of rows, n : # of columns, nnz : # of nonzeros.

Seq. Soln. Time: Sequential solution time (seconds) on a single core.

S.M.: Shared Memory, D.M.: Distributed Memory.

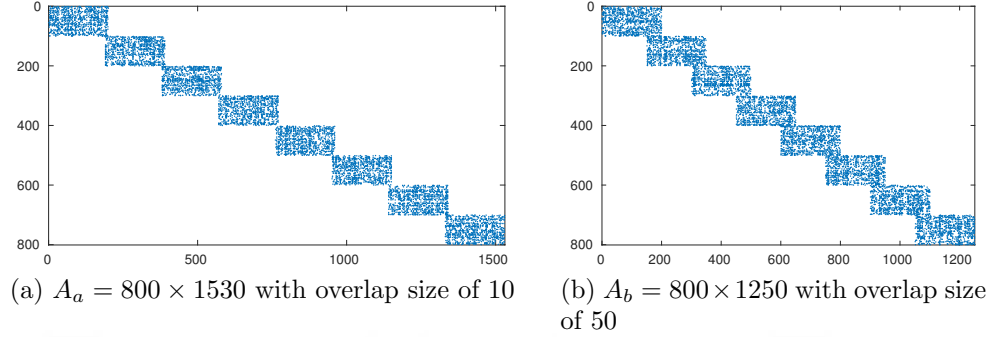


Figure 3.1: Two sample matrices with 8 sparse diagonal blocks of size 100×200

MATLAB version 2008. Test matrices with different amounts of column overlaps of $t_i = 5, 10, 20, 50$ and 100 are generated as in the realistic dataset.

Two sample matrices produced by our synthetic matrix generation tool are shown in Figure 3.1. These matrices have 8 rectangular sparse diagonal blocks of size 100×200 having 10 nonzeros per row. In Figure 3.1a, matrix A_a has a column overlap size of 10 and hence has a size of 800×1530 . In Figure 3.1b, matrix A_b has a column overlap size of 50 and hence has a size of 800×1250 .

3.3.1.3 Real-World Dataset

Staircase-structured linear programming problems may arise from multi-stage or time-phased systems such as structural design optimization [12] and economic planning over time [13]. There are methods that utilize Newton's approach for solving linear programming problems [62], quadratic programming problems [63] as well as linear feasibility problems [63, 64, 65]. At each Newton iteration, the minimum norm solution of an underdetermined linear system of equations needs to be obtained. Thus, solving such problems with staircase constraint matrices using Newton based approaches constitute a real world application of obtaining the minimum norm solution of an underdetermined linear least squares problem in which the coefficient matrix has a block diagonal column overlapping form.

One example of staircase-structured linear programming problem is

Table 3.2: Properties of synthetic test matrices.

Matrix	Diagonal Block			Coefficient Matrix				Seq. Soln. Time	
ID	m_i	$\tilde{n}_i$	nnz/m_i	t_i	m	n	nnz	S.M.	D.M.
1	4,014.45	5,013.11	10.02	5	256,925	320,466	2,573,387	256.68	158.30
2	4,012.75	5,011.20	9.99	10	256,816	320,151	2,565,271	263.36	161.79
3	3,984.64	5,002.05	10.00	20	255,017	318,871	2,549,146	261.81	164.05
4	4,003.02	5,006.02	9.97	50	256,193	317,235	2,554,582	289.03	179.56
5	3,979.41	5,003.92	10.00	100	254,682	313,951	2,546,786	338.93	202.36
6	4,000.36	5,011.11	20.01	5	256,023	320,396	5,122,404	501.07	320.49
7	4,007.44	5,000.70	19.97	10	256,476	319,415	5,120,651	510.48	328.27
8	3,995.78	4,995.19	19.94	20	255,730	318,432	5,100,235	536.83	343.52
9	4,007.44	5,000.56	19.97	50	256,476	316,886	5,120,651	607.22	392.03
10	4,003.39	5,008.63	20.00	100	256,217	314,252	5,123,315	783.57	499.13
11	4,010.45	4,998.69	29.91	5	256,669	319,601	7,676,243	652.17	428.48
12	4,000.47	4,993.03	29.87	10	256,030	318,924	7,648,085	667.33	436.80
13	3,997.77	4,995.28	29.89	20	255,857	318,438	7,646,384	723.38	470.64
14	4,020.23	5,002.38	29.93	50	257,295	317,002	7,700,351	872.08	567.53
15	3,998.67	5,011.69	30.00	100	255,915	314,448	7,674,427	1,038.83	798.59

m_i : average the number of rows, $\tilde{n}_i$: average the number of columns, nnz/m_i : average of the number nonzeros per row.

t_i : overlap size, m : # of rows, n : # of columns, nnz : # of nonzeros.

Seq. Soln. Time: sequential solution time (seconds) on a single core.

S.M.: Shared Memory, D.M.: Distributed Memory.

Table 3.3: Properties of real-world matrix instances.

Matrix ID	Coefficient Matrix					Seq. Soln. Time	
	Name	t_i	m	n	nnz	S.M.	D.M.
1	grow2560	20	51,200	110,080	962,540	1.45	1.20
2	grow5120	20	102,400	220,160	1,925,100	2.99	2.43
3	grow10240	20	204,800	440,320	3,85,0220	6.22	4.98
4	grow20480	20	409,600	880,640	7,700,460	13.29	10.30

t_i : overlap size, m : # of rows, n : # of columns, nnz : # of nonzeros.

Seq. Soln. Time: sequential solution time (seconds) on a single core.

S.M.: Shared Memory, D.M.: Distributed Memory.

grow15 [14] which models input-output analysis of the U.S. economy [15] with 15 time periods. Figure 3.2 shows the nonzero pattern of the coefficient matrix of **grow15**. This matrix has 15 diagonal blocks and each block consists of 20 rows and 63 columns (except the first block which has 43 columns) and successive diagonal blocks have 20 overlapping columns. In order to observe the parallel scalability of ParBaMiN, data of the same input-output analysis with 2560, 5120, 10240 and 20480 time periods were used. In the corresponding coefficient matrices, the number of periods is equal to the number of diagonal blocks and the amount of column overlap is $t_i = 20$. Properties of real-world dataset is shown in Table 3.3.

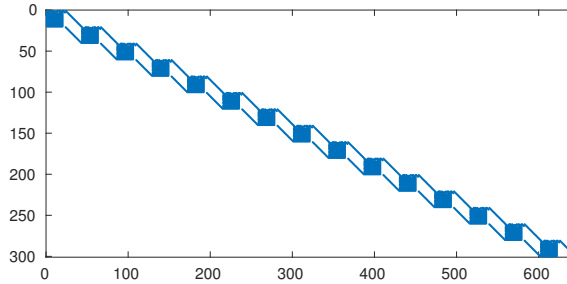


Figure 3.2: Nonzero pattern of **grow15**

3.3.2 Experimental Framework

The performance of ParBaMiN is tested on both shared and distributed memory architectures. As the shared memory architecture, we use a single node 64-core

computer that contains 4 AMD Opteron 6376 processors with each processor having 16 cores running at 2.3 GHz and a total of 128 GB of DDR3 memory. As the distributed memory architecture, we use SuperMUC (phase 1 thin nodes) [66] system. SuperMUC is an IBM System x iDataPlex dx360M4 system that consists of 9216 nodes, where each node contains 2 eight-core Intel(R) Xeon(R) E5-2680 processors running at 2.7 GHz and 32 GB RAM. Nodes are interconnected with a high-bandwidth low-latency switch network (Infiniband FDR10).

For comparing the performance of ParBaMiN, there are not any existing QR factorization routine specifically designed for matrices that have overlapping sparse diagonal blocks. Hence, as the baseline algorithm, we use SuiteSparseQR [28] which is a multithreaded multifrontal general sparse QR factorization procedure in SuiteSparse [51] software package. SuiteSparseQR uses Intel’s Threading Building Blocks (TBB) [67] library for providing parallelism on shared memory multicore architectures. METIS [68] reordering technique is used for both sequential and parallel SuiteSparseQR as recommended in [57]. We report sequential running times of the test matrices in Tables 3.1, 3.2 and 3.3 on two different computing platforms using sequential SuiteSparseQR.

Table 3.4: Settings for parallel SuiteSparseQR. T: Number of TBB threads, B: Number of BLAS threads.

System	Dataset	2 cores		4 cores		8 cores		16 cores		32 cores		64 cores	
		T	B	T	B	T	B	T	B	T	B	T	B
Shared Memory	Realistic	2	1	4	1	8	1	16	1	32	1	64	1
	Synthetic	2	1	4	1	8	1	8	2	16	2	64	1
	Real-World	2	1	4	1	8	1	16	1	32	1	64	1
Distributed Memory	Realistic	2	1	4	1	8	1	16	1	-	-	-	-
	Synthetic	2	1	2	2	8	2	16	4	-	-	-	-
	Real-World	2	1	4	1	8	1	16	1	-	-	-	-

In [28], the best performance results for SuiteSparseQR are achieved using a mixture of TBB threads and multithreaded BLAS. We have conducted experiments to see how the performance of SuiteSparseQR varies with using mixture of TBB threads and multithreaded BLAS on our datasets. In experiments for the real-world and realistic datasets, adding BLAS parallelism does not improve the performance so that the best performance is achieved via using only TBB threads

for all of the 34 matrix instances. For the synthetic dataset, using multiple BLAS threads improves the performance on some matrix instances. Table 3.4 displays the best settings according to average speedup values obtained for $1, 2, \dots, c$ TBB threads and $1, 2, \dots, c$ BLAS threads using c cores of the node of the distributed memory architecture and the shared memory architecture for $c = 2, 4, 8, 16$ and $c = 2, 4, 8, 16, 32, 64$, respectively. The parameters for SuiteSparseQR are selected as advised in [57], *SPQR_nthreads* is set to the number of cores, *SPQR_grain* is set to 2 times the number of cores and *SPQR_small* is set to 10^6 .

Intel compiler versions 14.0.1 and 15.0 with Intel Math Kernel Library (MKL) versions 11.1 and 11.2 are used on the shared and distributed memory architectures, respectively. ParBaMiN is implemented in C++ programming language. MPICH version 3.1.4 and IBM Platform MPI version 1.4 implementations of message passing interfaces are used on the shared and distributed memory architectures, respectively. In ParBaMiN, sequential SuiteSparseQR with single-threaded BLAS is used for local QR factorization in each block of ParBaMiN and each MPI process is mapped to one core on the shared and distributed memory architectures. Note that the performance of ParBaMiN may increase with adding the parallelism mechanisms of SuiteSparseQR and BLAS for the local QR factorizations. However, we have not observed any significant performance improvements for the datasets we used. Implementation of ParBaMiN is open source and can be found in [69]. In ParBaMiN, when we use smaller number of cores than the number of diagonal blocks, each core works on multiple successive blocks.

3.3.3 Scalability Results

In this subsection, we report the results of strong scalability tests performed for both ParBaMiN and the baseline algorithm. Figures 3.3, 3.4 and 3.5 display the scalability results for realistic, synthetic and real-world datasets, respectively. All of these three figures depict the results as speedup curves on both shared and distributed memory computing platforms. In Figures 3.3 and 3.4 speedup curve for each of the realistic and synthetic test matrices is obtained by averaging the

speedup values at each core count over 5 different column overlap sizes. That is, Figures 3.3 and 3.4 effectively display average speedup curves.

In Figures 3.3, 3.4 and 3.5, speedup values are displayed for 2, 4, 8, 16, 32 and 64 cores of the shared memory architecture for both ParBaMiN and baseline algorithm. For the distributed memory architecture, since the test matrices in realistic and synthetic datasets have 64 blocks, experiments for ParBaMiN are done with at most 64 cores across 4 distributed nodes. Thus Figures 3.3, 3.4 and 3.5 display the speedup values for 2, 4, 8, 16, 32 and 64 cores for ParBaMiN on the distributed memory architecture. However, all of these three figures do not display the performance of the baseline algorithm for 32 and 64 cores on the distributed memory architecture since SuiteSparseQR can only utilize thread level parallelism. Therefore, since the realistic dataset contains 30 test matrices, Figure 3.3 compares the speedup results of ParBaMiN against the baseline algorithm for $30 \times 6 = 180$ and $30 \times 4 = 120$ parallel running instances on the shared and distributed memory architectures, respectively. Similarly, since the synthetic dataset contains 15 test matrices, Figure 3.4 compares the speedup results of ParBaMiN against the baseline algorithm for $15 \times 6 = 90$ and $15 \times 4 = 60$ parallel running instances on the shared and distributed memory architectures, respectively. Since the real-world dataset contains 4 matrices, Figure 3.5 compares the speedup results of ParBaMiN against the baseline algorithm for $4 \times 6 = 24$ and $4 \times 4 = 16$ parallel running instances on the shared and distributed memory architectures, respectively.

In Figures 3.3, 3.4 and 3.5, solid lines (red color) and dashed lines (blue color) are respectively used to show the speedup curves for ParBaMiN and SuiteSparseQR. These figures also show the maximum average speedup values attained by both ParBaMiN and baseline algorithm on each core count for the sake of a better performance comparison.

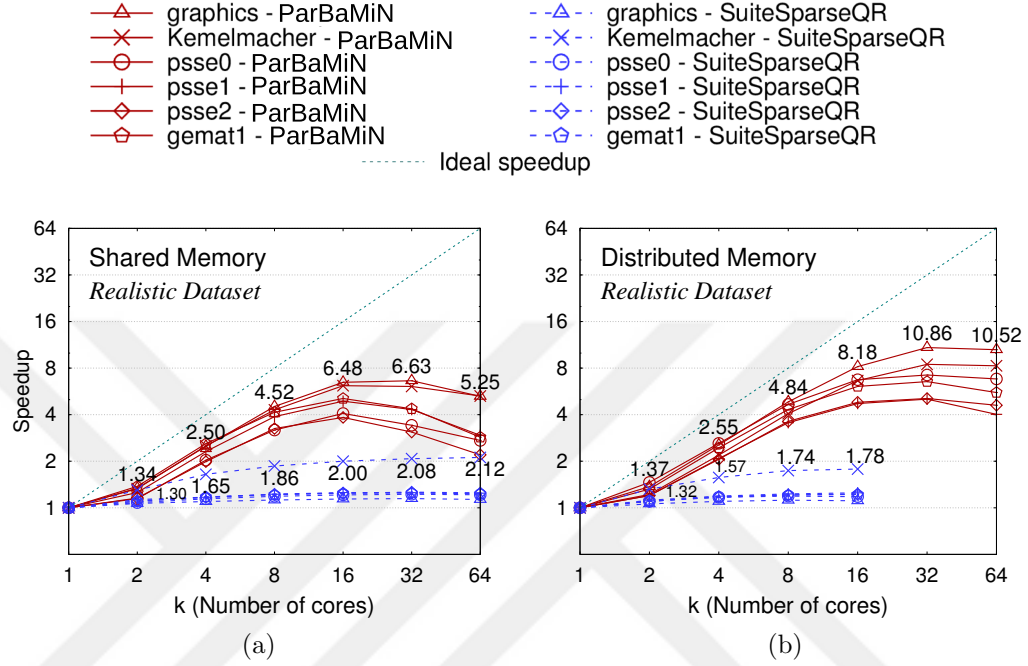


Figure 3.3: Average speedup curves for realistic matrices (averages over 5 different overlap sizes)

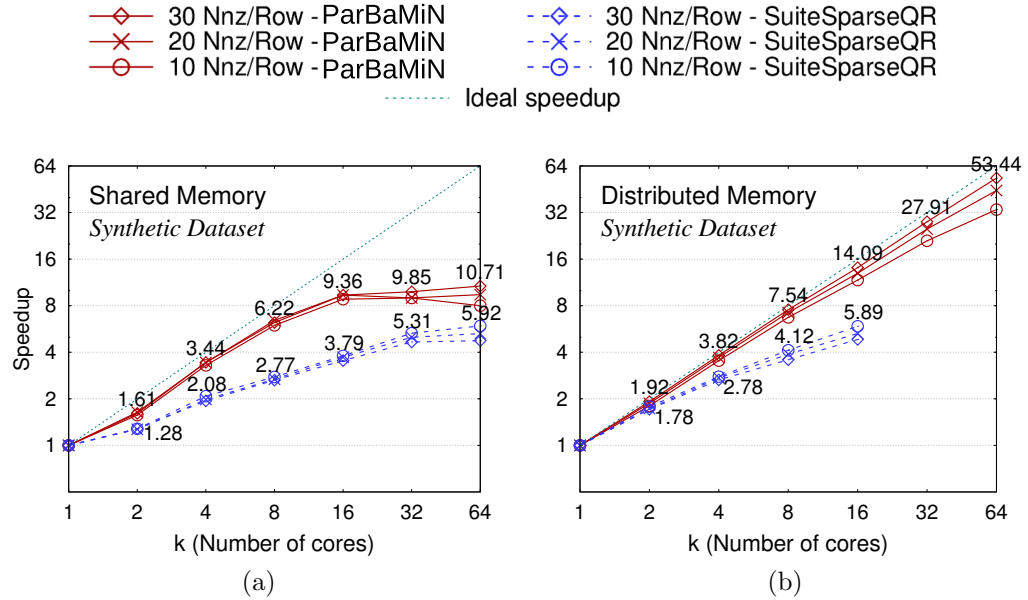


Figure 3.4: Average speedup curves for synthetic matrices (averages over 5 different overlap sizes)

3.3.3.1 Realistic Dataset

As seen in Figure 3.3, ParBaMiN shows much better scalability than the baseline algorithm for the realistic dataset. ParBaMiN achieves considerably better speedup values than the baseline algorithm on 275 out of 300 parallel running instances. The remaining 25 running instances, for which ParBaMiN fails to achieve better performance, are on 2 cores (15 out of 25 instances) and are for the matrices with the largest column overlap size of 100 on 32 and 64 cores (10 out of 25 instances).

As seen in Figure 3.3, the performance gap between ParBaMiN and baseline algorithm increases considerably with increasing core counts until 16 cores for the shared memory architecture and until 64 cores for the distributed memory architecture in favor of ParBaMiN, whereas the performance gap slightly decreases on 32 and 64 cores for the shared memory architecture. This is because the memory bandwidth begins to become a bottleneck for ParBaMiN on high number of cores in the shared memory architecture, since it uses relatively more memory than the baseline algorithm. As seen in the figure, ParBaMiN achieves best average speedups of 6.63 and 10.86 for the **graphics**-based matrices on the shared and distributed memory architectures, respectively. The baseline algorithm achieves best average speedups of 2.12 and 1.78 for the **Kemelmacher**-based matrices on the shared and distributed memory architectures, respectively.

3.3.3.2 Synthetic Dataset

As seen in Figure 3.4, ParBaMiN also scales much better than the baseline algorithm for the synthetic dataset. ParBaMiN achieves considerably better speedup values than the baseline algorithm on almost all of the 150 parallel running instances and fails to achieve better performance only for the 2 running instances which are on 2 cores of the distributed memory architecture.

As seen in Figure 3.4, for the synthetic dataset, the performance of ParBaMiN slightly decreases for sparser coefficient matrices. This is because, for a given

number of processors, with decreasing matrix density, the parallelization overhead due to communication and sequential solution of the reduced system remains almost the same, whereas the amount of concurrent local computations decrease.

As seen in Figure 3.4, ParBaMiN scales up until 32 and 64 cores on the shared and distributed memory architectures, respectively. ParBaMiN achieves maximum average speedup of 10.71 and 53.44 for matrices with 30 nonzeros per row on 64 cores of the shared and distributed memory architectures, respectively. On the other hand, the baseline algorithm achieves maximum average speedup of only 5.92 and 5.89 for the matrices with 10 nonzeros per row on 64 and 16 cores of the shared and distributed memory architectures, respectively.

3.3.3.3 Comparison of Realistic and Synthetic Datasets

Comparison of Figures 3.3 and 3.4 shows that both ParBaMiN and baseline algorithm achieve much better speedup for the synthetic dataset. This experimental finding is due to the fact that the problem sizes are much larger for synthetic dataset compared to those for the realistic dataset. Note that the problem size refers to the amount of work involved in the sequential algorithm which is also indicated in the sequential running times in the last two columns of Tables 3.1 and 3.2.

3.3.3.4 Effects of Overlap Size

Figure 3.6 is given in order to show the effect of overlap size on the performance of ParBaMiN and baseline algorithm for both realistic and synthetic datasets. In the figure, the speedup value for each overlap size is depicted as the average of the speedups obtained for 6 realistic or 3 synthetic test matrices (varied nonzero densities of 10, 20 and 30 nnz/row) on the given core count.

As seen in Figure 3.6, for both realistic and synthetic datasets, the performance of ParBaMiN decreases considerably with increasing overlap size, whereas

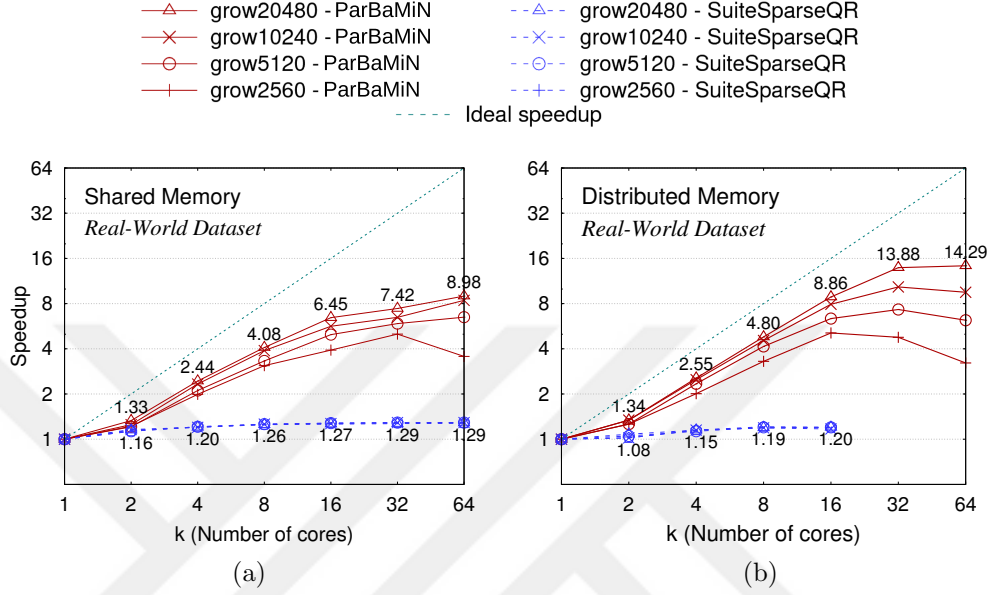


Figure 3.5: Speedup curves of real-world matrices

the performance of the baseline algorithm increases slightly with increasing overlap size. The former experimental finding is expected since increasing overlap size increases the size of the reduced system and thus increases the sequential portion of ParBaMiN. The latter experimental finding is because of the fact that the performance of the baseline algorithm is not sensitive to the overlap size and the column size of the coefficient matrix decreases with increasing overlap size in our experimental setting. Despite these expected experimental findings, ParBaMiN achieves better average speedup than the baseline algorithm for all parallel running instances except for the realistic matrices with largest overlap size of 100 on 64 cores of the shared memory architecture.

In accordance with the discussion given above, ParBaMiN achieves maximum speedup values for the matrices with smallest column overlap size of 5, whereas the baseline algorithm achieves maximum speed values for the matrices with the largest overlap size of 100. As seen in Figure 3.6, on the shared memory architecture, ParBaMiN achieves the maximum average speedup of 8.40 and 11.08 on 32 cores for realistic and synthetic datasets, respectively, whereas the baseline algorithm obtains maximum speedup of 1.39 and 6.02 on 64 cores for realistic

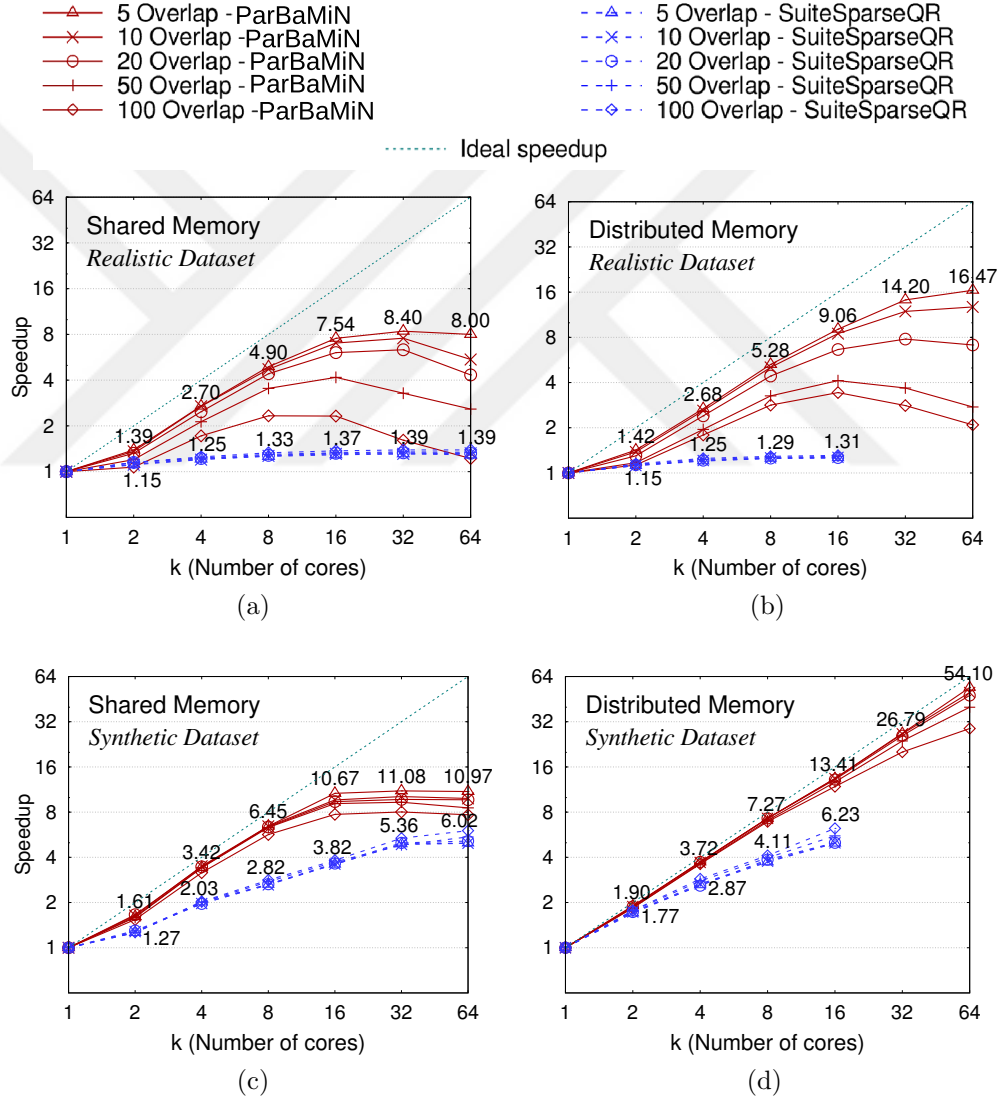


Figure 3.6: Variation of average speedup curves with varying overlap sizes

and synthetic datasets, respectively. On the distributed memory architecture, ParBaMiN obtains maximum speedup of 16.47 and 54.10 on 64 cores for the realistic and synthetic datasets, respectively, whereas the baseline algorithm achieves speedups of only 1.31 and 6.23 on 16 cores for realistic and synthetic dataset, respectively.

3.3.3.5 Real-World Dataset

Figure 3.5 depicts a much better parallel scalability for ParBaMiN compared to the baseline algorithm for the real-world dataset. The scalability of ParBaMiN improves as the problem size gets larger. Hereby, ParBaMiN scales up until 64 cores for the largest problem (**grow20480**) on both architectures.

As seen in Figure 3.5, ParBaMiN achieves significantly better speedup values for all test instances. ParBaMiN achieves a speedup of 8.98 and 14.29 for **grow20480** on 64 cores of the shared and distributed memory architectures, respectively. On the other hand, the baseline algorithm gives almost the same speedup of 1.29 and 1.20 for all matrices on 64 and 16 cores of the shared and distributed memory architectures, respectively.

3.3.4 Accuracy of the Numerical Results

In the earlier subsection, we have studied the parallel scalability of our algorithm on two different platforms compared to the baseline algorithm. In this subsection, we will look into the numerical accuracy of the results obtained using the same datasets given earlier. As the exact solution is not known, we will closely examine the relative residuals and the norm of the solution vectors obtained.

Figure 3.7 shows relative residual norms Γ_u of the underdetermined linear systems using the sequential SuiteSparseQR and ParBaMiN for 2, 4, 8, 16, 32 and 64 cores for the realistic, synthetic and real-world datasets. Relative residual

norm Γ_u is calculated as follows:

$$\Gamma_u = \frac{\|b - Ax\|_2}{\|b\|_2}. \quad (3.15)$$

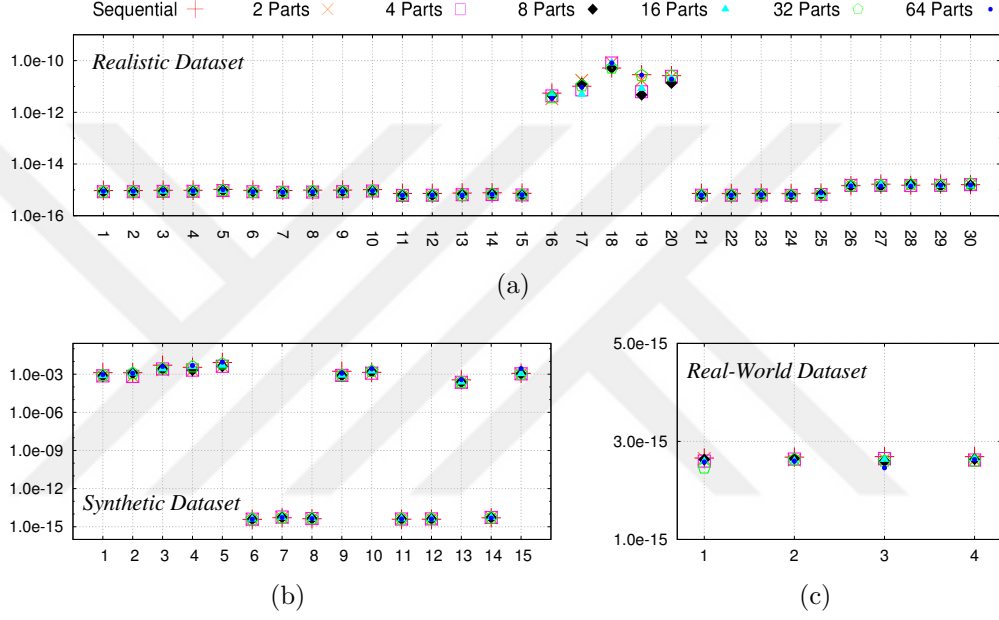


Figure 3.7: Relative residual norms of sequential algorithm and parallel ParBaMiN for the datasets

ParBaMiN produces very similar relative residuals with respect to the sequential baseline algorithm for almost all cases. In a few test instances only, the relative residuals differ slightly. We believe the reason for this is that the condition number of the matrix is always greater than the condition number of diagonal blocks. For instance when using 64 cores, each core applies the QR factorization on just one block which has the smallest condition number. When smaller number of cores are used, each core works on more than one block which have a much larger condition number than those of the constituent diagonal blocks. Furthermore, matrices having larger overlap sizes are also likely to have larger condition numbers compared to ones having smaller overlap sizes. This implies that the sequential algorithm is expected to give slightly larger relative residual norm compared to ParBaMiN.

Figure 3.8 illustrates the 2-norm of the solution vectors (i.e., $\|x\|_2$) obtained

using the sequential algorithm and ParBaMiN for 2, 4, 8, 16, 32 and 64 cores. As shown in the figure, ParBaMiN finds a solution that agrees with the solution vector obtained by the sequential QR factorization.

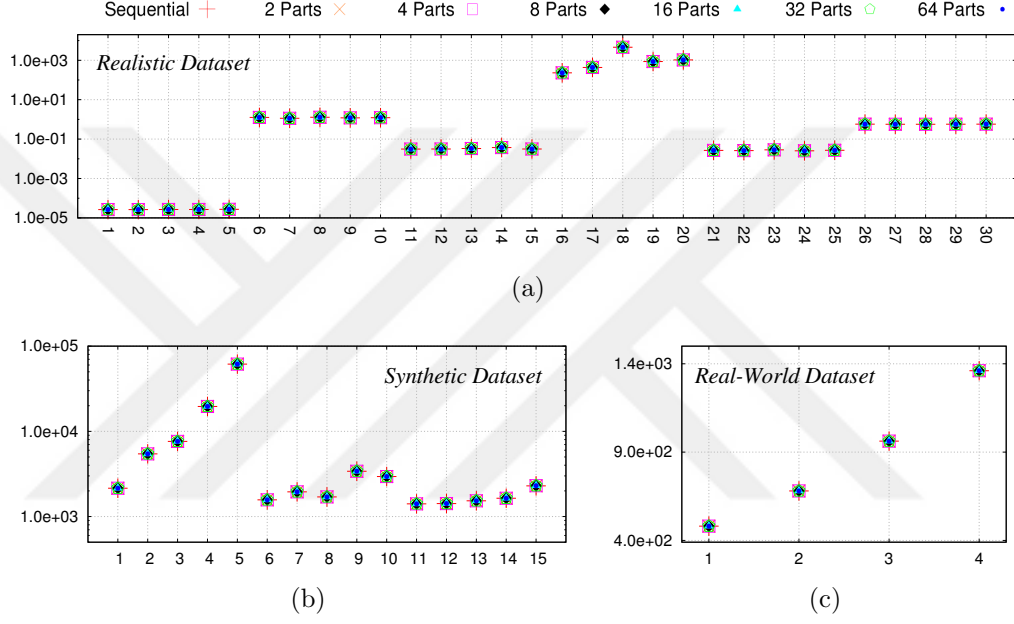


Figure 3.8: Sequential and parallel $\|x\|_2$ values for the datasets

In summary, the results show that ParBaMiN is not only more scalable, but also gives accurate results comparable to the existing algorithm based on the QR factorization. In Section 3.4, we will further study the accuracy of ParBaMiN.

3.4 Forward and Backward Errors

In this section, we further look into the numerical behavior of ParBaMiN compared to the Q and seminormal equations (SNE) methods. Norm-wise relative errors of both methods are bounded by $c\kappa_2(A)u$ [55], where c is a constant, u is the unit roundoff error and $\kappa_2(A)$ is defined as

$$\kappa_2(A) = \|A^+\|_2 \|A\|_2. \quad (3.16)$$

In [55], tighter error bounds for underdetermined linear systems are introduced. In their work, the term $\kappa_2(A)$ is replaced by

$$\text{cond}_2(A) = \|A^+\|A\|_2, \quad (3.17)$$

where

$$A^+ = A^T(AA^T)^{-1} \quad (3.18)$$

is the pseudoinverse of A . They note that $\text{cond}_2(A)$ could be arbitrarily smaller but could not be much bigger than $\kappa_2(A)$ and the component-wise condition number is defined as

$$\text{cond}_2(A, x) = (\|I - A^+A\|A^T\|A^{+^T}x\|_2 + \|A^+\|(|b| + \|A\|x\|_2)/\|x\|_2. \quad (3.19)$$

We have ran numerical experiments in MATLAB [56], in which the *double precision* and *single precision* have unit roundoff $u \approx 2.2 \times 10^{-16}$ and $u \approx 1.2 \times 10^{-7}$, respectively. Different from simulating single precision arithmetic in [55], we use true *single precision* arithmetic in our experiments. This gave us a chance to verify the results of [55] by conducting experiments with *single* precision arithmetic rather than simulating single precision. For forward error analysis, we regard the solution with double precision as the exact solution in our experiments.

Table 3.5: Forward and backward errors using three different algorithms for solving problems with different condition numbers.

Mtx	Metrics	Soln. Method	$p^N(\hat{x})$	$p^R(\hat{x})$	$p^C(\hat{x})$	$\gamma_2(\hat{x})$
1	$\kappa_2(A) = 1.03e4$	QR	8.52e-10	1.61e-10	5.64e-08	2.33e-04
	$\text{cond}_2(A) = 4.05e4$	SNE	5.11e-06	9.62e-07	3.66e-04	5.21e-01
	$\text{cond}_2(A, x) = 9.61e4$	ParBaMiN	6.84e-10	1.29e-10	4.52e-08	4.21e-02
2	$\kappa_2(A) = 1.02e5$	QR	7.86e-10	1.63e-10	5.62e-08	1.92e-03
	$\text{cond}_2(A) = 3.31e5$	SNE	1.11e-05	2.30e-06	5.32e-04	1.50e+01
	$\text{cond}_2(A, x) = 1.72e6$	ParBaMiN	1.15e-09	2.37e-10	8.15e-08	5.70e-02
3	$\kappa_2(A) = 1.05e6$	QR	5.82e-10	1.50e-10	5.63e-08	1.73e-02
	$\text{cond}_2(A) = 2.94e6$	SNE	2.11e-05	5.46e-06	1.91e-03	1.00e+00
	$\text{cond}_2(A, x) = 8.37e10$	ParBaMiN	6.97e-10	1.80e-10	6.73e-08	4.37e-02

The coefficient matrices in Table 3.5 are random matrices whose sizes are 200×395 with cascaded two dense blocks size of 100×200 with column overlap size of 5. In each matrix, the block matrices were generated using *randsvd* [70] routine. Singular values (σ_i) of block matrices were distributed geometrically, i.e., $\sigma_i = \alpha^{(1-i)/(n-1)}$, $i = 1 \dots n$, where $\alpha = \kappa_2(A)$ is a parameter [71] of *randsvd*. Entries of the right hand sides, b vectors, were normally distributed between $(0, 1)$.

We define

$$\gamma_2(\hat{x}) = \frac{\|\hat{x} - x\|_2}{\|x\|_2}, \quad (3.20)$$

where $\hat{x}$ is the computed solution. Three relative residuals [55] are

$$p^X(\hat{x}) = \max_i \frac{|b - A\hat{x}|_i}{(E_X|\hat{x}| + f_X)_i}, \quad X = N, R, C \quad (3.21)$$

where E_X and f_X are

$$\begin{aligned} \text{norm-wise} & : E_N = \|A\|_2 e_m e_n^T, & f_N &= \|b\|_2 e_m, \\ \text{row-wise} & : E_R = |A| e_m e_n^T, & f_R &= |b|, \\ \text{component-wise} & : E_C = |A|, & f_C &= |b|. \end{aligned} \quad (3.22)$$

Note that in Equation (3.22) $e_n = (1, 1, \dots, 1)^T \in \mathbb{R}^n$. In SNE method, iterative refinement was not used.

In Table 3.5, the error bounds of the work [55] are confirmed for SNE and QR methods. As given in [55], the error is estimated precisely within an order of magnitude by the equation

$$\gamma_2(\hat{x}) = \frac{\|\hat{x} - x\|_2}{\|x\|_2} \approx \text{cond}_2(A)u. \quad (3.23)$$

The results also show that for the given problems, ParBaMiN has better norm-wise relative error and relative residuals than the SNE method. Compared to the QR method, ParBaMiN gives comparable relative residuals for all test problems. The norm-wise relative errors are comparable to the QR method for the matrices with larger condition numbers.

3.5 Conclusion

In this chapter, a new parallel algorithm, ParBaMiN, is proposed and implemented for computing the minimum norm solution of sparse block diagonal column overlapped underdetermined systems. ParBaMiN exploits the special structure of the coefficient matrix by handling the block diagonals independently and concurrently. Parallel scalability of ParBaMiN was shown on two different parallel architectures: a shared memory (multicore) architecture and a distributed memory (cluster) architecture for realistic, synthetic and real-world datasets. In the best case, ParBaMiN achieves a speedup of 54.1 on 64 cores. Scalability performance of ParBaMiN degrades with increasing column overlap size as expected. Experimental results on all datasets show the validity of ParBaMiN on shared memory and distributed memory architectures.

We also study numerical stability of ParBaMiN. We experimentally compare and confirm the error bound of ParBaMiN against the QR factorization based techniques by using true single precision arithmetic.

In conclusion, ParBaMiN can be also considered as an scalable extension of any multithreaded general sparse QR factorization algorithm to distributed memory architectures for computing minimum 2-norm solution of underdetermined linear least squares problems.

Chapter 4

A Novel Graph Theoretical Row-Block Partitioning Method for Block Cimmino Algorithm

4.1 Introduction

In this chapter we study linear system of equations of the form

$$Ax = b, \tag{4.1}$$

where A is an $n \times n$ sparse nonsingular matrix, whereas x and b are column vectors of size n . Solving such linear system is studied by many works [72, 17, 73, 74, 75]. One way to solve such linear systems is to use row projection methods [76, 77]. These methods are originally iterative methods which compute the solution through successive projections in each iteration. Kaczmarz [78] and Cimmino [79] are two types of row projection method. Kaczmarz and Cimmino obtains the solution through a product and a sum of orthogonal projections, respectively. Cimmino algorithm has much more potential for parallelism than Kaczmarz [77, 80]. Convergence rate of Cimmino algorithm may be slow. In order to increase the convergence rate, block Cimmino [81] algorithm is proposed. There

are many works [80, 81, 82, 83, 84, 85, 41] which study iterative block Cimmino algorithm extensively. Recently, a pseudo-direct version of block Cimmino is proposed and implemented in [86, 87]. In this method, the row blocks are enforced to be orthogonal to each other by augmenting the sub-matrices. Since it is a direct method, its memory requirement is higher, yet it may be preferred if the number of iterations is exceptionally high in the iterative approach.

The system (4.1) is partitioned into k row blocks, with $k \leq n$, as follows:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}. \quad (4.2)$$

Here A_i is an $m_i \times n$ submatrix for $i = 1, 2, \dots, k$, where

$$n = \sum_{i=1}^k m_i. \quad (4.3)$$

The right-hand side vector b is also partitioned conformably.

Algorithm 3 describes the block Cimmino algorithm. In the algorithm A_i^+ is the Moore-Penrose pseudoinverse of A_i which is

$$A_i^+ = A^T(AA^T)^{-1}. \quad (4.4)$$

In the algorithm, A_i^+ is used for the notation of underdetermined matrix inverse and it is not explicitly computed. Note that δ_i can be computed independently in parallel without any communication, hence block Cimmino algorithm is naturally suitable for parallel computing platforms.

ALGORITHM 3: Block Cimmino algorithm

```

1 Initial guess  $x^{(0)}$ 
2  $t = 0$ 
3 while  $t=0,1,2,\dots$ , until convergence do
4   for  $i = 1 \dots k$  do
5      $\delta_i = A_i^+(b_i - A_i x^{(t)})$ 
6   end
7    $x^{(t+1)} = x^{(t)} + \omega \sum_{i=1}^k \delta_i$ 
8 end

```

We can conceive an iteration of block Cimmino as follows:

$$\begin{aligned}
x^{(t+1)} &= x^{(t)} + \omega \sum_{i=1}^k A_i^+ (b_i - A_i x^{(t)}) \\
&= \left(I - \omega \sum_{i=1}^k A_i^+ A_i \right) x^{(t)} + \omega \sum_{i=1}^k A_i^+ b_i \\
&= (I - H)x^{(t)} + \omega \sum_{i=1}^k A_i^+ b_i \\
&= Jx^{(t)} + \omega \sum_{i=1}^k A_i^+ b_i,
\end{aligned} \tag{4.5}$$

where J is the iteration matrix for block Cimmino algorithm and $H = I - J$ is the sum of projections defined by

$$H = \omega \sum_{i=1}^k A_i^+ A_i. \tag{4.6}$$

In order to obtain the projections in each iteration of block Cimmino, there exist several approaches to find the minimum 2-norm solution. These are normal equations, QR factorization, seminormal equations and the augmented system approach. These approaches are explained in Section 2.4.1.1. We use the augmented system approach since its distributed memory parallelism capability is higher than the other approaches by using a distributed parallel sparse direct solver [81]. In the augmented system approach, the solution of

$$A_i \delta_i = d_i, \tag{4.7}$$

where $d_i = b_i - A_i x^{(t)}$ is obtained by solving the linear system

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} \delta_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ d_i \end{pmatrix}. \quad (4.8)$$

Hence, the projection δ_i is obtained via the solution of the augmented system.

If the coefficient matrix A_i of the systems in Equation 4.6 is in the structure of column overlapped block diagonal, we can use ParBaMiN which is proposed in [50] and in Chapter 3. However, we assume more general case where A_i submatrices are sparse matrices without any restriction.

4.1.1 The Conjugate Gradient Acceleration of Block Cimmino Algorithm

Since the convergence rate of block Cimmino algorithm may be slow, the Conjugate Gradient (CG) iterative method is used to accelerate block Cimmino algorithm [80]. This work states that the robustness and the convergence rate of the CG accelerated Cimmino method are better than the preconditioned Generalized Minimum Residual (GMRES) and Conjugate Gradient on Normal Equations (CGNE).

Since H matrix in Equation (4.6) is symmetric and positive definite, we can solve the system

$$Hx = \xi \quad (4.9)$$

by using CG, where $\xi = \sum_{i=1}^K A_i^+ b_i$ and x is the original unknown vector of the system (4.1).

Note that ω is omitted in Equation (4.9) since it appears on both sides of the equation. Algorithm 4 describes pseudocode for the CG accelerated Block Cimmino method. Note that Algorithm 4 is the classical CG (see Section 2.4.2)

applied on the system given in Equation (4.9) in which $H = \sum_{u=1}^K A_u^+ A_u$.

ALGORITHM 4: Conjugate Gradient acceleration of block Cimmino algorithm

```

1 Choose  $x^{(0)}$ 
2  $r^{(0)} = \xi - \sum_{i=1}^k A_i^+ A_i x^{(0)}$ 
3  $p^{(0)} = r^{(0)}$ 
4 while  $t = 0, 1, 2, \dots$ , until convergence do
5    $z^{(t)} = \sum_{i=1}^k A_i^+ A_i p^{(t)}$ 
6    $\alpha^{(t)} = (r^{(t)T} r^{(t)}) / (p^{(t)T} z^{(t)})$ 
7    $x^{(t+1)} = x^{(t)} + \alpha^{(t)} p^{(t)}$ 
8    $r^{(t+1)} = r^{(t)} - \alpha^{(t)} z^{(t)}$ 
9    $\beta^{(t)} = (r^{(t+1)T} r^{(t+1)}) / (r^{(t)T} r^{(t)})$ 
10   $p^{(t+1)} = r^{(t+1)} + \beta^{(t)} p^{(t)}$ 
11 end

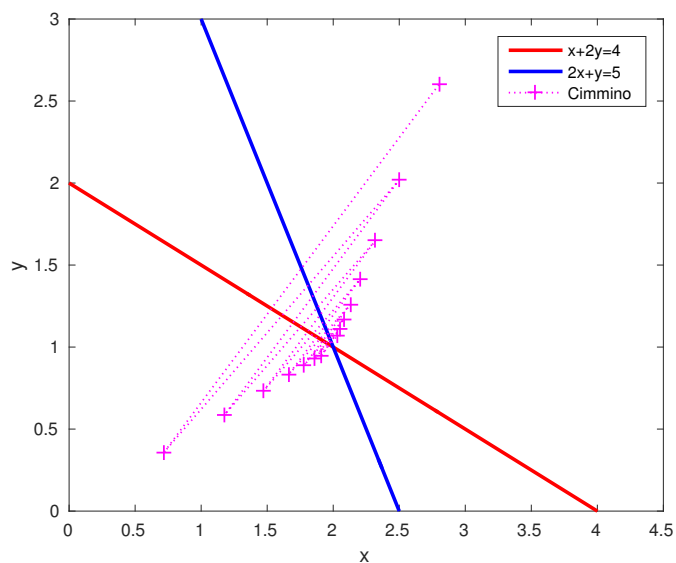
```

At line 5 in Algorithm 4, k independent underdetermined systems are solved. These independent systems can be solved in parallel and the communication is only need for the summation of projections.

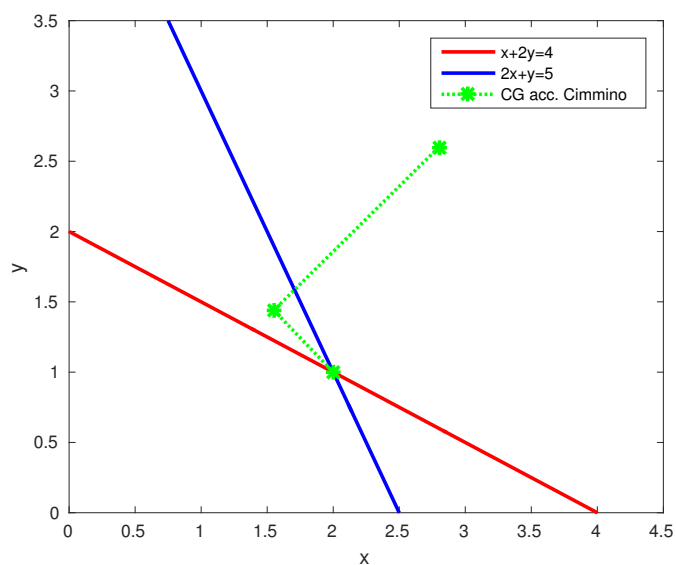
As an example, in Figure 4.1, we track the steps of the algorithms for solving a system with 2 equations that are

$$\begin{aligned} x + 2y &= 4 \\ 2x + y &= 5, \end{aligned} \tag{4.10}$$

where the exact solution is $x = 2$ and $y = 1$. Figures 4.1a and 4.1b show the solution of each iteration for classical Cimmino algorithm and CG accelerated Cimmino algorithm, respectively. As seen in the figures, the CG accelerated Cimmino algorithm reaches the solution faster, after 3 iterations, whereas the classical Cimmino algorithm reaches the solution after 20 iterations.



(a) Classical Cimmino algorithm



(b) CG accelerated Cimmino algorithm

Figure 4.1: Comparison of classical Cimmino algorithm and Conjugate Gradient accelerated Cimmino algorithm

4.1.2 The effect of partitioning

The convergence rate of block Cimmino is related to the eigenvalue distribution of the iteration matrix, which is determined by the partitioning of the coefficient matrix into row blocks. In addition, convergence rate of Cimmino algorithm only depends on the row block partition of A and independent of any column permutation [41].

Let QR decomposition (see Section 2.4.1) of A_i^T is given by

$$A_i^T = Q_i R_i, \quad (4.11)$$

where the matrices Q_i and R_i have sizes of $n \times m_i$ and $m_i \times m_i$, respectively. Then, we have

$$\begin{aligned} H &= \sum_{i=1}^k A_i^T (A_i A_i^T)^{-1} A_i \\ &= \sum_{i=1}^k Q_i Q_i^T \\ &= (Q_1, \dots, Q_k)(Q_1, \dots, Q_k)^T. \end{aligned} \quad (4.12)$$

From the theory of singular value decomposition [88], the eigenvalue spectrum of matrix $(Q_1, \dots, Q_K)(Q_1, \dots, Q_K)^T$ is the same as the spectrum of matrix $(Q_1, \dots, Q_K)^T(Q_1, \dots, Q_K)$ [88]. Therefore, H can be shown as

$$\begin{pmatrix} I_{m_1 \times m_1} & Q_1^T Q_2 & \dots & Q_1^T Q_K \\ Q_2^T Q_1 & I_{m_2 \times m_2} & \dots & Q_2^T Q_K \\ \vdots & \dots & \ddots & \vdots \\ Q_K^T Q_1 & \dots & \dots & I_{m_K \times m_K} \end{pmatrix}. \quad (4.13)$$

The principal angles between the subspaces A_i and A_j is represented by the singular values of each off-diagonal block matrix $Q_i^T Q_j$ [89]. Decreasing the principal angles between subspaces makes the row blocks more orthogonal to each other.

Hence, by the Gershgorin theorem [90], smaller the off-diagonals in Equation (4.13) lead to more eigenvalues of H to be clustered around 1. Therefore, the orthogonality between A_i blocks determine the convergence rate of the block Cimmino method. In other words, if A_i is orthogonal to A_j , than every row r_i in A_i is orthogonal to every row r_j in A_j . Note that, the row inner products between blocks would be arbitrarily small if A_i blocks are more orthogonal to each other. Therefore, the required number of iterations for convergence is expected to be reduced.

Drummond et al. [41] attempt to find good row-block partitions. They propose three partitioning strategies. However, these methods consider only the structural orthogonality, such as applying a permutation that does not consider the magnitude of the matrix elements like Reverse Cuthill-McKee (RCM) or they consider the numerical orthogonality of the subspaces by dropping small elements in magnitude during a preprocessing stage. Furthermore, they proposed a hypergraph partitioning strategy [41] in order to attain better convergence rate. They provide a comprehensive comparison of RCM based strategies with hypergraph partitioning strategy and they state that hypergraph partitioning strategy performs better than the RCM based methods.

In this chapter, we present a graph theoretical row-block partitioning method for block Cimmino algorithm. Using this row-block partitioning, we aim to reduce the required number of iterations in CG significantly. The proposed partitioning scheme takes numerical orthogonality of A into account via exploiting the row inner product values of A . An undirected graph model and a row-block partitioning method on this graph model is proposed. Objective of this partitioning encapsulates minimizing the sum of inner products between row blocks. We also conduct experiments on a large set of matrices that arise in real life applications. The experimental results confirm the validity of the proposed partitioning method against the baseline methods.

The rest of this chapter is organized as follows. We present the proposed partitioning method and its implementation details in Section 4.2. In Section 4.3, we compare the proposed partitioning strategy against the baseline partitioning

strategies in terms of convergence rate, preprocessing time and parallel execution time. Furthermore, in Section 4.3.6 we demonstrate the strong scalability results using up to 256 cores on the distributed memory architecture. In Section 4.3.7, we introduce a parallel partitioning method in order to reduce the preprocessing overhead of the proposed partitioning method.

4.2 A Novel Row-Block Partitioning method for Block Cimmino Algorithm

In this section, first we define row-to-row inner product graph for A and then model the row partitioning/reordering problem as a graph partitioning problem on this graph. Here and hereafter this model will be shortly referred to as the row inner-product graph model. Furthermore, we show the impact of the row-block partitioning on the eigenvalue spectrum of the iteration matrix. Finally, we give the implementation details of the proposed graph model and the proposed partitioning method.

4.2.1 Row Inner-product Graph Model

In the row inner-product graph $\mathcal{G}_{\text{RIP}}(A) = (\mathcal{V}, \mathcal{E})$ of matrix A , rows of A are represented by vertices and nonzero inner products between rows are represented by edges. In other words, for each row $\mathbf{r}_i$ of A there exists a vertex $v_i \in \mathcal{V}$ and for each nonzero inner product of $\mathbf{r}_i$ and $\mathbf{r}_j$ there exists an edge $(v_i, v_j) \in \mathcal{E}$. Then, the set $\mathcal{E}$ is defined as

$$\mathcal{E} = \{(v_i, v_j) \mid \mathbf{r}_i \mathbf{r}_j^T \neq 0\}. \quad (4.14)$$

Vertex weights ($w(v_i)$) can be associated as

$$w(v_i) = 1 \quad \text{or} \quad w(v_i) = \text{nnz}(\mathbf{r}_i). \quad (4.15)$$

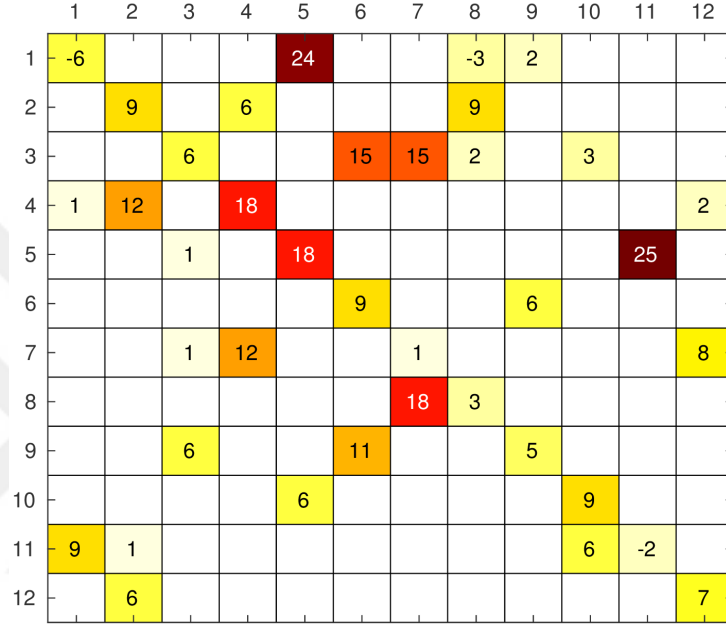
Edge costs ($cost(v_i, v_j)$) for each edge $(v_i, v_j) \in \mathcal{E}$ is associated with

$$cost(v_i, v_j) = |\mathbf{r}_i \mathbf{r}_j^T|. \quad (4.16)$$

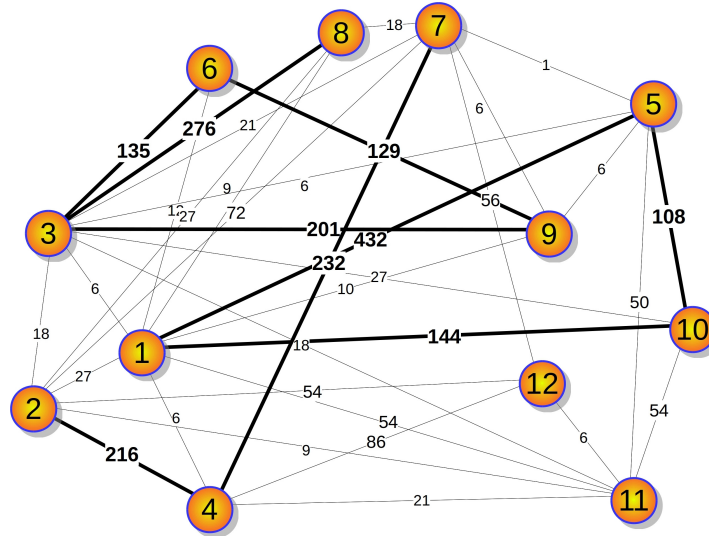
Note that the adjacency matrix representation of the row inner-product graph of A is equivalent to the standard graph representation of $C = AA^T$. Moreover, $cost(v_i, v_j)$ is equal to the absolute value of nonzero $|c_{ij}|$ of matrix C .

Figure 4.2a illustrates an example matrix which is a sparse matrix of size 12×12 with 38 nonzeros. In the figure, we use darker color for the nonzero values whose absolute values are higher. The proposed row inner-product graph $\mathcal{G}_{\text{RIP}}$ representation of the sample matrix is shown in Figure 4.2b. $\mathcal{G}_{\text{RIP}}$ has 12 vertices which correspond rows of the matrix. $\mathcal{G}_{\text{RIP}}$ contains an edge for each nonzero row-inner product, therefore there are 35 edges in the graph. For instance, the inner product of rows $\mathbf{r}_3$ and $\mathbf{r}_6$ is nonzero and results of the inner product is $\mathbf{r}_3 \mathbf{r}_6^T = 9 \times 15 = 135$. Therefore, $\mathcal{G}_{\text{RIP}}$ has an edge between v_3 and v_6 with $cost(v_3, v_6) = 135$. On the other hand, the inner product of rows $\mathbf{r}_{10}$ and $\mathbf{r}_{12}$ is zero, therefore there is no (v_{10}, v_{12}) edge in the graph. In Figure 4.2b, we use thick lines in order to make high inner-product values more visible for edges with cost larger than 100.

Figure 4.3 illustrates the absolute values of matrix $C = AA^T$ for the sample matrix in Figure 4.2a. Diagonal entries of C does not affect the $\mathcal{G}_{\text{RIP}}$ (self-loop edges in the graph), therefore these values are discarded in the figure. We color black the cells whose values are larger than 100 in order to indicate these large values. Note that the row inner-product graph in Figure 4.2b is the standard graph representation of the matrix in Figure 4.3 and edge costs of the graph represents the value of the respective nonzero entries of the matrix C . For example, the edge (v_3, v_6) with $cost(v_3, v_6) = 135$ is incurred by the nonzero value of c_{36} in matrix C .



(a) Sample matrix A



(b) Row inner-product graph $\mathcal{G}_{\text{RIP}}$ of A

Figure 4.2: Row inner-product graph model

	1	2	3	4	5	6	7	8	9	10	11	12
1		27	6	6	432	12		9	10	144	54	
2	27		18	216			72	27			9	54
3	6	18			6	135	21	276	201	27	18	
4	6	216					232				21	86
5	432		6				1		6	108	50	
6	12		135						129			
7		72	21	232	1			18	6			56
8	9	27	276				18					
9	10		201		6	129	6					
10	144		27		108						54	
11	54	9	18	21	50					54		6
12		54		86			56				6	

Figure 4.3: $C = AA^T$ matrix for the sample matrix in Figure 4.2a

4.2.2 Row-block partitioning via partitioning the row inner-product graph

In this subsection, we introduce row-block partitioning problem on the row inner-product graph model. We show some examples in order to indicate how a “good” row-block partitioning should be. Finally we compare the proposed partitioning method against uniform partitioning method in terms of the eigenvalue spectrum of the iteration matrix.

Consider a k -way partition $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_k\}$ of $\mathcal{G}_{\text{RIP}}$. Π can be used for finding a partial permutation on the rows of A . Thus, partially permuted matrix A^Π is obtained by using this permutation. Let P be a row permutation matrix

and it is obtained by the k -way partition Π ,

$$A^\Pi = PA = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_i \\ \vdots \\ \mathcal{R}_k \end{bmatrix}, \quad (4.17)$$

where $\mathcal{R}_i$ denotes the set of rows of matrix A_i . In k -way partition Π , each vertex set $\mathcal{V}_i$ corresponds a row-block set $\mathcal{R}_i$. The vertex order in a set $\mathcal{V}_i$ is arbitrary. Therefore the row order in a row-block set $\mathcal{R}_i$ is not important and can be decided randomly. Column permutation of the matrix has not an impact upon the convergence of the block Cimmino algorithm [41], therefore we preserve the original column ordering of the matrix.

In partition Π of $\mathcal{G}_{\text{RIP}}$, the weight of part $\mathcal{V}_i$ is denoted as W_i . W_i is defined as

$$W_i = |\mathcal{R}_i| \quad \text{or} \quad W_i = \sum_{r_d \in \mathcal{R}_i} nnz(r_d). \quad (4.18)$$

So, the constraint of maintaining balance on vertex parts corresponds to either maintaining balance on the number of rows in $\mathcal{R}_i$ or maintaining balance on the number of nonzeros in $\mathcal{R}_i$.

In partition Π of $\mathcal{G}_{\text{RIP}}$, a nonzero inner product $r_d r_e^T > 0$ between row blocks $\mathcal{R}_i$ and $\mathcal{R}_j$, where $r_d \in \mathcal{R}_i$ and $r_e \in \mathcal{R}_j$, is represented by a cut edge $(v_e, v_d) \in \mathcal{E}_{\text{cut}}$ between vertex parts $\mathcal{V}_i$ and $\mathcal{V}_j$. So, the cutsize is given by

$$\text{cutsiz}(\Pi) = \sum_{(v_d, v_e) \in \mathcal{E}_{\text{cut}}} \text{cost}(v_d, v_e) = \sum_{1 \leq i < j \leq k} \sum_{\substack{r_d \in \mathcal{R}_i \\ r_e \in \mathcal{R}_j}} |r_d r_e^T|. \quad (4.19)$$

The objective of minimizing cutsize of the graph partitioning encapsulates minimizing the sum of inner products between different row blocks. As a result, the proposed partitioning method increases the orthogonality among row blocks through minimizing the inner products between row blocks. By this way, we expect the convergence of the block Cimmino to be improved.

Figures 4.4 and 4.5 are presented in order to explain the row inner-product graph partitioning method for row-block partitioning of A . Figure 4.4a depicts a

uniform 3-way row-block partition $\{\mathcal{R}_1^u, \mathcal{R}_2^u, \mathcal{R}_3^u\}$ of the sample matrix A . In the uniform partitioning, four consecutive rows in the original order constitute the row blocks. Figure 4.4b depicts the 3-way vertex partition $\Pi^u(\mathcal{V}) = \{\mathcal{V}_1^u, \mathcal{V}_2^u, \mathcal{V}_3^u\}$ of $\mathcal{G}_{\text{RIP}}$ which is induced by the uniform 3-way row-block partition. On the other hand, Figure 4.5a depicts a “good” 3-way vertex partition of $\Pi^g(\mathcal{V})$ of $\mathcal{G}_{\text{RIP}}$ which is obtained by multilevel graph partitioning tool METIS [91]. Figure 4.5b depicts the permuted A^Π matrix. In the figure, 3-way row-block partition is shown by dashed lines. This 3-way row-block partition is induced by the 3-way vertex partition given in Figure 4.5a. Here and hereafter, the terminology of $\mathcal{G}_{\text{RIP}}$ partitioning, graph partitioning and “good” partitioning are used interchangeably and they refer to the same partitioning strategy.

The row-block partitions in Figures 4.4 and 4.5, have a good balance on the number rows in row blocks. However, difference of cutsize in these two partitioning is quite large. As seen in Figure 4.4b, cutsize of uniform partitioning is 2201, whereas cutsize of graph partitioning is 254 as seen in Figure 4.5a. As also seen in Figure 4.5a, all thick edges remain inside of the parts.

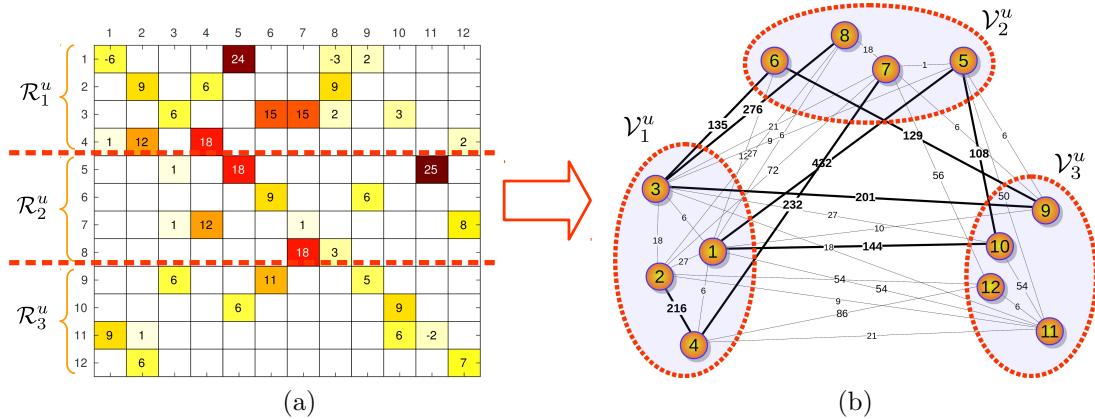


Figure 4.4: (a) Uniform 3-way row partition of A and (b) 3-way vertex partition of $\mathcal{G}_{\text{RIP}}(A)$ induced by (a)

Figure 4.6 shows the quality difference of the partitioning methods by using the matrix $C = AA^T$. Figures 4.6a and 4.6b show the checkerboard partitioning

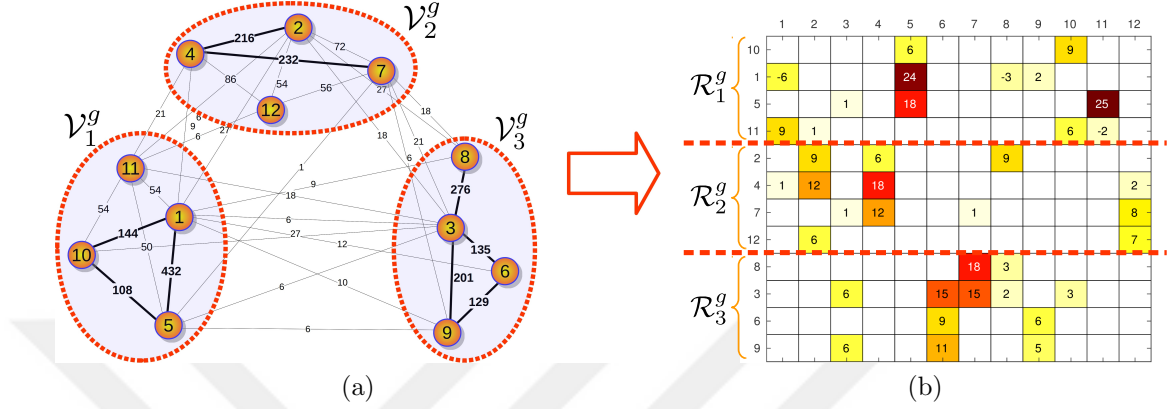


Figure 4.5: (a) “Good” 3-way vertex partition of $\mathcal{G}_{\text{RIP}}(A)$ and (b) 3-way row-block partition of A induced by (a)

of matrix C induced by the row permutation which is obtained by uniform partitioning and $\mathcal{G}_{\text{RIP}}$ partitioning, respectively. We partition the rows and columns of C matrices accordingly to the row-block partition on A by using each partitioning method. The off-diagonal block entries of C matrix reflect the edge costs which contribute the cutsizes of the partition. Comparison of Figures 4.6a and 4.6b shows that, graph partitioning based row-block partition of A produces a C matrix whose off-diagonal nonzero values are much smaller than the C matrix which is produced by a row-block partition induced by the uniform partitioning.

We introduce IP matrix in order to explicitly show the partitioning quality in terms of row inner products. In other words, IP matrix shows the sum of inner products of different row blocks of A . Tables 4.1 and 4.2 respectively show IP matrices for uniform partitioning and $\mathcal{G}_{\text{RIP}}$ partitioning. For example, the cell ip_{12} of the matrix $\text{IP}(\Pi^u)$ depicts the sum of the inner products between $\mathcal{R}_1^u$ and $\mathcal{R}_2^u$ for uniform row-block partitioning of A . The sum of row inner-product between different parts can be easily seen by these tables. Moreover, we introduce term of sumIP which denotes the sum of row inner-product between different parts. Thus, sumIP for the uniform partitioning

$$\begin{aligned} \text{sumIP}(\Pi^u) &= \text{ip}_{12}^u + \text{ip}_{13}^u + \text{ip}_{23}^u \\ &= 1222 + 624 + 355 = 2201 \end{aligned} \tag{4.20}$$

	1	2	3
1		1222	624
2	1222		355
3	624	355	

Table 4.1: $\text{IP}(\Pi^u)$

	1	2	3
1		70	94
2	70		90
3	94	90	

Table 4.2: $\text{IP}(\Pi^g)$

whereas for $\mathcal{G}_{\text{RIP}}$ partitioning

$$\begin{aligned} \text{sumIP}(\Pi^g) &= \text{ip}_{12}^g + \text{ip}_{13}^g + \text{ip}_{23}^g \\ &= 70 + 94 + 90 = 254. \end{aligned} \quad (4.21)$$

Furthermore, sumIP is also equal to the cutsize of the partitioning.

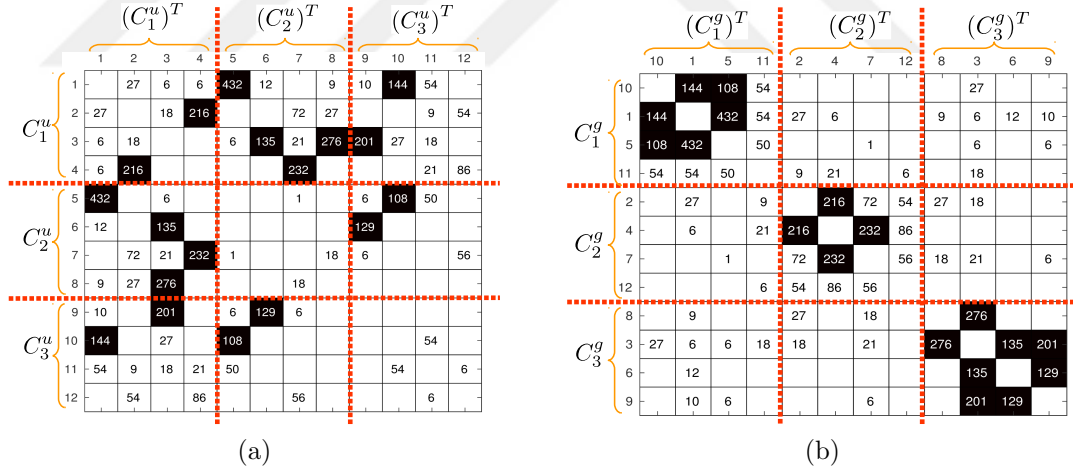


Figure 4.6: 3×3 checkboard partition of matrix $C = AA^T$ induced by 3-way (a) uniform (Figure 4.4a) and (b) $\mathcal{G}_{\text{RIP}}$ (Figure 4.5b) block-row partitions of matrix A .

Figures 4.7a and 4.7b show the eigenvalue distribution of the iteration matrix of A in CG accelerated block Cimmino algorithm for uniform partitioning and graph partitioning, respectively. As seen in Figure 4.7b, $\mathcal{G}_{\text{RIP}}$ partitioning makes the eigenvalues better clustered around 1, which means more numerically orthogonal row blocks, than the uniform row-block partitioning. This finding confirms that the reduction in sumIP and cutsize in the partitioning corresponds to more

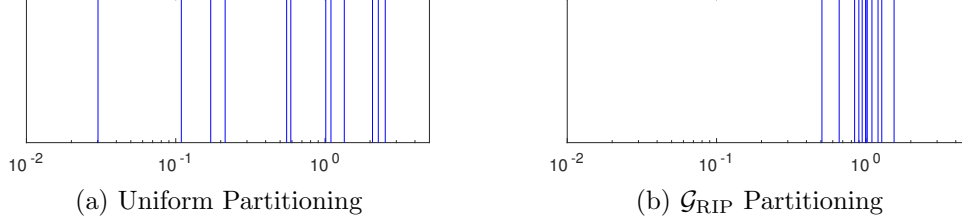


Figure 4.7: Eigenvalue spectrum of H for row-block partitionings on A

clustered eigenvalues around 1. Therefore, we expect a faster convergence rate in the block Cimmino algorithm by using the $\mathcal{G}_{\text{RIP}}$ partitioning method.

4.2.3 Implementation Details

In this subsection, we give implementation details of the proposed partitioning method. The proposed method consists of two parts; constructing $\mathcal{G}_{\text{RIP}}$ and k-way partitioning of the $\mathcal{G}_{\text{RIP}}$.

4.2.3.1 Constructing of row inner-product graph

As mentioned in Section 4.2, standard graph representation of the resulting matrix $C = AA^T$ has the same characteristics with $\mathcal{G}_{\text{RIP}}$ representation of A . Each off-diagonal nonzero in C indicates an edge in $\mathcal{G}_{\text{RIP}}$ of A and also absolute value of each off-diagonal nonzero determines the cost of an edge in $\mathcal{G}_{\text{RIP}}$. For this reason, we exploit a sparse matrix-matrix multiplication (SpGEMM) [92, 93] kernel to obtain $C = AA^T$. There are some additional advantages of using SpGEMM kernel, e.g, they are very efficient and also they are amenable for parallel execution.

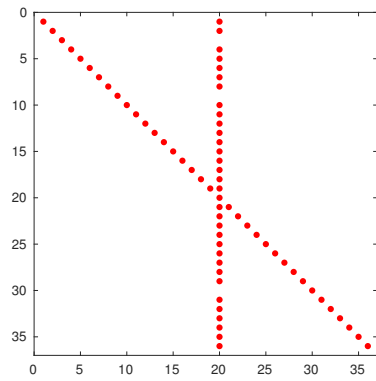
Constructing $\mathcal{G}_{\text{RIP}}$ for matrices with dense columns, may consume a huge amount of time and memory. For this reason, we introduce a sparsening technique to reduce time and memory consumption for matrices with dense columns. This

problem stems from the fact that when a column in A has c nonzeros, corresponding $C = AA^T$ matrix will have at least c^2 nonzeros. For example, a sample matrix A is shown in Figure 4.8a. This matrix is a 36×36 sparse matrix which has a dense column having 33 nonzero elements. As seen in Figure 4.8b, matrix $C = AA^T$ is very dense since it has 1092 (larger than $33^2 = 1089$) nonzero elements, hence the nonzero density of C is 84%. Clearly, the resulting dense matrix increases the memory requirement and computation cost of SpGEMM. Besides, large number of edges in $\mathcal{G}_{\text{RIP}}$ increase the time requirement for partitioning $\mathcal{G}_{\text{RIP}}$. In order to alleviate this problem, we propose the following methodology for sparsening C .

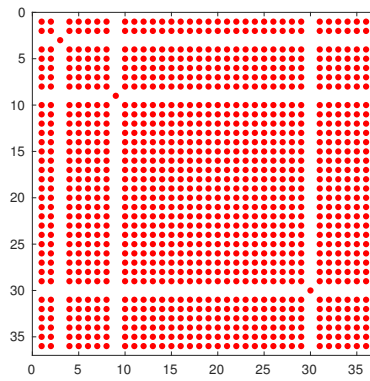
For the sparsening methodology, we introduce a new matrix $\tilde{A}$. If a column has more nonzero elements than $\sqrt{n}$ then we call it *dense column*. In the methodology, $\tilde{A}$ keeps nonzero elements of all columns of A except for the dense columns. For dense columns, $\tilde{A}$ keeps only the largest $\sqrt{n}$ values in magnitude for the respective columns. We try to alleviate excessive deterioration on numerical characteristic of AA^T by keeping largest magnitude $\sqrt{n}$ values in the columns. By this methodology, we limit the generated number of nonzero for each column at most $\sqrt{n}^2$ for the computation of $\tilde{C} = \tilde{A}\tilde{A}^T$ matrix. For instance, Figure 4.8c illustrates an example scenario of a sparsened matrix $\tilde{A}$ for the original matrix A in Figure 4.8a. As seen in Figure 4.8d the resulting $\tilde{C}$ matrix is much sparser than C in Figure 4.8b. While the nonzero density of $\tilde{C}$ is 5%, the nonzero density of C is 84%. Eventually, this methodology also makes sparser $\mathcal{G}_{\text{RIP}}$ which can be constructed and partitioned more easily. We note that matrices $\tilde{A}$ and $\tilde{C}$ are used only in the partitioning stage and after the partitioning stage these matrices are ignored.

4.2.3.2 Partitioning Row Inner-product Graph

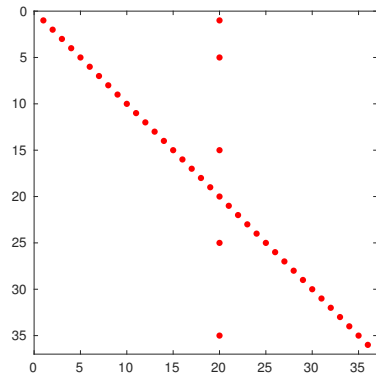
In order to partition $\mathcal{G}_{\text{RIP}}$ we use multilevel graph partitioning tool METIS [91]. Since METIS can not use floating point weights for edges, we cast the edge costs of $\mathcal{G}_{\text{RIP}}$ from floating point to integer values. The following function determines



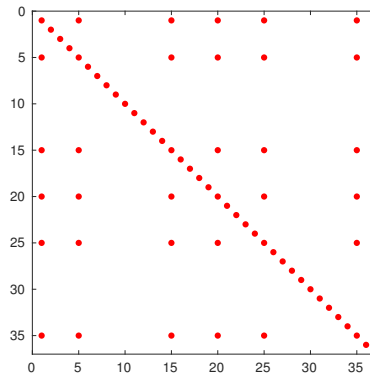
(a) A



(b) $C = AA^T$



(c) $\tilde{A}$



(d) $\tilde{C} = \tilde{A}\tilde{A}^T$

Figure 4.8: Nonzero patterns of A , $C = AA^T$, $\tilde{A}$ and $\tilde{C} = \tilde{A}\tilde{A}^T$.

the edge weights ($wgt(v_i, v_j)$) which are used as an input for METIS

$$wgt(v_i, v_j) = 1 + \left\lfloor 10^6 \times \frac{cost(v_i, v_j) - cost_{min}}{cost_{max} - cost_{min}} \right\rfloor, \quad (4.22)$$

where $cost(v_i, v_j)$ is the cost edges in $\mathcal{G}_{RIP}$ and $cost_{max}$ and $cost_{min}$ is the maximum and minimum edge costs, respectively. The function scales the floating point range $[cost_{min}, cost_{max}]$ to the integer range $[1, 10^6 + 1]$.

Ideally, we expect an uniform distribution for integer edge weight after the scaling formula. However, we observe that for some type of problems the edge weight distribution is clustered on the extremes. For example, in some instances, most of the edge weights are stuck in 1 because the extremely large $cost_{max}$ deteriorate uniformity of the distribution of edge weights in the integer scale. Differentiating the small edge cost is also important in order to reach better partitioning and convergence rate. To handle this problem, we pre-scale all edge costs in $\mathcal{G}_{RIP}$ by an square root function. Therefore, we expect the distance between $cost_{min}$ and $cost_{max}$ is decreased and more uniform distribution of $wgt(v_i, v_j)$.

4.3 Numerical Experiments

4.3.1 Experimental Framework

We use some implementations available in open-source package ABCD Solver v1.0-beta [87]. These are the implementations of matrix scaling [94] and obtaining row projections with MUMPS [27]. We changed only the parameters of the matrix scaling algorithm by setting NB1, NB2 and NB3 parameters to 10, 20 and 20, respectively, as in one of the previous version of ABCD Solver (version on 2014-09-15). Apart from these, we implement the parallel classical CG accelerated block Cimmino algorithm (see Algorithm 4). In the experiments, we use normwise backward error [95] as the stopping criterion for CG. The normwise backward error at iteration t ($\omega^{(t)}$) is obtained by

$$\omega_t = \frac{\|Ax^{(t)} - b\|_\infty}{\|A\|_\infty \|x^{(t)}\|_1 + \|b\|_\infty} < 10^{-10}. \quad (4.23)$$

In the experiments, the maximum number of iterations is 5,000 for CG.

In the experiments, we compare the performance of row-block partitioning strategies in terms of the number of iterations, parallel solution time, preprocessing time and total execution time. We compare the proposed $\mathcal{G}_{\text{RIP}}$ partitioning method against two baseline row-block partitioning methods which are introduced in [41]. The first baseline method is uniform partitioning method (UP) which partitions the coefficient matrix into given number of row blocks. The row blocks are almost equal sizes in terms of the number of rows.

The second baseline row-block partitioning method in ABCD Solver is hypergraph partitioning method (HP) [41]. In the method, they adopt column-net hypergraph model [96]. In their model, rows and columns of the sparse matrix A are represented by vertices and nets (hyperedges), respectively and weight of vertices and nets are 1. For each column in A , there exist a net which connects a set of vertices corresponding to the rows that share a nonzero with the respective column. The constraint of the partitioning is to maintaining balance on the size of row blocks in terms of number of rows. The objective of the partitioning is to minimize the number of columns which contain nonzeros in more than one row block.

In ABCD Solver, multilevel hypergraph partitioning tool PaToH [97] is used for HP method. We use 50% and 100% as parameters respectively for final imbalance and initial imbalance parameters as specified in ABCD Solver. We use default values for the other parameters in PaToH.

In the proposed partitioning method GP, we use the multilevel graph partitioning tool METIS in order to partition the $\mathcal{G}_{\text{RIP}}$ of A . In METIS, the load imbalance parameter is set to 10% and direct k-way partitioning scheme is used. In GP method we use unit vertex weights in $\mathcal{G}_{\text{RIP}}$, therefore GP method maintain balance on the number of rows for each row blocks as in HP method. We use default values for the other parameters in METIS. As PaToH and METIS use randomized algorithms, the results of the geometric mean of 10 experiments with different seeds are reported.

Here and hereafter, the terms of UP, HP and GP is used both for the partitioning method and the solution of a system in which the coefficient matrix is partitioned by the respective method.

Experiments are conducted on a 64-core shared memory machine which has 4 AMD Opteron 6376 processors each of has 16 cores running at 2.3GHz. We use MPI implementation of OpenMPI v1.10.2. In the experiments, parallel running times are obtained by using 32 cores due to memory bandwidth congestion.

4.3.2 Effect of Row-block Partitioning on the Eigenvalue Spectrum

The eigenvalue spectrum of the iteration matrix determine the convergence rate of the CG accelerated block Cimmino algorithm. In the block Cimmino algorithm, most of the eigenvalues of H are clustered around 1 naturally. However, some eigenvalues can be at edge of the spectrum.

In this subsection, using real problems we study the effect of the row-block partitionings on the eigenvalue spectrum of H by comparing the GP method against two baseline method UP and HP. In these experiments, we use **sherman3** and **LeGresley_4908** matrices from SuiteSparse Matrix Collection [60]. **sherman3** and **LeGresley_4908** are unsymmetric sparse matrices of size $5,005 \times 5,005$ with 20,033 nonzeros and $4,908 \times 4,908$ with 30,482 nonzeros, respectively. **sherman3** and **LeGresley_4908** arise in Computational Fluid Dynamics problem [98] and Power Network problem, respectively. These matrices is partitioned into 8 row blocks by using the three partitioning strategies.

Figure 4.9 shows eigenvalue spectrums of H of **sherman3** and **LeGresley_4908** for each partitioning methods. Figure 4.9 also depicts the number of CG iteration with minimum eigenvalue (λ_{min}) and maximum eigenvalue (λ_{max}) for each test instances. Figures 4.9a, 4.9b and 4.9c show the eigenvalue spectrum of the iteration matrices of **sherman3** for UP, HP and GP partitioning methods, respectively. Similarly, Figures 4.9d, 4.9e and 4.9f show the eigenvalue spectrum of the

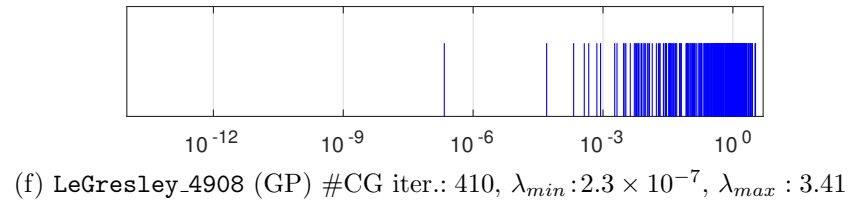
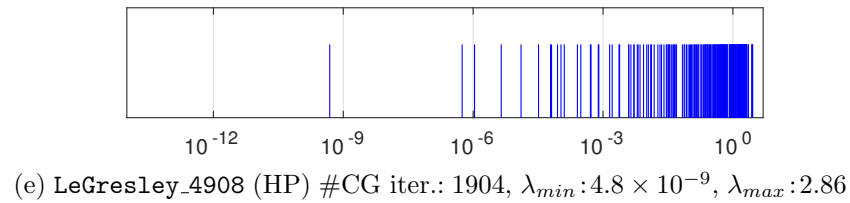
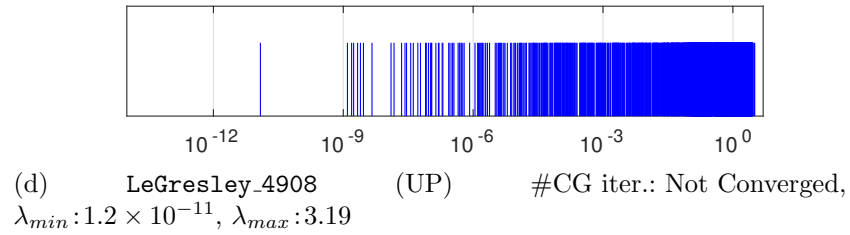
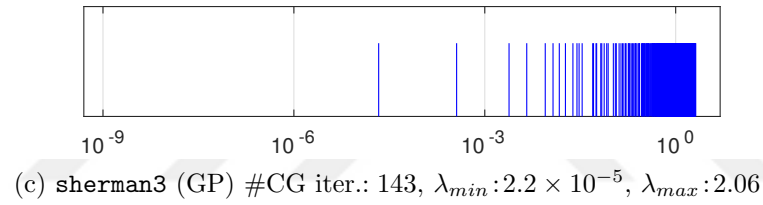
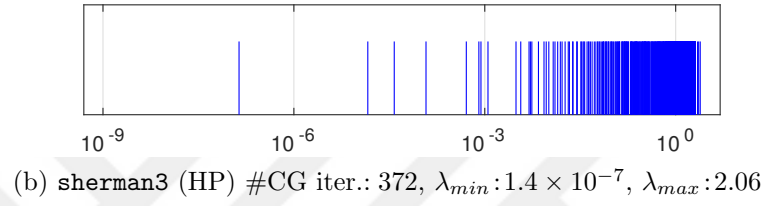
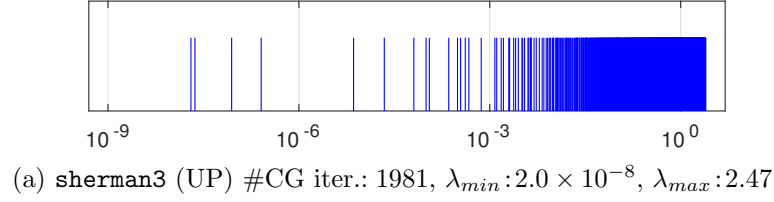


Figure 4.9: The eigenvalue spectrum of H for **sherman3**, **LeGresley_4908**

iteration matrices of `LeGresley_4908` for UP, HP and GP partitioning methods, respectively. As shown in the figures, GP partitioning method gives the best cluster of eigenvalues around 1 compared to the other partitioning strategies. GP partitioning method also gives rise to the largest λ_{min} which is closest to 1. As a comparison of the partitioning method on the problem of `sherman3` in terms of the number of CG iterations, GP partitioning method requires the smallest (143) CG iteration, whereas HP and UP methods require 372 and 1981 CG iterations, respectively. Therefore, the improvement on the eigenvalue spectrum is reflected to decrease in the required number of CG iteration as expected.

4.3.3 Dataset for Performance Analysis

For extensive performance analysis of the partitioning methods, we selected all non-singular unsymmetric matrices from the SuiteSparse matrix collection [60]. The dimensions of matrices are between 50,000 and 5,000,000. The number of matrices based on this criteria come out to be 112. `HV15R` and `cage14` matrices are excluded from the dataset due to insufficient memory space during the factorization. In 80 matrices out of 110, at least one of the partitioning methods was able converge before 5,000 CG iterations. These 80 matrices are shown in Table 4.3 with their properties. We use the same procedure with ABCD Solver to create right hand side vectors for the systems. The procedure multiply the coefficient matrices with a vector whose elements are randomly chosen in the range of $[1/99, 100/99]$. All test matrices are partitioned into 256 row blocks for all of the partitioning methods.

4.3.4 Convergence and Parallel Performance

Tables 4.4 and 4.5 respectively show the number of CG iteration and parallel CG times for each matrix. In these tables the best result for each test instances is shown in bold. In the tables, **F** and **S** denote the failures caused by reaching the maximum number of iterations and factorizing a singular block, respectively.

Table 4.3: Properties of the test matrices which are sorted in descending order according to the number of nonzeros. (n : number of rows/columns, nnz : number of nonzeros)

Matrix name	n	nnz	Matrix name	n	nnz
ML_Geer	1,504,002	110,686,677	para-10	155,924	2,094,873
RM07R	381,689	37,464,962	para-6	155,924	2,094,873
ML_Laplace	377,002	27,582,698	cage12	130,228	2,032,536
Transport	1,602,111	23,487,281	ASIC_320k	321,821	1,931,828
rajat31	4,690,002	20,316,253	rajat21	411,676	1,876,011
CoupCons3D	416,800	17,277,420	crashbasis	160,000	1,750,416
Freescall	3,428,755	17,052,626	majorbasis	160,000	1,750,416
circuit5M_dc	3,523,317	14,865,409	venkat01	62,424	1,717,792
memchip	2,707,524	13,343,948	venkat50	62,424	1,717,777
atmosmodm	1,489,752	10,319,760	venkat25	62,424	1,717,763
atmosmodl	1,489,752	10,319,760	ASIC_680ks	682,712	1,693,767
atmosmodj	1,270,432	8,814,880	ASIC_320ks	321,671	1,316,085
atmosmodd	1,270,432	8,814,880	language	399,130	1,216,334
torso1	116,158	8,516,500	twotone	120,750	1,206,265
PR02R	161,070	8,185,136	matrix_9	103,430	1,205,518
cage13	445,315	7,479,343	torso2	115,967	1,033,473
ohne2	181,343	6,869,939	transient	178,866	961,368
pre2	659,033	5,834,044	ASIC_100k	99,340	940,621
Hamrle3	1,447,360	5,514,242	matrix-new_3	125,329	893,984
Chebyshev4	68,121	5,377,761	m133-b3	200,200	800,800
largebasis	440,020	5,240,084	shar_te2-b3	200,200	800,800
tmt_unsym	917,825	4,584,801	trans4	116,835	749,800
torso3	259,156	4,429,042	Baumann	112,211	748,331
xenon2	157,464	3,866,688	rajat25	87,190	606,489
laminar_duct3D	67,173	3,788,857	rajat28	87,190	606,489
rajat29	643,994	3,760,246	rajat20	86,916	604,299
FEM_3D_thermal2	147,900	3,489,300	LeGresley_87936	87,936	593,276
stomach	213,360	3,021,648	ASIC_100ks	99,190	578,890
thermomech_dK	204,316	2,846,228	3D_51448_3D	51,448	537,038
ASIC_680k	682,862	2,638,997	ibm_matrix_2	51,448	537,038
poisson3Db	85,623	2,374,949	hcircuit	105,676	513,072
barrier2-10	115,625	2,158,759	lung2	109,460	492,564
barrier2-11	115,625	2,158,759	2D_54019_highK	54,019	486,129
barrier2-9	115,625	2,158,759	rajat17	94,294	479,246
barrier2-12	115,625	2,158,759	rajat18	94,294	479,151
mc2depi	525,825	2,100,225	mark3jac140sc	64,089	376,395
para-8	155,924	2,094,873	shyy161	76,480	329,762
para-9	155,924	2,094,873	mark3jac120sc	54,929	322,483
para-5	155,924	2,094,873	circuit_4	80,209	307,604
para-7	155,924	2,094,873	bayer01	57,735	275,094

Table 4.4: Number of CG Iterations, best result for each matrix is shown in bold

Matrix name	UP	HP	GP	Matrix name	UP	HP	GP
ML_Geer	355	492	408	para-10	F	F	3743
RM07R	1966	2995	506	para-6	F	F	3756
ML_Laplace	760	952	971	cage12	16	18	17
Transport	2214	1666	1520	ASIC_320k	1248	856	677
rajat31	521	606	239	rajat21	3358	F	3978
CoupCons3D	1285	479	152	crashbasis	263	182	163
Freescape1	F	4083	480	majorbasis	239	162	154
circuit5M_dc	68	38	7	venkat01	179	95	85
memchip	298	380	53	venkat50	F	3245	3154
atmosmodm	44	23	42	venkat25	F	2491	2250
atmosmodl	38	70	103	ASIC_680ks	6	5	7
atmosmodj	167	148	138	ASIC_320ks	66	3	9
atmosmodd	171	155	149	language	759	438	146
torso1	1559	854	1050	twotone	2643	1092	742
PR02R	F	3505	2194	matrix_9	4549	4291	2089
cage13	14	17	16	torso2	42	20	19
ohne2	F	2750	771	transient	2930	2044	F
pre2	2343	264	524	ASIC_100k	1057	138	169
Hamrle3	3493	2058	1583	matrix-new_3	600	151	222
Chebyshev4	F	F	173	m133-b3	22	23	20
largebasis	1379	554	924	shar_te2-b3	22	23	20
tmt_unsym	495	880	770	trans4	F	F	2011
torso3	162	75	27	Baumann	F	F	2986
xenon2	25	19	21	rajat25	3054	2414	918
laminar_duct3D	247	403	323	rajat28	2636	1280	842
rajat29	S	14	28	rajat20	3186	1912	893
FEM_3D_thermal2	86	73	47	LeGresley_87936	F	935	137
stomach	85	37	15	ASIC_100ks	203	51	21
thermomech_dK	26	7	7	ibm_matrix_2	F	2055	3527
ASIC_680k	23	8	5	3D_51448_3D	F	1926	3884
poisson3Db	F	1880	1457	hcircuit	F	F	2686
barrier2-10	F	F	4049	lung2	34	8	17
barrier2-11	F	F	4270	2D_54019_highK	F	275	60
barrier2-9	F	F	3803	rajat17	1695	584	386
barrier2-12	F	F	4043	rajat18	1123	406	393
mc2depi	262	487	409	mark3jac140sc	F	F	1716
para-8	F	3167	1821	shyy161	108	116	115
para-9	F	F	2826	mark3jac120sc	F	F	1709
para-5	F	F	3972	circuit_4	F	F	2400
para-7	F	F	3504	bayer01	648	25	27

As shown in Table 4.4, out of 80 of the test matrices, the GP method can succeed to converge on 79 instances, whereas HP and UP methods can succeed to converge on 63 and 52 instances, respectively. Out of 80 instances, the proposed GP method performs best in terms of the required number of iterations in 54 instances, whereas HP and GP perform best only in 17 and 10 instances, respectively. In terms of reaching the desired backward error in 5,000 iterations, the proposed GP method fails in only one test instances, whereas UP and HP methods fail to converge on 27 and 17 test instances, respectively.

In Table 4.5, out of 80 instances, GP performs best parallel solution time in 51 instances, whereas HP and UP methods perform best in 18 and 11 instances, respectively. Comparison of Tables 4.4 and 4.5 indicates that decrease on the number of CG iterations are reflected as decrease on the parallel CG times.

4.3.5 Preprocessing Overhead

In this subsection we compare the preprocessing overhead of the partitioning methods and the effect of preprocessing overhead on the total execution time.

Table 4.6 shows preprocessing time for each partitioning method. These times also include time of scaling the matrix, row-block partitioning, creating the row blocks and scattering these row blocks among processors. In Table 4.6, as expected, UP performs the best preprocessing time for all cases because UP is very straightforward and it incurs negligible cost to preprocessing overhead. As seen in the table, GP method incurs less preprocessing time than HP method on 78 out of 80 the matrices except `pre2` and `cirucit_4`.

Table 4.7 shows the total execution time for each partitioning method. The total execution time includes preprocessing, factorization and parallel solution with CG accelerated Block Cimmino algorithm times. As seen in the table, GP achieves the fastest total execution time in 50 instances, whereas HP and UP achieves the fastest execution time in only 7 and 23 instances, respectively.

Table 4.5: Parallel CG time, best result for each matrix is shown in bold

Matrix name	UP	HP	GP	Matrix name	UP	HP	GP
ML_Geer	293.0	602.9	999.9	para-10	F	F	308.4
RM07R	1140.0	1228.3	246.1	para-6	F	F	332.8
ML_Laplace	162.0	314.0	314.6	cage12	1.3	1.7	1.7
Transport	1400.0	1709.0	1416.5	ASIC_320k	416.0	347.6	315.1
rajat31	515.0	645.8	248.6	rajat21	787.0	F	935.5
CoupCons3D	333.0	114.4	39.1	crashbasis	18.7	10.0	8.9
Freescall1	F	2996.2	337.6	majorbasis	13.0	7.9	8.7
circuit5M_dc	77.7	28.6	5.4	venkat01	4.2	1.8	1.7
memchip	262.0	209.0	30.9	venkat50	F	65.5	68.3
atmosmodm	27.8	12.8	23.2	venkat25	F	53.8	48.6
atmosmodl	21.2	38.2	60.1	ASIC_680ks	1.1	1.0	1.1
atmosmodj	69.4	67.6	65.9	ASIC_320ks	6.6	0.3	0.8
atmosmodd	80.7	73.5	73.4	language	104.0	53.9	16.5
torso1	211.0	114.3	188.9	twotone	145.0	39.1	30.6
PR02R	F	391.7	258.9	matrix_9	214.0	184.3	105.3
cage13	4.9	7.7	7.9	torso2	1.2	0.7	0.6
ohne2	F	383.7	123.3	transient	583.0	253.8	F
pre2	535.0	47.9	137.1	ASIC_100k	180.0	21.7	25.5
Hamrle3	2030.0	715.2	521.3	matrix-new_3	35.1	6.4	9.9
Chebyshev4	F	F	46.4	m133-b3	1.7	1.8	1.5
largebasis	247.0	67.7	111.1	shar_te2-b3	1.9	1.8	1.6
tmt_unsym	89.1	202.1	194.3	trans4	F	F	335.3
torso3	20.7	12.1	4.2	Baumann	F	F	103.8
xenon2	1.8	1.2	1.3	rajat25	169.0	154.0	51.2
laminar_duct3D	13.7	18.2	19.3	rajat28	149.0	82.0	43.8
rajat29	S	11.8	25.1	rajat20	198.0	124.3	45.2
FEM_3D_thermal2	5.1	5.2	3.6	LeGresley_87936	F	15.0	2.3
stomach	8.5	4.0	1.5	ASIC_100ks	5.1	1.2	0.5
thermomech_dK	2.4	0.5	0.5	ibm_matrix_2	F	26.9	57.9
ASIC_680k	17.0	7.9	4.0	3D_51448_3D	F	28.3	72.6
poisson3Db	F	82.7	70.7	hcircuit	F	F	38.4
barrier2-10	F	F	309.6	lung2	0.6	0.1	0.3
barrier2-11	F	F	323.0	2D_54019_highK	F	3.3	0.7
barrier2-9	F	F	278.1	rajat17	92.7	32.1	20.6
barrier2-12	F	F	280.3	rajat18	65.3	22.3	20.2
mc2depi	25.5	70.2	54.1	mark3jac140sc	F	F	24.3
para-8	F	279.6	149.8	shyy161	1.3	1.9	1.7
para-9	F	F	249.5	mark3jac120sc	F	F	19.0
para-5	F	F	374.1	circuit_4	F	F	48.4
para-7	F	F	321.2	bayer01	6.1	0.3	0.3

Table 4.6: Preprocessing times in seconds.

Matrix name	UP	HP	GP	Matrix name	UP	HP	GP
ML_Geer	69.9	462.2	194.2	para-10	2.0	14.1	6.7
RM07R	19.5	208.7	84.2	para-6	1.9	14.3	6.7
ML_Laplace	15.8	114.9	46.6	cage12	1.8	19	9.2
Transport	22.4	126.2	50.4	ASIC_320k	2.5	20.3	7.7
rajat31	34.7	85.6	47.4	rajat21	2.7	23.6	15
CoupCons3D	10.4	60.6	29.8	crashbasis	1.5	7.1	4.2
Freescall1	27.8	81.3	42.1	majorbasis	1.6	7.2	4.1
circuit5M_dc	26.3	72.6	40.8	venkat01	1.2	5.7	3.0
memchip	20.5	58.7	36.1	venkat50	1.2	5.6	3.0
atmosmodm	12.5	67.5	25.3	venkat25	1.2	5.6	3.0
atmosmodl	12.5	67.6	24.5	ASIC_680ks	3.5	9.0	7.5
atmosmodj	8.9	57.7	21.3	ASIC_320ks	2.1	6.8	5.9
atmosmodd	10.1	58.5	22.3	language	2.5	9.0	7.0
torso1	4.1	143.3	11.8	twotone	1.2	6.9	6.6
PR02R	4.9	31.0	14.3	matrix_9	1.2	8.1	4.3
cage13	7.4	71.8	33.4	torso2	0.9	4.4	1.8
ohne2	4.5	32.0	19.2	transient	1.2	6.2	3.8
pre2	6.2	59.3	72.7	ASIC_100k	1.0	6.5	3.9
Hamrle3	9.8	30.8	16.5	matrix-new_3	0.9	6.2	3.4
Chebyshev4	2.7	19.2	14.6	m133-b3	1.3	9.2	8.3
largebasis	4.7	12.3	8.3	shar_te2-b3	1.3	9.1	8.4
tmt_unsym	6.4	19.6	10.1	trans4	0.8	4.7	2.6
torso3	3.5	24.0	9.2	Baumann	0.7	5.9	2.1
xenon2	2.9	14.4	7.5	rajat25	0.7	5.2	4.1
laminar_duct3D	2.2	17.0	9.9	rajat28	0.7	5.1	3.9
rajat29	4.3	28.0	14	rajat20	0.6	5.1	4.1
FEM_3D_thermal2	2.6	15.5	6.7	LeGresley_87936	0.7	2.6	1.5
stomach	2.5	15.1	5.9	ASIC_100ks	0.7	3.9	3.1
thermomech_dK	3.1	11.4	5.7	ibm_matrix_2	0.5	4.0	2.2
ASIC_680k	4.3	35.4	17.5	3D_51448_3D	0.5	4.1	2.2
poisson3Db	2.2	16.3	9.6	hcircuit	0.6	2.2	1.8
barrier2-10	1.9	15.4	7.6	lung2	0.6	1.5	1.0
barrier2-11	1.9	15.5	7.5	2D_54019_highK	0.5	2.6	1.3
barrier2-9	1.9	15.3	7.6	rajat17	0.6	4.7	3.6
barrier2-12	2.0	15.5	7.7	rajat18	0.6	4.7	3.5
mc2depi	3.3	10.9	5.4	mark3jac140sc	0.5	3.3	2.4
para-8	2.0	14.0	6.8	shyy161	0.4	1.9	0.9
para-9	2.0	14.0	6.8	mark3jac120sc	0.4	3.0	2.1
para-5	2.0	14.0	6.6	circuit_4	0.4	2.4	2.7
para-7	1.9	14.1	6.7	bayer01	0.3	1.7	1.1

Table 4.7: Total execution times (including preprocessing) in seconds.

Matrix name	UP	HP	GP	Matrix name	UP	HP	GP
ML_Geer	377.7	1100.9	1252.1	para-10	F	F	316.9
RM07R	1168	1471.6	341.8	para-6	F	F	340.6
ML_Laplace	181.1	432.8	366.4	cage12	4.3	22.3	12.6
Transport	1429.3	1872.5	1487.6	ASIC_320k	457.6	499.2	399.7
rajat31	562.3	745.5	326.7	rajat21	803.4	F	963.8
CoupCons3D	346.4	179.7	72.9	crashbasis	21.2	18.0	14.0
Freescal1	F	3086.6	394.3	majorbasis	15.5	16.0	13.6
circuit5M_dc	115.4	110.6	55.4	venkat01	6.1	8.0	5.3
memchip	291.4	284.7	74.5	venkat50	F	71.8	71.9
atmosmodm	45.9	91.3	56.6	venkat25	F	60	52.2
atmosmodl	39.4	117.3	96.7	ASIC_680ks	6.7	13.3	11.0
atmosmodj	83.1	133.7	94.8	ASIC_320ks	9.7	8.4	8.2
atmosmodd	95.6	140.4	103.4	language	108.3	65.1	25.0
torso1	220.3	286.9	224.7	twotone	147.2	47.9	38.6
PR02R	F	425.9	275.1	matrix_9	215.9	193.3	110.6
cage13	19.4	101	71.1	torso2	2.5	5.6	3.0
ohne2	F	421.4	144.8	transient	587.1	264.8	F
pre2	543.3	111.7	215.9	ASIC_100k	214.6	64.8	56.7
Hamrle3	2045.3	751.4	543.0	matrix-new_3	36.8	14.0	14.3
Chebyshev4	F	F	98.7	m133-b3	3.8	12	10.7
largebasis	253.1	81.6	121.0	shar_te2-b3	4.2	11.7	11
tmt_unsym	97.7	224.8	207.1	trans4	F	F	360.5
torso3	25.4	38.3	15.1	Baumann	F	F	106.6
xenon2	5.7	16.7	9.9	rajat25	170.9	161.8	56.7
laminar_duct3D	16.6	36.0	30.2	rajat28	150.8	89.4	49.1
rajat29	S	1109.4	825.1	rajat20	199.8	131.8	50.8
FEM_3D_thermal2	8.8	21.8	11.3	LeGresley_87936	F	18.1	4.4
stomach	12.0	20.3	8.5	ASIC_100ks	6.4	5.8	4.2
thermomech_dK	6.7	12.8	7.1	ibm_matrix_2	F	31.7	61.2
ASIC_680k	331.9	719.8	295.9	3D_51448_3D	F	33.2	75.2
poisson3Db	F	100	81.0	hcircuit	F	F	40.6
barrier2-10	F	F	318.0	lung2	1.6	1.9	1.8
barrier2-11	F	F	331.7	2D_54019_highK	F	8.1	2.8
barrier2-9	F	F	286.8	rajat17	94.5	38.3	25.6
barrier2-12	F	F	289.0	rajat18	67.0	28.4	25.3
mc2depi	30.1	83.0	61.2	mark3jac140sc	F	F	27.2
para-8	F	294.7	158.0	shyy161	1.9	4.4	3.1
para-9	F	F	257.5	mark3jac120sc	F	F	21.6
para-5	F	F	381.8	circuit_4	F	F	51.5
para-7	F	F	329.3	bayer01	6.7	2.5	1.9

The performance of GP drops a bit compared to the UP in terms of the total time since for some matrices there is a trade-off between increasing the preprocessing overhead and reducing the parallel CG solution time. UP gives the best results on 23 test instances because the original permutation of those matrices have a good characteristic already and they do not need any permutation in order to improve the CG convergence rate. Overhead of intelligent partitioning methods increases their total execution time with respect to UP method.

4.3.6 Parallel Scalability

In this sub-section we measure parallel scaling performance of the CG accelerated block Cimmino algorithm by using the proposed GP method with respect to the HP and UP methods.

We have conducted parallel scalability tests on Marenostrum Supercomputer [99] which is a distributed memory system. Marenostrum consists of 48,896 cores and 3,056 distributed nodes, where each node contains two 8-core Intel Xeon E5-2670 processors running at 2.6 Ghz and 32 GB RAM. Nodes are interconnected with Infiniband Mellanox FDR10 network. We use all 16 cores of each node and increase the number of nodes upto 16. So, in order to observe the scalability performance we use 1, 4, 16, 64, and 256 MPI processes, where each MPI process is mapped to one core.

For the experiments, we use a small dataset with four sparse matrices from Table 4.3. The properties of the matrices are shown in Table 4.8. We report the sequential MUMPS solver times including analysis, factorization and solution steps in the last column of the table. In the experiments each matrix is divided into 256 row blocks.

Figure 4.10 shows the CG speedup curves of the coefficient matrix with the three partitioning strategies. Speed-up values are calculated according to the best sequential time which are shown in Table 4.9 for each matrix.

Table 4.8: Properties of test matrices for parallel scalability and the sequential solutions times using the MUMPS direct solver in seconds. Mem. means the direct solver ran out of memory due to the fill-in.

Matrix	Nnz	Row	Type	MUMPS
RM07R	37,464,962	381,689	computational fluid dynamics problem	Mem.
CoupCons3D	17,277,420	416,800	structural problem	75.07
ohne2	6,869,939	181,343	semiconductor device problem	206.54
Hamrle3	5,514,242	1,447,360	circuit simulation problem	395.00

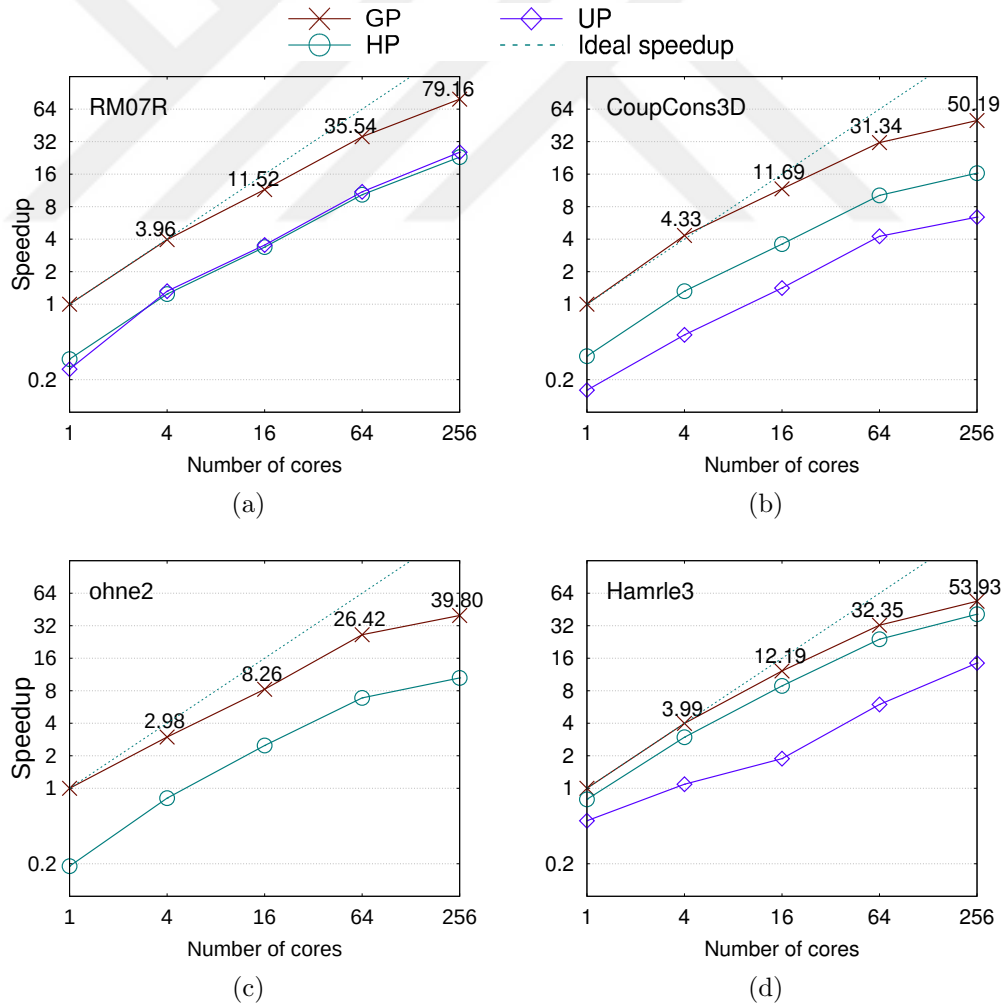


Figure 4.10: Speedup curves for CG time on the distributed memory architecture compared to the best sequential running time (given in Table 4.10)

Table 4.9: The number of CG iterations and parallel CG running times (in seconds).

	#Cores	# of CG iterations			CG time		
		UP	HP	GP	UP	HP	GP
RM07R	1	1990	2128	506	3760	3071	947
	4	1991	2128	505	719	761	239
	16	1991	2128	506	268	281	82
	64	1991	2129	506	86	92	27
	256	1991	2131	506	37	41	12
CoupCons3D	1	1278	480	152	737	361	119
	4	1277	479	153	228	90	27
	16	1278	479	152	84	33	10
	64	1278	479	152	28	12	4
	256	1279	480	152	19	7	2
ohne2	1	F	2480	693	F	1084	320
	4	F	2498	700	F	261	74
	16	F	2524	693	F	85	24
	64	F	2512	689	F	31	10
	256	F	2511	716	F	20	8
Hamrle3	1	3561	2055	1583	5670	3602	2858
	4	3553	2056	1583	2620	962	717
	16	3560	2057	1596	1520	322	234
	64	3569	2058	1596	479	119	88
	256	3562	2060	1576	198	70	53

UP: Uniform partitioning, HP: Hypergraph partitioning, GP: $\mathcal{G}_{\text{RIP}}$ partitioning. F: Failure due to maximum iteration reached. Times are given in seconds.

Figures 4.10a, 4.10b and 4.10d show the speedup curves of the CG time for all three partitioning methods. However, in Figure 4.10c, since the CG with UP partitioning method does not converge on `ohne2`, the figure reports the speed-up curves for only HP and GP methods.

As seen in Figure 4.10, the proposed algorithm improve the scalability performance of the CG accelerated block Cimmino algorithm via decreasing the number of CG iteration. The maximum speed-up of 79.16 is obtained by the `RM07R` matrix via GP method on 256 number of cores.

4.3.7 Parallel GP method for reducing the preprocessing overhead

In this subsection, we express the parallelization steps of the GP method in order to reduce preprocessing overhead. The proposed partitioning strategy is pretty suitable for distributed-memory parallelism. In order to further improve the preprocessing time, a parallel version of the proposed partitioning method is proposed and implemented. Parallelization scheme consists of 2 steps. In the first step we parallelize sparse matrix-matrix multiplication AA^T operation. In the second step, we utilize parallel graph partitioning tool via using partial results of AA^T on each processor.

In the first step, parallel sparse matrix-matrix multiplication $C = AA^T$ algorithm is implemented. The computation process of the resulting matrix C is divided into p row blocks where p is the number of MPI processes in the environment. Processors shares the rows of the resulting matrix C evenly in terms of number of rows. Each processor computes $C_i = A_i A^T$ submatrix, where $C_i = m_i \times n$, $A_i = m_i \times n$ and $n = \sum_i^p m_i$, then constructs a partial graph model for C_i .

In the second step, a parallel graph partitioning tool Parmetis [100] is used with the same parameters of METIS as defined in Section 4.3.1. Graph of C_i in each processor is used as an input for Parmetis. Finally, row blocks of A are

determined according to the partitioning result of Parmetis.

In ABCD Solver, a sequential matrix scaling algorithm is used. However, we exploit a parallel version of the same scaling algorithm given in [94] before partitioning the system via UP, HP and GP. For HP, since PaToH does not have a parallel version, we use a sequential hypergraph partitioning for parallel experiments.

Table 4.10: Dissection of parallel running times (in seconds) and the number of CG iterations.

	#Cores	Preprocessing time				# of CG iterations				CG time				Total time			
		UP	HP	GP	pGP	UP	HP	GP	pGP	UP	HP	GP	pGP	UP	HP	GP	pGP
RM07R	1	12	170	46	48	1990	2128	506	505	3760	3071	947	962	3818	3312	1078	1097
	4	6	165	38	25	1991	2128	505	838	719	761	239	294	736	953	308	336
	16	4	168	38	15	1991	2128	506	773	268	281	82	92	276	459	130	115
	64	4	168	38	17	1991	2129	506	629	86	92	27	24	94	266	70	47
	256	4	168	38	10	1991	2131	506	652	37	41	12	11	46	215	55	26
CoupCons3D	1	6	46	16	17	1278	480	152	156	737	361	119	124	758	434	164	169
	4	3	43	13	11	1277	479	153	157	228	90	27	28	236	141	48	46
	16	3	44	13	7	1278	479	152	162	84	33	10	10	89	81	26	20
	64	3	44	13	6	1278	479	152	156	28	12	4	4	33	59	20	12
	256	3	45	14	6	1279	480	152	162	19	7	2	2	26	56	20	11
ohne2	1	3	20	11	11	F	2480	693	455	F	1084	320	211	F	1121	346	236
	4	2	19	9	10	F	2498	700	666	F	261	74	71	F	285	87	85
	16	1	19	9	8	F	2524	693	738	F	85	24	25	F	106	35	35
	64	1	19	9	5	F	2512	689	627	F	31	10	8	F	52	20	14
	256	1	19	9	4	F	2511	716	624	F	20	8	5	F	42	19	11
Hamrie3	1	6	17	9	10	3561	2055	1583	1720	5670	3602	2858	3077	5705	3675	2924	3145
	4	6	17	9	8	3553	2056	1583	1524	2620	962	717	679	2629	988	736	697
	16	5	17	9	7	3560	2057	1596	1496	1520	322	234	219	1528	342	247	230
	64	5	16	8	6	3569	2058	1596	1454	479	119	88	80	486	138	100	89
	256	5	16	8	7	3562	2060	1576	1458	198	70	53	49	212	94	69	64

UP: Uniform partitioning, HP: Hypergraph partitioning, GP: $\mathcal{G}_{RIP}$ partitioning, pGP: Parallel $\mathcal{G}_{RIP}$

partitioning. F: Failure due to maximum iteration reached. Times are given in seconds.

Table 4.10 shows dissection of parallel running times of the four methods. In the table, preprocess time includes parallel scaling the matrix, partitioning into row blocks, creating the partitions and distributing data time. Total time includes the times of preprocessing, analysis and factorization of the systems and parallel CG accelerated block Cimmino.

As seen in Table 4.10, the proposed parallel partitioning method (pGP) can reduce the preprocessing time by 79% (4.8 speedup) for RM07R on 256 cores. On average, by using pGP, we get 41% extra time improvement with respect to the

preprocessing time of GP on 256 cores.

As also seen in Table 4.10, the proposed partitioning algorithm improves the total execution time of the algorithm via reducing the preprocessing time. The pGP method achieves a speedup of 41.6 in terms of the total solution time, whereas GP, HP and UP methods respectively achieve 19.5, 5.3 and 23.7 speedup with respect to the best sequential algorithm for matrix RM07R using 256 cores. For the same matrix, pGP method can achieve 2.11 times better speedup compared to the GP method.

4.4 Conclusion

In this chapter, we introduced a new partitioning scheme in order to reduce the number of iterations in the block Cimmino algorithm. In the block Cimmino algorithm, the coefficient matrix is divided into number of row blocks. In each iteration of the block Cimmino, minimum 2-norm solution of the systems in which the coefficient matrices are these row blocks. In order to solve these system, one can use ParBaMiN algorithm, which is proposed in Chapter 3, if the row blocks are in the column overlapping block diagonal form. The proposed partitioning method increases the numerical orthogonality between row blocks of input matrix. Additionally proposed method takes less preprocessing time than the state-of-the-art hypergraph partitioning method. In order to improve the parallelism capability, scalable distributed parallel partitioning method was also introduced. Conducted experiments on large set of real matrices confirmed the improvements on the convergence of block Cimmino and preprocessing time for partitioning.

Chapter 5

BCDcols: A Parallel Solution Scheme Based on Block Cimmino Algorithm for Matrices with Dense Columns

In this chapter, we study linear system of equations of the form

$$\mathcal{A}x = b, \tag{5.1}$$

where $\mathcal{A}$ is an $n \times n$ sparse nonsingular matrix, whereas x and b are column vectors of size n . The coefficient matrix A has dense columns. We propose a new parallel solution scheme based on block Cimmino algorithm (see Chapter 4) to solve the system in Equation 5.1.

As stated in the previous chapter, convergence rate of the block Cimmino algorithm depends on the orthogonality between row blocks. In Section 4.2, we proposed and introduced a new partitioning method which focuses on improving numerical orthogonality by using the row-inner-product graph $\mathcal{G}_{\text{RIP}}$ model. The objective of minimizing cutsize of graph partitioning on $\mathcal{G}_{\text{RIP}}$ corresponds to minimizing sum of row inner-products between different row blocks. In Section 4.3,

we showed that row-block partitioning induced by $\mathcal{G}_{\text{RIP}}$ partitioning significantly reduces the required number of iterations for the block Cimmino algorithm. In this chapter, we propose a new solution method in order to reduce the number of iterations further for matrices with dense columns.

5.1 Motivation and Problem Statement

Recall the fact that orthogonality between row blocks determine the convergence rate of the block Cimmino algorithm. If a column of a matrix contains at least two nonzero elements which are located in distinct row blocks, this column may disturb the orthogonality among those row blocks. Therefore, dense columns in the matrix have much more potential to deteriorate the orthogonality since they are more possibly lie on different row blocks.

In order to illustrate the effect of dense columns on the orthogonality of row blocks, we consider `init_adder1_6` matrix from the SuiteSparse Sparse Matrix Collection. `init_adder1_6` is a sparse nonsymmetric nonsingular circuit simulation problem of size $1,813 \times 1,813$ with 11,153 nonzeros. Figure 5.1 shows nonzero pattern of this matrix. This matrix has four dense columns whose number of nonzero elements are more than 5% of the column size.

The eigenvalue spectrum of H matrix (see Equation (4.5)) of the block Cimmino algorithm for the original system and the system without dense columns are shown in Figure 5.2, where the matrices are partitioned into 8 row blocks uniformly. As seen in Figure 5.2, dropping dense columns from the original matrix makes better clustering around 1 on the eigenvalue spectrum of H matrix. Moreover, the condition number of H of the original system is reduced from 9.2×10^7 to 4.5×10^6 after dropping dense columns.

Therefore we can exploit the improvement in the orthogonality by proposing a new solution method which disengages the dense columns from the solution of block Cimmino algorithm. Instead, the dense columns are solved separately.

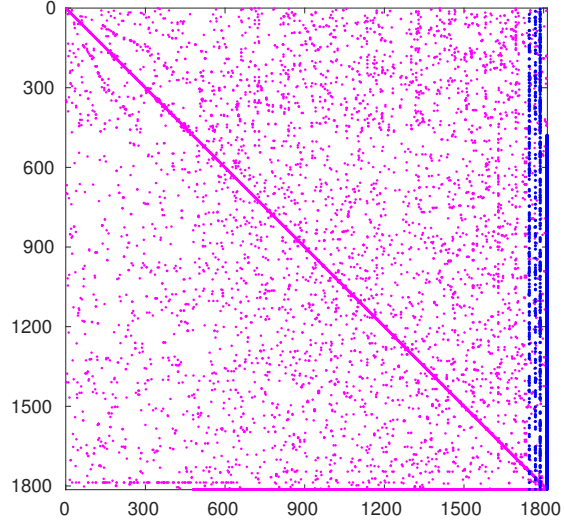
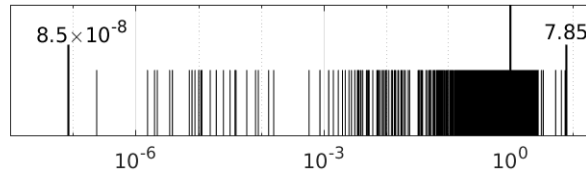
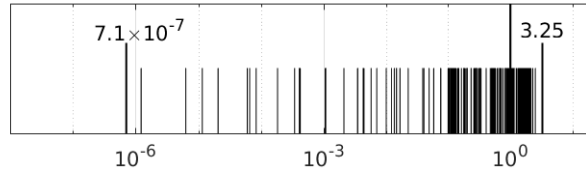


Figure 5.1: Sparsity pattern of `init_adder1_6`, dense columns are in blue color.



(a) Original matrix



(b) After dense columns removed

Figure 5.2: Eigenvalue spectrum of H matrix (with the smallest and largest eigenvalues)

In the next section, we introduce BCDcols for solving linear systems with dense columns.

5.2 BCDcols

In this section, we propose a new solution scheme, BCDcols, which combines the block Cimmino method and a direct method for the solution of a given linear system of equations. In BCDcols, we factorize the matrix into its block LU factors and form the resulting Schur complement [101, 102].

In the Schur complement approach, the matrix $\mathcal{A}$ in Equation 5.1 is partitioned into 2×2 matrix blocks

$$\mathcal{A} = \begin{bmatrix} A & B \\ C^T & D \end{bmatrix}, \quad (5.2)$$

where A is $n_A \times n_A$ and D is $s \times s$ square matrix, whereas B and C are $n_A \times s$ rectangular matrices. Since $\mathcal{A}$ is an $n \times n$ matrix, we have

$$n = n_A + s.$$

Vectors x and b are also partitioned conformably with $\mathcal{A}$. Therefore the linear system $\mathcal{A}x = b$ can be represented as

$$\begin{bmatrix} A & B \\ C^T & D \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}, \quad (5.3)$$

where u and v are column vectors of size n_A and s , respectively.

If A and D are not singular matrices, the block LU factorization of the coefficient matrix $\mathcal{A}$ can be obtained by

$$\begin{bmatrix} A & B \\ C^T & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ C^T A^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ 0 & S \end{bmatrix}, \quad (5.4)$$

where $S = D - C^T A^{-1} B$ is called Schur complements of $\mathcal{A}$ with respect to A and S is an $s \times s$ square matrix. Therefore the block factorization of the linear system

is

$$\begin{bmatrix} I & 0 \\ C^T A^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ 0 & S \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}. \quad (5.5)$$

The solution process can be divided into two stages. Let

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & S \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix}, \quad (5.6)$$

then the first stage is to solve the system

$$\begin{bmatrix} I & 0 \\ C^T A^{-1} & I \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}. \quad (5.7)$$

The solution of (5.7) is obtained by

$$\begin{aligned} a &= u \\ C^T A^{-1} a + b &= v \\ b &= v - C^T A^{-1} a \\ b &= v - C^T A^{-1} u. \end{aligned} \quad (5.8)$$

In the second stage, by using a and b sub-vectors, the solution of the following system is obtained

$$\begin{bmatrix} A & B \\ 0 & S \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}. \quad (5.9)$$

We substitute for a and b with the solution of the first stage obtained in Equation 5.8:

$$\begin{bmatrix} A & B \\ 0 & D - C^T A^{-1} B \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v - C^T A^{-1} u \end{bmatrix}. \quad (5.10)$$

Then, the solution of Equation (5.10) is obtained by

$$\begin{aligned} Ay + Bz &= u \\ (D - C^T A^{-1} B)z &= v - C^T A^{-1} u. \end{aligned} \quad (5.11)$$

As a result, in order to obtain solution subvectors y and z , one needs to solve

$$\begin{aligned} y &= A^{-1}(u - Bz) \\ z &= (D - C^T A^{-1} B)^{-1}(v - C^T A^{-1} u). \end{aligned} \quad (5.12)$$

In the next subsection, we will explain the solution steps of the proposed scheme by using Equation (5.12).

5.2.1 Solution Process

In BCDcols, we expect that matrix A is quite large and sparse, matrix B is dense and tall-and-skinny whereas matrix D is very small and dense. Therefore, one intelligent approach is to use an iterative hybrid method in order to solve the systems with the coefficient matrix A and use a direct method to solve the system with D . Figure 5.3 shows an illustration of the matrix partitioning in BCDcols.

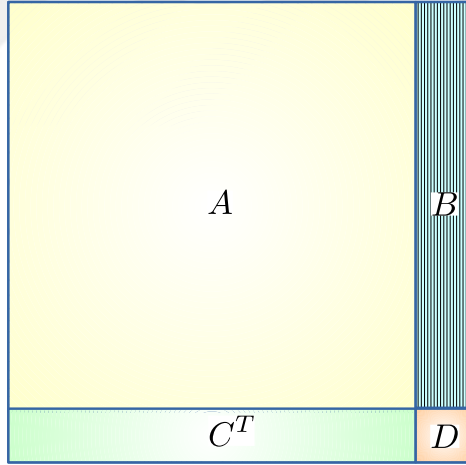


Figure 5.3: Matrix partitioning of $\mathcal{A}$ for BCDcols

According to (5.12), there are 4 systems to be solved. Systems $A^{-1}B$ and $A^{-1}u$ can be solved simultaneously since the coefficient matrix is the same and both B and u are known beforehand. Therefore, there are actually 3 systems to be solved in BCDcols. We explain the solution details of BCDcols in 3 steps.

In the first step, we use stabilized Block CG accelerated Block Cimmino algorithm to solve the large sparse systems with multiple right-hand-side vectors of the form

$$[F \ g] = A^{-1}[B \ u], \quad (5.13)$$

where F and g are the solutions of the system.

In the second step, by using the solution of Equation (5.13), we obtain z vector by the following equation

$$\begin{aligned} z &= S^{-1}(v - C^T A^{-1}u), \\ z &= (D - C^T A^{-1}B)^{-1}(v - CA^{-1}u), \end{aligned} \tag{5.14}$$

where S is the Schur complement of $\mathcal{A}$. We substitute for $A^{-1}B$ and $A^{-1}u$ respectively with F and g . The new equation can be rewritten as

$$z = (D - C^T F)^{-1}(v - Cg), \tag{5.15}$$

where computation of $(D - C^T F)$ and $(v - Cg)$ are very straightforward. In other words, $(D - C^T F)$ is obtained by matrix-matrix multiplication accompanied with matrix subtraction and $(v - C^T g)$ is obtained by matrix-vector multiplication accompanied with vector subtraction.

Note that in BCDcols, S is formed explicitly and its size is assumed to be quite small. Therefore, we use a dense direct solver for the system where the coefficient matrix is S . Then, the solution of this dense small system

$$\begin{aligned} z &= (D - C^T F)^{-1}(v - C^T g) \\ z &= S^{-1}(v - C^T g) \end{aligned} \tag{5.16}$$

is obtained by a direct solver in BCDcols.

In the third step, we solve the large sparse system

$$y = A^{-1}(u - Bz) \tag{5.17}$$

by using z vector obtained in Equation 5.16. In this step, we use classical CG accelerated block Cimmino algorithm in order to solve $A^{-1}(u - Bz)$ since there is one right-hand-side vector. Finally, we obtain y after this step and we construct the solution of the original system $\begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} x \end{bmatrix}$.

To sum up, obtaining the global solution in BCDcols consists of the following steps:

1. $[F \ g] = A^{-1}[B \ u]$: This sparse system with multiple right-hand-side vectors is solved by using an iterative hybrid method; stabilized Block CG accelerated Block Cimmino algorithm [82].
2. $z = S^{-1}(v - C^T g)$: Since the size of this system is very small, a dense direct linear system solver is preferred.
3. $y = A^{-1}(u - Bz)$: This sparse system with single right-hand-side vector is solved by an iterative hybrid method; classical CG accelerated Block Cimmino algorithm [103].

In the next subsection, we explain the parallelization of BCDCols and we describe the pseudo-code of BCDCols with its implementation details.

5.3 Parallelization and Implementation Details

For parallelization, we adopt a row-parallel algorithm, where each processor is responsible for a set of distinct rows. Figure 5.4 shows the row-block partitioning of sub-matrices A and B , where each sub-matrix is partitioned into 4 row blocks conformably. Figure 5.5 shows all data partitioning on the global system. The subvector u is also partitioned conformably. In BCDCols, since the computational works for the sub-matrices C^T and D are very small and take a negligible time, their computations are done sequentially.

Algorithm 5 shows the pseudo-code of the proposed algorithm. In the algorithm, lines 1-6 constitute the preprocessing stage. In this stage, we first permute $\mathcal{A}$ according to the output of *hsl_mc64* [104] subroutine in HSL Mathematical Software Library [105] in order to ensure that there is no zero value on the diagonal entries of $\mathcal{A}$. We use *hsl_mc64* with the option of maximizing the product of the diagonal entries of the permuted matrix. At line 2, we scale the rows and columns of $\mathcal{A}$ by using the algorithm in [94]. At line 3, we identify most s dense columns in $\mathcal{A}$. In the algorithm, the sparse matrix is stored in CSC format (see Section 2.2), therefore we can find the number of nonzero in a column very

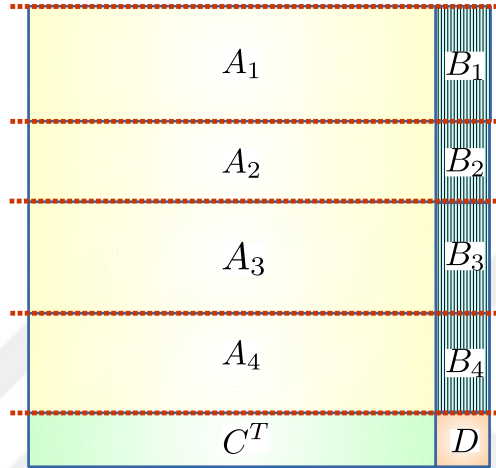


Figure 5.4: Row-block partitioning of submatrices A and B

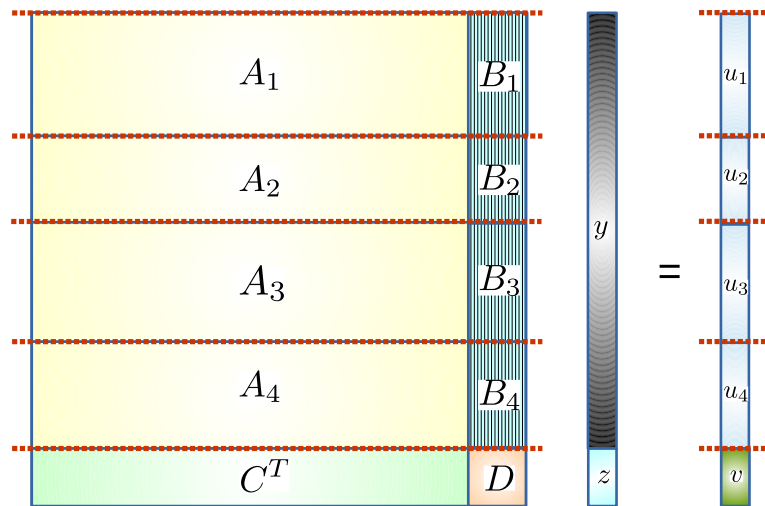


Figure 5.5: Row-block partitioning of the global system

easily in $O(1)$ time. At line 4, we construct submatrices A, B, C and D where dense columns are located in B submatrix, the respective rows according to the column indices of B are moved to C submatrix and the intersection of B and C submatrices constitutes D submatrix. At line 5, we partition A into row blocks according to the $\mathcal{G}_{\text{RIP}}$ partitioning on A . At line 6, we permute rows of B and u conformally with the row-block partition of A .

The solution stage starts at 7th line. At line 7, we solve the large sparse system which has multiple right-hand-side vectors including submatrix B and subvector u . In this step, each processor possesses A_i, u_i and B_i according to the row-block partition. In order to solve this system we use parallel stabilized block CG (BCG) accelerated block Cimmino algorithm [82, 103]. This algorithm is a parallel iterative hybrid method which uses a direct method within an iterative method and it can solve multiple right-hand-side vectors simultaneously. This step is the most time consuming step in our algorithm. The master processor gathers the F matrices from the other processors and does some basic matrix operations sequentially in lines 8-10. At line 11, the master processor solves very small dense system (Schur Complement of A) by using a direct solver. In our experiments, the size of this system is not larger than 5×5 . At line 12, the master processor broadcasts the solution sub-vector z to other processors. At line 13, all processors construct the right-hand-side vector for the system at 14th line through matrix and vector operations. This step is perfectly parallel which does not need any communication among processors. In the final step, we obtain subvector y by using parallel classical CG accelerated block Cimmino algorithm (CG). CG is implemented for efficient solution of systems with single right-hand-side vector. CG is usually second most time consuming step after BCG in the proposed algorithm. A sample MATLAB code for BCDcols is included in Appendix A.2.

For parallel implementation of the proposed method we use C++ programming language with MPI Library [58]. We use the algorithms implemented in open-source package ABCD Solver [87], which are the implementations of parallel matrix scaling [94], obtaining row projections with MUMPS [27] and parallel stabilized block CG accelerated block Cimmino algorithm [103]. Apart from

ALGORITHM 5: BCDcols

Input: $\mathcal{A}, b$

Output: y, z

- 1: Apply `hsl_mc64` on $\mathcal{A}$
 - 2: Matrix scaling on $\mathcal{A}$
 - 3: Select s dense columns in $\mathcal{A}$
 - 4: Construct submatrices A, B, C, D and subvectors u, v
 - 5: Row block partitioning on A via $\mathcal{G}_{\text{RIP}}$ partitioning
 - 6: Permute rows of B and u conformably
 - 7: Solve $AF = [u \ B]$ // Parallel BCG acc. block Cimmino
 - 8: $G = CF$ // Sequential
 - 9: $S = D - G(:, 2 : \text{end})$ // Sequential
 - 10: $q = v - G(:, 1)$ // Sequential
 - 11: Solve $Sz = q$ // Direct solution for z
 - 12: Broadcast z
 - 13: $j = u - Bz$ // Parallel
 - 14: Solve $Ay = j$ // Parallel CG acc. block Cimmino
-

these, we implement parallel classical CG accelerated block Cimmino algorithm (see Algorithm 4 in Chapter 4).

We use the same procedures for construction of $\mathcal{G}_{\text{RIP}}$ and partitioning $\mathcal{G}_{\text{RIP}}$ in Section 4.2. We use multilevel graph partitioning tool METIS [91] for partitioning $\mathcal{G}_{\text{RIP}}$. In the partitioning step, for each remaining dense columns in A which contains more than $\sqrt{n}$ nonzeros, we perform the same sparsening technique for these dense columns in A . The edge costs of $\mathcal{G}_{\text{RIP}}$ are determined with the same scaling procedure which transforms the floating point edge costs in $\mathcal{G}_{\text{RIP}}$ to integer edge weights required by METIS.

For the direct solution for z at line 11 of Algorithm 5, we use double precision general dense linear system solver `dgesv` subroutine in Intel Math Kernel Library LAPACK package [93]. At line 12, we use `mpi::broadcast` function in Boost MPI library [106] in order to broadcast the vector from master to the other processors.

We adopt a straightforward method for selecting dense columns in BCDcols. The method selects the most s dense columns in descending order in terms of number of nonzeros on the columns.

5.4 Numerical Experiments

In this subsection we conduct extensive numerical experiments on a parallel computer in order to study the performance of BCDcols. As a baseline for comparison, we use classical CG accelerated block Cimmino algorithm. For a fair comparison, we also use $\mathcal{G}_{\text{RIP}}$ partitioning method in order to determine row blocks and the input matrix is permuted by *hsl_mc64* in the baseline algorithm as in BCDcols.

5.4.1 Dataset

In experiments, we use real square matrices of size $n > 10,000$ from SuiteSparse Matrix Collection [60]. We selected matrices which have at least one dense column that has more number of nonzeros than 5% of n . We discard the matrices for which the baseline or BCDcols cannot reach the solution before 10,000 iterations. Table 5.1 shows the properties of 20 test matrices which satisfy these criteria. In the dataset, there are 9 different category of problems. These categories are 2D/3D problem, chemical process simulation problem, circuit simulation problem, subsequent circuit simulation problem, circuit simulation problem sequence, computer graphics/vision problem, optimization problem, subsequent optimization problem and structural problem. We use random right-hand-side vectors which are obtained by the default procedure that is implemented in ABCD Solver.

In BCDcols, we partition A into 256 row blocks by $\mathcal{G}_{\text{RIP}}$ partitioning method which is proposed in Section 4.2. We partition B and u conformably with the row-block partitioning on A .

5.4.2 Experimental Framework

Experiments are performed on a 64-core shared memory computer which has 4 AMD Opteron 6376 processors each of which has 16 cores running at 2.3GHz. We use MPI implementation of OpenMPI v1.10.2. We use gcc v4.7.2 compiler with

Table 5.1: Properties of test matrices.

Matrix name	n	nnz	category
ASIC_100k	99,340	940,621	circuit simulation problem
blockqp1	60,012	340,022	optimization problem
boyd2	466,316	890,091	optimization problem
bundle1	10,581	390,696	computer graphics/vision problem
c-58	37,595	295,073	optimization problem
c-62ghs	41,731	300,535	subsequent optimization problem
c-71	76,638	468,079	optimization problem
crankseg_1	52,804	5,333,507	structural problem
crankseg_2	63,838	7,106,348	structural problem
dc1	116,835	766,396	circuit simulation problem sequence
dc2	116,835	766,396	subsequent circuit simulation problem
dc3	116,835	766,396	subsequent circuit simulation problem
FullChip	2,987,012	26,621,983	circuit simulation problem
rajat30	643,994	6,175,244	circuit simulation problem
shermanACb	18,510	145,149	2D/3D problem
trans4	116,835	749,800	circuit simulation problem sequence
trans5	116,835	749,800	subsequent circuit simulation problem
Zd_Jac2	22,835	1,642,407	chemical process simulation problem
Zd_Jac3	22,835	1,915,726	chemical process simulation problem
Zd_Jac6	22,835	1,711,557	chemical process simulation problem

Intel® Math Kernel Library v11.1. For the iterative hybrid algorithms in baseline method and BCDcols, we use the same stopping criterion in Equation (4.23), with $\omega < 10^{-12}$.

As stated in Section 4.2.3, METIS graph partitioning tool is used in $\mathcal{G}_{\text{RIP}}$ partitioning. Since METIS uses randomized algorithms, we report the geometric mean of 5 runs with different partitioning seeds for both the baseline and the proposed algorithm.

5.4.3 Results and Discussion

We discuss the results of the experiments by comparing BCDcols against the baseline algorithm. Therefore we divide the dataset into two sets based on the robustness of the baseline algorithm for the test matrices. The matrices which cannot be solved by the baseline algorithm constitute the first set which contains 9 matrices. The matrices which can be solved by the baseline algorithm constitute the second set which contains 11 matrices. In both algorithms, we use set the maximum number of iterations as 10,000.

Tables 5.2 and 5.3 respectively show the required number of iterations for convergence and the parallel solution time for $s = 1, \dots, 5$, where s indicates the number of selected dense columns and it also indicates the size of Schur complement and column size of B . In the tables, **NC** means that an algorithm did not converge on the respective test instance before reaching 10,000 iterations.

In Table 5.2, we compare the number of iterations of BCDcols against the baseline algorithm. Since BCDcols has two major time consuming solution steps, which are BCG and CG, we indicate the number of iterations for BCG and CG individually. **BCG** column indicates the number of iterations for the first step which refers to the stabilized block CG accelerated block Cimmino algorithm. In BCG, we solve a system with multiple right-hand side vectors of size $s + 1$. The column **CG** indicates the number of iterations for the second step which refers to the classical CG accelerated block Cimmino algorithm. In CG, we solve a system

with single right-hand side.

Table 5.2: Number of iterations for block Cimmino algorithm and BCDcols. (NC: Not Converged)

Matrix	Cimmino	BCDcols									
		s=1		s=2		s=3		s=4		s=5	
		BCG	CG	BCG	CG	BCG	CG	BCG	CG	BCG	CG
c-58	NC	3630	3493	2830	4183	2536	4495	2436	4802	2290	5240
crankseg_1	NC	NC	-	NC	-	4657	8117	3526	7118	3383	7119
crankseg_2	NC	NC	-	NC	-	4566	8486	3738	6695	3388	6608
dc1	NC	NC	-	1885	2188	2467	2346	1406	1938	NC	-
dc2	NC	NC	-	913	1412	889	1748	910	1562	715	1543
dc3	NC	NC	-	1080	2275	724	1773	681	1840	701	2044
rajat30	NC	2333	7906	1340	5374	931	3780	572	2608	588	2633
trans4	NC	NC	-	98	173	102	229	55	112	65	199
trans5	NC	NC	-	109	194	101	202	78	172	87	191
ASIC_100k	393	118	180	34	45	33	40	32	45	32	43
blockqp1	183	52	74	47	93	NC	-	NC	-	NC	-
boyd2	895	1016	889	387	288	613	200	999	617	708	951
bundle1	1223	860	1266	619	1185	471	1131	416	1183	355	1238
c-62ghs	5371	2508	2262	2225	2178	2042	2247	1936	2861	1801	3417
c-71	2013	676	755	550	672	543	868	517	956	509	974
FullChip	2900	1165	1909	858	1174	59	169	49	124	36	62
shermanACb	9661	5257	7498	2857	2748	2608	6563	925	2040	685	4107
Zd_Jac2	430	381	396	237	273	319	384	186	403	361	633
Zd_Jac3	1325	158	493	333	219	743	1000	410	822	576	1670
Zd_Jac6	932	420	475	514	808	235	311	310	419	241	518

Table 5.2 shows the number of iterations for the baseline algorithm (Cimmino) and BCDcols for $s = 1, \dots, 5$. For example, for the matrix **ASIC_100k**, the baseline algorithm requires 393 iterations on average for the solution whereas BCDcols with $s = 1$ requires 118 BCG and 180 CG iterations. In this test instance we use only 1 dense column, therefore column size of B and C are 1 and S is a scalar (1×1 matrix). For $s = 2$ on **ASIC_100k**, by using BCDcols, the number of iterations is reduced to 34 and 45 for CG and BCG, respectively. As can be seen in the table, the proposed algorithm reduces the required number of iterations for almost all test instances. In the best case, BCDcols can reduce the number of iterations to 0.01% and 0.02% respectively for BCG and CG with respect to the baseline algorithm on **FullChip** problem with $s = 5$.

Table 5.3 indicates the parallel solution times on 16 cores for the baseline algorithm and BCDcols. The parallel solution time for BCDcols includes times of

Table 5.3: Parallel solution time in seconds for block Cimmino algorithm and BCDcols. (NC: Not Converged)

Matrix	Cimmino	BCDcols				
		$s=1$	$s=2$	$s=3$	$s=4$	$s=5$
c-58	NC	180.7	209.3	217.8	265.9	327.0
crankseg_1	NC	NC	NC	1081.3	976.3	1034.0
crankseg_2	NC	NC	NC	1397.9	1223.8	1338.2
dc1	NC	NC	637.9	892.2	606.0	NC
dc2	NC	NC	354.9	408.0	441.8	427.7
dc3	NC	NC	434.6	340.3	388.4	447.4
rajat30	NC	9051.0	2922.0	2368.8	1456.8	1948.5
trans4	NC	NC	33.9	43.1	25.8	39.6
trans5	NC	NC	36.3	42.3	37.0	48.5
ASIC_100k	60.7	58.3	18.5	19.6	21.1	23.1
blockqp1	14.7	13.4	9.2	NC	NC	NC
boyd2	374.5	679.3	281.7	750.7	1400.6	1164.9
bundle1	101.6	147.4	139.7	115.5	157.3	145.1
c-62ghs	105.0	134.2	168.5	186.4	225.7	270.2
c-71	85.5	68.1	80.9	96.2	114.2	138.0
FullChip	11171.3	16713.3	13494.0	1292.1	1154.3	802.5
shermanACb	252.8	391.6	167.9	300.4	75.2	110.8
Zd_Jac2	10.4	22.6	14.6	25.1	20.5	42.6
Zd_Jac3	34.2	16.6	18.2	68.9	45.4	97.9
Zd_Jac6	21.1	24.6	27.1	17.9	21.0	27.0

all solution steps which are described in Algorithm 5 at lines 7-14. As can be seen in Table 5.3 most of the time, we get the best parallel time improvement when $s = 2$. When s increases, the improvement over the baseline algorithm starts to decrease for almost all test instances except for **FullChip** which continues to improve until for $s = 5$. In the best case, BCDcols achieves approximately 14 times faster parallel solution time with respect to the baseline algorithm on **FullChip** with $s = 5$.

5.5 Conclusion

In this chapter, we proposed a new parallel algorithm, BCDcols, which is based on Schur complement technique. BCDcols separates the dense columns from the solution of block Cimmino method and solves the system via treating the dense columns privately. We observed that dense columns in the coefficient matrix deteriorate structural and numerical orthogonality among row blocks in block Cimmino algorithm. Experimental results shows that using a few columns for BCDcols significantly improves the convergence of block Cimmino method and hence reduces the parallel time for the solution.

Chapter 6

Conclusion

In this dissertation, we considered the parallel solution of large sparse linear systems based on row-block partitioning of the coefficient matrix. Our approaches aimed to reduce the parallel solution time of the linear systems with respect to the state-of-the-art baseline methods. We proposed three novel contributions; a new parallel direct method which finds the minimum 2-norm solution of the block diagonal column overlapping underdetermined systems, a graph theoretical row-block partitioning method for accelerating the block Cimmino algorithm and a new parallel hybrid method for further accelerating the block Cimmino for matrices with dense columns. For each of our approaches we confirmed the validity of the proposed algorithms and methods by conducting extensive numerical experiments.

In Chapter 3, a new parallel algorithm was proposed and implemented for computing the minimum norm solution of sparse block diagonal column overlapped underdetermined systems. We call the proposed algorithm ParBaMiN: Parallel Balance Scheme Algorithm for Minimum Norm Solution of Underdetermined Systems. ParBaMiN exploits the special structure of the coefficient matrix by handling the block diagonals independently and concurrently. Parallel scalability of ParBaMiN was shown on two different parallel architectures, one is a shared memory (multicore) architecture and the other is a distributed memory (cluster)

architecture. ParBaMiN achieves up to 54.1 speedup on 64 cores. Scalability performance of ParBaMiN degrades with increasing column overlap size as expected. Experimental results on all datasets show the validity of ParBaMiN on shared memory and distributed memory architectures. Furthermore, numerical stability of ParBaMiN was studied. ParBaMiN can be also considered as a scalable extension of any multithreaded general sparse QR factorization algorithm to distributed memory architectures for computing minimum 2-norm solution of underdetermined linear least squares problems.

In Chapter 4, we proposed a novel graph theoretical partitioning scheme in order to accelerate the convergence of the block Cimmino iterative algorithm. In each iteration of the algorithm, the minimum 2-norm solution of the row blocks are obtained. In order to solve these systems, one can use ParBaMiN, if the row blocks are in the column overlapping block diagonal form. In order to further improve the parallelism capability, a scalable distributed parallel partitioning method was also introduced and implemented. Conducted experiments on large set of real matrices confirmed the improvements on the convergence of block Cimmino.

In Chapter 5, we proposed a new parallel hybrid method, BCDcols, which is based on block Cimmino algorithm. Since orthogonality among row blocks is affected adversely by dense columns in the coefficient matrix, BCDcols separates the dense columns from the solution of block Cimmino method and solves the global system via treating the dense columns separately. BCDcols adopts well-known Schur complement approach which obtains the solution through partitioning the coefficient matrix into 2×2 block matrices. Experimental results showed that using a few dense columns for BCDcols significantly improves the convergence of block Cimmino method and hence reduces the parallel time for the solution. BCDcols achieved approximately 14 times faster parallel solution time with respect to the baseline algorithm.

Bibliography

- [1] M. Manguoglu, K. Takizawa, A. H. Sameh, and T. E. Tezduyar, “A parallel sparse algorithm targeting arterial fluid mechanics computations,” *Computational Mechanics*, vol. 48, no. 3, p. 377, 2011.
- [2] M. Manguoglu, K. Takizawa, A. H. Sameh, and T. E. Tezduyar, “Solution of linear systems in arterial fluid mechanics computations with boundary layer mesh refinement,” *Computational Mechanics*, vol. 46, no. 1, pp. 83–89, 2010.
- [3] S. E. Cohn, A. Da Silva, J. Guo, M. Sienkiewicz, and D. Lamich, “Assessing the effects of data selection with the dao physical-space statistical analysis system,” *Monthly Weather Review*, vol. 126, no. 11, pp. 2913–2926, 1998.
- [4] M. J. Miranda and P. L. Fackler, *Applied computational economics and finance*. MIT press, 2004.
- [5] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.
- [6] R. R. Vallabhajosyula, V. Chickarmane, and H. M. Sauro, “Conservation analysis of large biochemical networks,” *Bioinformatics*, vol. 22, no. 3, pp. 346–353, 2005.
- [7] J. Ye and Q. Li, “A two-stage linear discriminant analysis via qr-decomposition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 6, pp. 929–941, 2005.
- [8] M. J. Peyrovian and A. A. Sawchuk, “Image restoration by spline functions,” *Appl. Opt.*, vol. 17, pp. 660–666, Feb 1978.

- [9] H. Huang, J. M. Dennis, L. Wang, and P. Chen, “A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography,” *Procedia Computer Science*, vol. 18, pp. 581–590, 2013.
- [10] J. K. Ho and E. Loute, “A set of staircase linear programming test problems,” *Mathematical Programming*, vol. 20, no. 1, pp. 245–250, 1981.
- [11] R. Fourer, “Staircase matrices and systems,” *SIAM review*, vol. 26, no. 1, pp. 1–70, 1984.
- [12] J. K. Ho, “Optimal design of multi-stage structures: a nested decomposition approach,” *Computers & Structures*, vol. 5, no. 4, pp. 249–255, 1975.
- [13] R. Fourer, “Solving staircase linear programs by the simplex method, 1: Inversion,” *Mathematical Programming*, vol. 23, no. 1, pp. 274–313, 1982.
- [14] R. Fourer, “Solving staircase linear programs by the simplex method, 2: Pricing,” *Mathematical Programming*, vol. 25, no. 3, pp. 251–292, 1983.
- [15] C. R. Glassey and P. Benenson, “Quadratic programming analysis of energy in the united states economy. final report,” tech. rep., California Univ., Berkeley (USA), 1975.
- [16] G. H. Golub and C. F. Van Loan, *Matrix computations*, p. 530. John Hopkins University Press, 1983.
- [17] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.
- [18] T. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, “A survey of direct methods for sparse linear systems,” *Acta Numerica*, vol. 25, pp. 383–566, 2016.
- [19] I. Duff and B. Uçar, “Direct methods for sparse matrix solution,” *Scholarpedia*, vol. 8, no. 10, p. 9700, 2013.
- [20] T. N. Bui and C. Jones, “A heuristic for reducing fill-in in sparse matrix factorization,” tech. rep., Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States), 1993.

- [21] P. R. Amestoy, T. A. Davis, and I. S. Duff, “An approximate minimum degree ordering algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.
- [22] T. A. Davis, “A column pre-ordering strategy for the unsymmetric-pattern multifrontal method,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 165–195, 2004.
- [23] O. Kaya, E. Kayaaslan, B. Uçar, and I. S. Duff, *Fill-in reduction in sparse matrix factorizations using hypergraphs*. PhD thesis, INRIA, 2014.
- [24] X. S. Li, “An overview of superlu: Algorithms, implementation, and user interface,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 302–325, 2005.
- [25] T. A. Davis, “Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.
- [26] C. G. Petra, O. Schenk, M. Lubin, and K. Gärtner, “An augmented incomplete factorization approach for computing the schur complement in stochastic optimization,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C139–C162, 2014.
- [27] P. R. Amestoy, I. Duff, J. Koster, and J.-Y. L’Excellent, “A fully asynchronous multifrontal solver using distributed dynamic scheduling,” *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.
- [28] T. A. Davis, “Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 8, 2011.
- [29] P. R. Amestoy, I. S. Duff, and C. Puglisi, “Multifrontal qr factorization in a multiprocessor environment,” *Numerical linear algebra with applications*, vol. 3, no. 4, pp. 275–300, 1996.

- [30] C. Ashcraft and R. G. Grimes, “Spooles: An object-oriented sparse matrix library,” in *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1999.
- [31] A. Buttari, “Fine-grained multithreading for the multifrontal QR factorization of sparse matrices,” *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C323–C345, 2013.
- [32] P. E. Gill and W. Murray, “A numerically stable form of the simplex algorithm,” *Linear Algebra and Its Applications*, vol. 7, no. 2, pp. 99–138, 1973.
- [33] M. A. Saunders, “Large-scale linear programming using the cholesky factorization,” 1972.
- [34] M. Arioli, I. S. Duff, and P. P. de Rijk, “On the augmented system approach to sparse least-squares problems,” *Numerische Mathematik*, vol. 55, no. 6, pp. 667–684, 1989.
- [35] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, “PETSc users manual,” Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.
- [36] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “Cuspars library,” in *GPU Technology Conference*, 2010.
- [37] Y. Saad, “Sparskit: A basic tool kit for sparse matrix computations,” 1990.
- [38] B. Carpentieri, L. Giraud, and I. Duff, “Some sparse pattern selection strategies for robust frobenius norm minimization preconditioners in electromagnetism,” tech. rep., SCAN-0005096, 2000.
- [39] I. S. Duff, “Combining direct and iterative methods for the solution of large systems in different application areas,” *Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, Toulouse, France, Tech. Rep. TR/PA/04/128*, 2004.

- [40] C. Balsa, R. Guivarch, D. Ruiz, and M. Zenadi, “An hybrid approach for the parallelization of a block iterative algorithm,” in *VECPAR*, pp. 116–128, Springer, 2010.
- [41] L. Drummond, I. Duff, R. Guivarch, D. Ruiz, and M. Zenadi, “Partitioning strategies for the block cimmino algorithm,” *Journal of Engineering Mathematics*, vol. 93, no. 1, pp. 21–39, 2015.
- [42] C. L. Lawson and R. J. Hanson, *Solving least squares problems (Classics in Applied Mathematics)*. Society for Industrial Mathematics, 1987.
- [43] Å. Björck, *Numerical Methods for Least Squares Problems*. Philadelphia: SIAM, 1996.
- [44] M. S. Zhdanov, *Geophysical inverse theory and regularization problems*, vol. 36. Elsevier, 2002.
- [45] M. K. Sen and P. L. Stoffa, *Global optimization methods in geophysical inversion*. Cambridge University Press, 2013.
- [46] A. Bruckstein, D. Donoho, and M. Elad, “From sparse solutions of systems of equations to sparse modeling of signals and images,” *SIAM Review*, vol. 51, no. 1, pp. 34–81, 2009.
- [47] S. F. Cotter, B. D. Rao, K. Engan, and K. Kreutz-Delgado, “Sparse solutions to linear inverse problems with multiple measurement vectors,” *Signal Processing, IEEE Transactions on*, vol. 53, no. 7, pp. 2477–2488, 2005.
- [48] K. Matsuura and Y. Okabe, “Selective minimum-norm solution of the bi-magnetic inverse problem,” *Biomedical Engineering, IEEE Transactions on*, vol. 42, pp. 608–615, June 1995.
- [49] J.-Z. Wang, S. J. Williamson, and L. Kaufman, “Magnetic source images determined by a lead-field analysis: the unique minimum-norm least-squares estimation,” *Biomedical Engineering, IEEE Transactions on*, vol. 39, no. 7, pp. 665–675, 1992.

- [50] F. S. Torun, M. Manguoglu, and C. Aykanat, “Parallel minimum norm solution of sparse block diagonal column overlapped underdetermined systems,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, p. 31, 2017, Reprinted by permission.
- [51] T. A. Davis, “Suitesparse packages, released 4.2. 1,” 2013.
- [52] G. Golub, A. H. Sameh, and V. Sarin, “A parallel balance scheme for banded linear systems,” *Numerical Linear Algebra with Applications*, vol. 8, no. 5, pp. 285–299, 2001.
- [53] A. H. Sameh and V. Sarin, “Parallel algorithms for indefinite linear systems,” *Parallel Comput.*, vol. 28, no. 2, pp. 285–299, 2002.
- [54] T. E. Tezduyar and A. Sameh, “Parallel finite element computations in fluid mechanics,” *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 13-16, pp. 1872 – 1884, 2006. A Tribute to Thomas J.R. Hughes on the Occasion of his 60th Birthday.
- [55] J. W. Demmel and N. J. Higham, “Improved error bounds for underdetermined system solvers,” *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 1–14, 1993.
- [56] MATLAB, *version 8.6.0 (R2015b)*. Natick, Massachusetts: The MathWorks Inc., 2015.
- [57] T. A. Davis, “User’s guide for suitesparseqr, a multifrontal multithreaded sparse qr factorization package,” 2009.
- [58] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the mpi message passing interface standard,” *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.
- [59] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petit, *et al.*, *ScaLAPACK users’ guide*, vol. 4. SIAM, 1997.
- [60] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

- [61] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra, “Matrix market: a web resource for test matrix collections.,” in *Quality of Numerical Software*, pp. 125–137, 1996.
- [62] S.-P. Han, “Least-squares solution of linear inequalities.,” tech. rep., DTIC Document, 1980.
- [63] R. Bramley and B. Winnicka, “Solving linear inequalities in a least squares sense,” *SIAM Journal on Scientific Computing*, vol. 17, no. 1, pp. 275–286, 1996.
- [64] M. Ç. Pinar, “Newton’s method for linear inequality systems,” *European journal of operational research*, vol. 107, no. 3, pp. 710–719, 1998.
- [65] A. Dax, “A hybrid algorithm for solving linear inequalities in a least squares sense,” *Numerical Algorithms*, vol. 50, no. 2, pp. 97–114, 2008.
- [66] L. S. C. GCS Supercomputer, “Supermuc petascale system,” 2012.
- [67] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* ” O’Reilly Media, Inc.”, 2007.
- [68] G. Karypis and V. Kumar, “Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0,” 1995.
- [69] F. S. Torun, “Parbamin: Parallel balance scheme algorithm for minimum norm solution of underdetermined systems,” 2016. See <http://www.sukrutorun.com/>.
- [70] N. J. Higham, “Algorithm 694: A collection of test matrices in matlab,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 17, no. 3, pp. 289–305, 1991.
- [71] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002.
- [72] M. Benzi, “Preconditioning techniques for large linear systems: a survey,” *Journal of computational Physics*, vol. 182, no. 2, pp. 418–477, 2002.

- [73] Y. Saad, “Krylov subspace methods for solving large unsymmetric linear systems,” *Mathematics of computation*, vol. 37, no. 155, pp. 105–126, 1981.
- [74] O. Schenk and K. Gärtner, “Solving unsymmetric sparse systems of linear equations with pardiso,” *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.
- [75] H. A. Van der Vorst, “Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems,” *SIAM Journal on scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.
- [76] C. Kamath and A. Sameh, “A projection method for solving nonsymmetric linear systems on multiprocessors,” *Parallel Computing*, vol. 9, no. 3, pp. 291–312, 1989.
- [77] R. B. Bramley, *Row projection methods for linear systems*. PhD thesis.
- [78] S. Kaczmarz, “Angenäherte auflösung von systemen linearer gleichungen,” *Bulletin International de l’Academie Polonaise des Sciences et des Lettres*, vol. 35, pp. 355–357, 1937.
- [79] G. Cimmino and C. N. delle Ricerche, *Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari*. Istituto per le applicazioni del calcolo, 1938.
- [80] R. Bramley and A. Sameh, “Row projection methods for large nonsymmetric linear systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 168–193, 1992.
- [81] M. Arioli, I. Duff, J. Noailles, and D. Ruiz, “A block projection method for sparse matrices,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 47–70, 1992.
- [82] M. Arioli, I. Duff, D. Ruiz, and M. Sadkane, “Block lanczos techniques for accelerating the block cimmino method,” *SIAM Journal on Scientific Computing*, vol. 16, no. 6, pp. 1478–1511, 1995.
- [83] T. Elfving, “Block-iterative methods for consistent and inconsistent linear equations,” *Numerische Mathematik*, vol. 35, no. 1, pp. 1–12, 1980.

- [84] Y. Censor, “Parallel application of block-iterative methods in medical imaging and radiation therapy,” *Mathematical Programming*, vol. 42, no. 1, pp. 307–325, 1988.
- [85] T. Elfving and T. Nikazad, “Properties of a class of block-iterative methods,” *Inverse Problems*, vol. 25, no. 11, p. 115011, 2009.
- [86] I. Duff, R. Guivarch, D. Ruiz, and M. Zenadi, “The augmented block cimmino distributed method,” *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1248–A1269, 2015.
- [87] M. Zenadi, D. Ruiz, and R. Guivarch, “The augmented block cimmino distributed solver.” <http://abcd.enseeiht.fr/>, 2015. ABCD Solver v1.0-beta.
- [88] G. Golub and W. Kahan, “Calculating the singular values and pseudo-inverse of a matrix,” *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965.
- [89] A. Björck and G. H. Golub, “Numerical methods for computing angles between linear subspaces,” *Mathematics of computation*, vol. 27, no. 123, pp. 579–594, 1973.
- [90] S. A. Gershgorin, “Über die abgrenzung der eigenwerte einer matrix,” *Bulletin de l’Académie des Sciences de l’URSS. Classe des sciences mathématiques et na*, no. 6, pp. 749–754, 1931.
- [91] G. Karypis and V. Kumar, “Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.1.” <http://www.cs.umn.edu/~metis>, 2013.
- [92] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.
- [93] Intel, “Intel math kernel library.” software.intel.com/mkl, 2017. Intel MKL 2017.

- [94] P. R. Amestoy, I. Duff, D. Ruiz, and B. Uçar, “A parallel matrix scaling algorithm,” in *International Conference on High Performance Computing for Computational Science*, pp. 301–313, Springer, 2008.
- [95] M. Arioli, I. Duff, and D. Ruiz, “Stopping criteria for iterative solvers,” *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 138–144, 1992.
- [96] U. V. Catalyurek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.
- [97] Ü. V. Catalyürek and C. Aykanat, “PatoH: A multilevel hypergraph partitioning tool,” *Tech. Rep. BU-CE-9915*, 1999.
- [98] I. Duff, R. G. Grimes, and J. G. Lewis, “Users’ guide for the harwell-boeing sparse matrix collection (release i),” 1992.
- [99] B.-C. Marenstrum Supercomputer, “Marenstrum hpc,” 2016.
- [100] G. Karypis, K. Schloegel, and V. Kumar, “Parmetis: Parallel graph partitioning and sparse matrix ordering library,” *Version 4.0.3, Dept. of Computer Science, University of Minnesota*, 1997.
- [101] F. Zhang, *The Schur complement and its applications*, vol. 4. Springer Science & Business Media, 2006.
- [102] C. Paige and C. Van Loan, “A schur decomposition for hamiltonian matrices,” *Linear Algebra and its Applications*, vol. 41, pp. 11–32, 1981.
- [103] M. Zenadi, *Méthodes hybrides pour la résolution de grands systèmes linéaires creux sur calculateurs parallèles*. PhD thesis, École Doctorale Mathématiques, Informatique et Télécommunications (Toulouse); 142547247, 2013.
- [104] I. S. Duff and J. Koster, “On algorithms for permuting large entries to the diagonal of a sparse matrix,” *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 4, pp. 973–996, 2001.

- [105] A. HSL, “collection of fortran codes for large-scale scientific computation,”
See <http://www.hsl.rl.ac.uk>, 2007.
- [106] B. Dawes, D. Abrahams, *et al.*, “Boost library.” <http://www.boost.org/>,
2017.



Appendix A

Code

A.1 ParBaMiN

A MATLAB code for parallel minimum 2-norm solution of block diagonal column overlapping underdetermined system of linear equations is given below.

```
function x = ParBaMiN(A,f,blkptr,colptr,p)
    [Q,R,P,Q2] = factorize(p,A,blkptr,colptr);
    [x] = solve(p,f,blkptr,colptr,Q,R,P,Q2);
    % Relative residual
    rel_res = norm(A*x-f)/norm(f)
    xnorm = norm(x)
return;

% -----
% Routine for QR factorization
%
function [Q,R,P,Q2] = factorize(p,A,blkptr,colptr)
```

```

Q2 = cell(p,1);
Q = cell(p,1);
R = cell(p,1);
P = cell(p,1);

for i=1:p,
    r=[blkptr(i):blkptr(i+1)-1]'; n1=size(r,1);
    c=[colptr(i,1):colptr(i,2)]'; n2=size(c,1);
    Ei=A(r,c);
    [mi,ni]=size(Ei);
    opts.Q='Householder';
    opts.ordering='default';
    opts.permutation = 'matrix';
    opts.econ=0;
    [Q{i},R{i},P{i}] =spqr (Ei',opts);
    Q2ind=speye(ni,ni);
    Q2ind=Q2ind(:,mi+1:ni);
    Q2{i}=spqr_qmult(Q{i},Q2ind,1);
end
return;

% -----
% Routine for solving
%
function [x] = solve(p,f,blkptr,colptr,Q,R,P,Q2)

Mptr = cumsum([1; 1+colptr(1:p-1,2)-colptr(2:p,1)]);
Cptr = cumsum(colptr(1:p,2)-colptr(1:p,1)+1 - (blkptr(2:p+1) - blkptr(1:p)));
m = Mptr(p)-1;
n = size(f,1);
M=sparse(m,Cptr(end)); % Initialize reduced M,g
g=sparse (m,1);

```

```

pi = cell(p,1);

for i=1:p,
    r=[blkptr(i):blkptr(i+1)-1]'; n1=size(r,1);
    c=[colptr(i,1):colptr(i,2)]'; n2=size(c,1);

    if (i>1), p3=[Mptr(i-1):Mptr(i)-1]'; else p3=[]; end;
    n3=size(p3,1);
    if (i<p), p4=[Mptr(i):Mptr(i+1)-1]'; else p4=[]; end;
    n4=size(p4,1);

%       pi{i} = spqr_qmult (Q,(R*P')'\fi,1) ;
mi = size(R{i},1);
ni =size(Q{i}.H,1);
fi=f(r,:);
rr=(R{i}*P{i}')'\fi;

zz = zeros(ni,1);
zz(1:mi,:)=rr;
pi{i}=spqr_qmult (Q{i},zz,1) ;

if (i==1),
    cc = 1: Cptr(i);
    M(p4,cc) = Q2{i}(n2-n4+1:n2,:);
    g(p4,:)=g(p4,:)-pi{i}(n2-n4+1:n2,:);
elseif (i<p),
    cc = Cptr(i-1) + 1 : Cptr(i); % : Mptr(i);
    M(p4,cc)=Q2{i}(n2-n4+1:n2,:);
    M(p3,cc)=-Q2{i}(1:n3,:);
    g(p4)=g(p4)-pi{i}(n2-n4+1:n2);
    g(p3)=g(p3)+pi{i}(1:n3);
else
    cc = Cptr(i-1) + 1: Cptr(i);

```

```

        M(p3,cc)=-Q2{i}(1:n3,:);
        g(p3,:)=g(p3,:)+pi{i}(1:n3,:);
    end;
end;

% Solving reduced system
y = spqr_solve(M,g,struct('solution','min2norm'));
x=zeros(n,1); % Initialize solution

for i=1:p, % Compute x-blocks
    r=[blkptr(i):blkptr(i+1)-1]'; n1=size(r,1);
    c=[colptr(i,1):colptr(i,2)]'; n2=size(c,1);

    if (i==1), cc = 1: Cptr(i); % y-index
    elseif (i<p), cc = Cptr(i-1) + 1 : Cptr(i);
    else cc=Cptr(i-1)+1: size(M,2);
    end;
    x(c)=pi{i}+ Q2{i} * y(cc);
end;
return;

```

A.2 BCDcols

A sample MATLAB code is given for the parallel solution scheme based on block cimmino algorithm for matrices with dense columns.

```

function [ X ] = BCDcols( A , b, p )
{
    [m,n] = size(A);
    [rowp,colp] = hsl_mc64(A,5);
    A(rowp,colp) = A;
    b = b(rowp,:);

```

```

f=b;

for i=1:m
    zc(i) = nnz(A(:,i));
end

denseCols = find(zc> sqrt(m)*10);
sz_Cols = size(denseCols,2);

B = A(:,denseCols);
B(denseCols,:) = [];
C = A(denseCols,:);
C(:,denseCols) = [];
D = A(denseCols,denseCols);
A(:,denseCols) = [];
A(denseCols,:) = [];
v = f(denseCols,:);
f(denseCols,:) = [];
u = f;

% y = inv( D - ( C*inv(A)*B ) ) * ( v - ( C*inv(A)*u ) );
% x = inv(A) * ( u-( B*y ) );
% X = [x;y];

res_Bu = A \ [u,B];
%res_Bu = bcgacc(A,[u,B],p,202);
% via Block CG accelereated Block Cimmino algorithm

%res_B = res_Bu(:,1:size(B,2));
%res_u = res_Bu(:,size(B,2)+1:end);

CB = C *res_Bu;
Dd = D- CB(:,2:end) ;

```

```

vC = v -CB(:,1);

y = Dd \ vC;

uB = u-( B*y );
x = A \ uB;
%x = cgacc(A,uB,p,200);
% via classical CG accelerated Block Cimmino algorithm

% y = inv( D - ( C*res_B ) ) * ( v - ( C*res_u ) );
% x = inv(A) * ( u-( B*y ) );
X = [x;y];
X(colp) = X;
return;
}

```