ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

ON THE ANALYSIS AND EVALUATION OF SPARSE HYBRID LINEAR SOLVERS

M.Sc. THESIS

Afrah Najib FAREA

Department of Computational Science and Engineering

Computational Science and Engineering Programme

JUNE 2018

ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

ON THE ANALYSIS AND EVALUATION OF SPARSE HYBRID LINEAR SOLVERS

M.Sc. THESIS

Afrah FAREA
(702151010)

Department of Computational Science and Engineering

Computational Science and Engineering Programme

Thesis Advisor: Prof. Dr. M.Serdar.ÇELEBI

JUNE 2018

ISTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

SPARSE HİBRİT DOĞRUSALININ ANALİZİ VE DEĞERLENDİRİLMESİ ÇÖZÜCÜLER

YÜKSEK LİSANS TEZİ

Afrah FAREA
(702151010)

Bilişim Enstitüsü
Hesaplamalı Bilim ve Mühendisliği Programı

Tez Danışmanı: Prof. Dr. M.Serdar.ÇELEBI

Haziran. 2018

Afrah FAREA, a M.Sc. student of İTU Graduate School of Science Engineering and Technology student ID 702151010, successfully defended the thesis/dissertation entitled “ON THE ANALYSIS AND EVALUATION OF SPARSE HYBRID SOLVERS”, which she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. M.Serdar.ÇELEBI** İstanbul Technical University	....
Jury Members :	**Prof. Dr. Mine Çağlar** Koş University	....
	Prof. Dr. Hakan Akyıldız İstanbul Technical University	....

Date of Submission : 21 May 2018
Date of Defense : 08 JUNE 2018

To my explicit zero entries...

FOREWORD

I would like to thank my supervisor Prof.Dr.M.Serdar.Çelebi for introducing me to the high performance computing and the parallel world through his cources and this thesis. It is a turning point in my career.

I would also like to thank people at Ulusal Yüksek Başarımlı Hesaplama Merkezi Projesi (UHeM) for the grant and answering my questions whatever naive they are. Many thanks to Superlu, Maphys, Pt-scotch, spack, stack overflow and PETSc developers for answering my questions. Many thanks to Prof.Dr.Tim Davis for Florida Sparse Matrix collection. Life would be difficult without this repository.

JUNE 2018 Afrah FAREA

TABLE OF CONTENTS

Page

ABBREVIATIONS

AMD	: Approximate Minimum Degree
BFS	: Breadth First Search
COO	: COOrdinate format
CSC	: Compressed Sparse Column
CSR	: Compressed Sparse Row
CG	: Conjugate Gradient
DFS	: Depth First Search
FGMRES	: Flexible Generalized Minimum Residual
FLOPS	: floating Points per second
GMRES	: Generalized Minimum Residual
GPU	: Graphics Processing Unit
HPC	: High Performance Computing
MPI	: Message Passing Interface
MMD	: Multiple Minimum Degree
MTX	: MaTriX market format
ND	: Nested Dissection
RSS	: Resident Set Size
SPD	: Symmetric Psitive Definite
RSA	: Rea Symmetric Assembled
RUA	: Real Unsymmetirc Assembled format

LIST OF TABLES

LIST OF FIGURES

Page

ON THE EVALUATİON AND ANALYSİS OF SPARSE HYBRİD SOLVERS

SUMMARY

A matrix is called sparse if many of its entries are zero. Such types of matrices are generated from discretization problems of many fields like numerical simulations, Fluid Dynamics, Graph theory, Optimization problems, Signal processing, Finance, industry, Linear Programming, Electromagnetics, 2D/3D and many other real-world applications. In general, we call a matrix sparse if we could exploit the sparsity of its elements in terms of memory storage and computation. Different sparse matrix formats are proposed which lead to huge memory and computation savings.

Similarly, solving large sparse linear systems becomes increasingly important as a kernel task for such scientific applications. Furthermore, recent years witnessed huge and complex development in modern microprocessor architecture and hardware in general. Besides, the parallel computing methods and languages has shown a kind maturation in solving real-world problems especially, the development of Message Passing Interface (MPI). Consequently, many algorithms and software packages have emerged to exploit the new developments in hardware and software alike. For instance, the development in the hierarchy of memory and computation nodes leads to developing new blocking algorithms which accommodate the small cache memory of modern architecture and increase the floating-point performance.

In general, there are two broad categories for solving linear systems mathematically: direct and iterative methods. Direct methods can give high accurate results (10^{-30}) and are more suitable for small problems (current direct methods can solve up to a couple of millions of equations) because of the memory consumption they require (for example, they can not solve large sparse 3D problems which may generate hundreds of millions of unknowns). Besides, they have limited parallel scalability. Preconditioned iterative methods, on the other hand, are more robust, require less memory and easier to parallelize. However, they are problem dependent and can converge faster with good preconditioning methods. These methods are the methods of choice when an approximate solution of the problem is sought. Our focus in this thesis is the parallel software packages used for solving large sparse linear systems. We investigated the various methods and techniques for solving sparse linear systems and concentrate on the open source hybrid approaches as they are more flexible and well-suited the hierarchal structure of modern computers.

The word hybrid has different meanings. Hybrid in programming means combining OpenMP and MPI so they work for shared and distributed memory. Hybrid in hardware means working on CPU and GPU. Hybrid in algorithms means combining direct and iterative methods. So, our focus is on the third meaning; combining direct and iterative algorithms in order to get more efficient methods for solving sparse matrices. Therefore, throughout this dissertation, when we mention the word hybrid, we mean direct/ iterative methods.

Hybrid solvers are the latest research in developing robust and scalable methods for solving sparse linear systems. These methods combine direct and iterative approaches in certain ways, mostly using Schur complement framework, with the aim of getting the desired features of both direct and iterative methods especially the speed and low memory consumption of the iterative methods and the robustness of the direct methods.

Most sparse hybrid solvers use the so called ‘Schur complement framework’ to combine direct and iterative methods. In this framework, the original matrix is divided into 2×2 block matrices. This approach is also known as sub-structuring domain decomposition method. Graph theory algorithms are responsible for ordering the global matrix and getting such block structure. The first block is a large block diagonal square matrix. Each diagonal block, called internal subdomain, is factorized using direct methods. The second and third block are the interfaces. If the matrix is symmetric, these interface blocks are transpose of each other. The fourth block is a square matrix called Schur complement matrix. This matrix has many desired features well-suited for iterative methods. It has smaller size than the original matrix and it is better conditioned than the original matrix. If the original matrix is symmetric positive definite, this Schur matrix will inherit this property and thus Conjugate Gradient can be used. Although Schur matrix is better conditioned than the original matrix, preconditioning the matrix before applying iterative method is necessary for faster convergence. The factorized internal subdomains are used to get an approximation of the Schur complement matrix and used as a preconditioner.

Solving large sparse linear systems can take days even with algorithmic hybrid approaches if performed in sequential. Schur complement framework is more appealing for parallel programming. With increasing development of hardware and supercomputers, these systems can be solved efficiently within few seconds. However, existing algorithms should be modified to accommodate the parallel environment constrains such as scalability and load balancing.

PDSLin and Maphys are the best existing public domain Schur-complement based hybrid solvers. They are based on different preconditioning methods which is a crucial ingredient in any iterative solver. PDSLin uses an approximation of the Schur complement as a precontitioner which gives a global view of the domain problem. Maphys uses an approximation of the local assembled Schur matrix which makes the solver more scalable. In our experiments, we thoroughly examined these solvers, test them on different matrix types and compare their results with the state-of-art Superlu-dist direct solver. We also investigated the effect of tuning preconditioning input parameters on PDSLin and Maphys with increasing number of processors.

Developed at Lawrence Berkeley National Laboratory (LBNL) by two distinguishing researchers in parallel linear algebra Ichitaro Yamazaki and X. Sherry Li; PDSLin solver is very robust and scales very well with increasing number of processors. However, it is very sensitive to the input parameters especially sparsifying tolerances which is not good. PDSLin is a powerful solver but has a lot of bugs and still needs a lot of work. Our results show that Maphys solver is more stable with the input values than PDSLin and gives better results. PDSLin results are unpredictable and sometimes fails with the slightest change of the input values. Maphys performance is better than both PDSLin and Superlu-dist in terms of time consumption and memory. Besides, the two-level parallelism of PDSLin is more robust than multithreading of Maphys. The serial partitioning time of Maphys is significant in many cases of our experiments and thus it is better to change into a parallel graph partitioner in Maphys than using a sequential partitioner. Sometimes the partitioning time is larger than summation of the other solution steps of Maphys and this is clearly shown in Audik and Freescale cases. Maphys solver also scales very well with increasing number of processors.

Our conclusion is that sparse hybrid solvers are more flexible because they have different components and each component can be substituted to accommodate the problem at hand. The developers of the solvers we considered have already aware of this feature and this is clearly seen through the different methods they integrated within their solvers. For example, PDSLin uses either MUMPS or Superlu-dist as a direct method which are different approches of factorizing matrices. Similarly, Maphys uses either MUMPS or PASTIX which also are also different.

Using two level parallelism can make hybrid solvers work efficiently and scalable up to thousand number of processors. In this level, Processors are distributed into levels and work concurrently and independently. This method alleviates the problem of increasing Schur complement size. However, load balancing is a challenging problem in this method. Hybrid solvers consume much less amount of memory than either pure direct or iterative methods.

SEYREK HİBRİD ÇÖZUCÜLERİN DEĞERLENDİRİLMESİ VE ANALİZİ

ÖZET

Birçok girdisinin sıfır olması durumunda bir matris seyrek olarak adlandırılır. Bu tür matrisler, sayısal simülasyonlar, Akışkanlar Dinamiği, Grafik Teorisi, Optimizasyon Problemleri, Sinyal İşleme, Finans, Endüstri, Doğrusal Programlama, Elektromanyetik, 2D / 3D ve diğer pek çok gerçek uygulama gibi birçok alanın ayrıklaştırma problemlerinden kaynaklanır. Genelde, bellek depolama ve hesaplama açısından öğelerinin seyrekliginden yararlanabilirsek ona seyrek matris diyoruz. Büyük bellek ve hesaplama tasarruflarına yol açan farklı seyrek matris formatları önerilmiştir.

Benzer şekilde, büyük sparse lineer sistemlerin çözümü, bu tür bilimsel uygulamalar için bir çekirdek görevinde olduğundan giderek önem kazanmaktadır. Dahası, son yıllarda modern mikroişlemci mimarisinde ve genel olarak donanımda büyük ve karmaşık gelişmeler yaşandı. Ayrıca, paralel hesaplama yöntemleri ve dilleri, gerçek dünyadaki problemleri çözmede, özellikle Mesaj Geçiş Arayüzü (MPI) geliştirilmesinde bir çeşit gelişme göstermiştir. Sonuç olarak birçok algoritma ve yazılım paketi donanımda yeni gelişmelere kapı açtı. Örnek verilecek olunursa, bellek ve hesaplama düğümlerinin hiyerarşisindeki gelişme, modern mimarinin küçük önbelleği barındıran ve floating point performansını arttıran yeni engelleme algoritmalarının geliştirilmesini sağlar.

Genel olarak, doğrusal sistemleri matematiksel olarak çözmek için iki ana kategori vardır: doğrudan ve yinelemeli yöntemler. Doğrudan yöntemler yüksek doğrulukta sonuçlar verebilir (10^{-30}) ve ihtiyaç duydukları düşük hafıza tüketimi nedeniyle (son yıllarda doğrudan yöntemler birkaç milyon odf denklemini çözebilir) problemler için daha uygundur (örneğin, Büyük Sparse 3D problemlerini çözemezler). Bunun yanında paralel boyutlandırmayı sınırlar. Öte yandan Preconditioned iterasyon yöntemleri daha sağlıklıdır, daha az bellek gerektirir ve paralelleştirilmesi daha kolaydır. Ancak, bunlar problem bağımlıdırlar ve iyi preconditioning metodlarla daha hızlı yakınsanabilir. Bu tezdeki odak noktamız, büyük Sparse lineer sistemlerini çözmek için kullanılan paralel yazılım paketleridir. Spars lineer sistemlerin çözümü için çeşitli yöntemler ve teknikler araştırdık ve modern bilgisayarların hiyerarşik yapısı daha esnek ve uygun oldukları için açık kaynaklı hibrit yaklaşımlara yoğunlaştık.

Hibrit kelimesinin farklı anlamları vardır. Programlamada hibrit, OpenMP ve MPI dillerini birleşimi anlamına gelmektedir çünkü paylaşımlı ve distrübe edilmiş hafızayı çalıştırabilirler. Donanımda ise hibrit, CPU ve GPU'nun birlikte çalışmasıdır. Hibrit algoritmada doğrudan ve iteratif yöntemleri birleştirmek anlamına gelir. Odak noktamız olan algoritma hibriti; daha verimli bir yöntem oluşturmak için doğrudan ve iteratif yöntemleri birleştirir. Bu nedenle, bu tez boyunca hibrit sözcüğünden bahsettiğimizde, doğrudan ve yinelemeli yöntemleri birlikte kastetmiş olacağız.

Çoğu seyrek hibrit çözümleyicileri "Schur complement framework" yöntemini kullanır. Buradaki amaç doğrudan ve iteratif metodları birleştirmektir. Bu çerçevede, orijinal matris 2×2 blok matrisine ayrılmıştır. Grafik teorisi algoritmaları matrisin düzenlenmesi ve bu blok yapısının elde edilmesine yarar. İlk blok büyük bir blok

köşegen kare matrisidir. Her diagonal blok iç alt domain olarak adlandırılır ve direk metodlar kullanılarak çarpanlarına ayrılırlar. İkinci ve üçüncü bloklar "interface"lerdir. Eğer matris simetrik ise bu interface bloklar birbirlerinin transpozesidir. Dördüncü blok Schur komplement matris olarak adlandırılan kare bir matristir. Bu matris asıl matristen daha küçüktür ve kullanmak için daha uygundur. Eğer asıl matris SPD matris ise Schur komplement matris de SPD matristir. Aynı zamanda Schur matris kullanmak asıl matrisi kullanmaktan daha kolaydır. Matrisi daha önceden preconditioning etmek daha hızlı bir yakınsama için gereklidir. Çarpanlarına ayrılmış iç alt domainler Schur complement matris yaklaşımı ve precondition yapmak için kullanılır.

Büyük seyrek lineer sistemlerini çözmek sekansiyel olarak çalıştırıldığında, algoritmik hibrit yaklaşımlarla bile günler alabilir. Schur tamamlayıcı çerçeve paralel programlama için daha fazla tercih edilir. Donanım ve süper bilgisayarların gelişmesiyle birlikte, bu sistemler birkaç saniye içinde verimli bir şekilde çözülebilir. Ancak, var olan algoritmalar ölçeklenebilirlik ve yük dengesi gibi paralel çevresel kısıtlamalara uyum sağlayabilmek için modifiye edilmelidir.
PDSLin ve Maphys, mevcut en iyi hibrit çözücü tabanlı açık kod Schur-complement'dir. Bu çözücüler üzerinde derinlemesine inceleme yaptık, bunları farklı matris türlerinde test ettik ve sonuçlarını son teknoloji Superlu-dist doğrudan çözücüsü ile karşılaştırdık.
Sonuç olarak, seyrek hibrit çözücüler daha esneklerdir çünkü farklı bileşenlere sahiplerdir ve her bileşen mevcut probleme uyarlanabilecek şekilde diğerinin yerine geçirilebilir. İki seviyeli paralellik kullanmak, hibrit çözücülerin verimli olmasını ve bin adet işlemci sayısına kadar ölçeklenebilir olmasını sağlayabilir. Hibrit çözücüler, doğrudan veya iteratif yöntemlerden çok daha az miktarda bellek kullanılır.

Büyük seyrek lineer sistemlerini çözmek sekansiyel olarak çalıştırıldığında, algoritmik hibrit yaklaşımlarla bile günler alabilir. Schur tamamlayıcı çerçeve paralel programlama için daha fazla tercih edilir. Donanım ve süper bilgisayarların gelişmesiyle birlikte, bu sistemler birkaç saniye içinde verimli bir şekilde çözülebilir. Ancak, var olan algoritmalar ölçeklenebilirlik ve yük dengesi gibi paralel çevresel kısıtlamalara uyum sağlayabilmek için modifiye edilmelidir.
PDSLin ve Maphys, mevcut en iyi hibrit çözücü tabanlı açık kod Schur-complement'dir. Bu çözücüler üzerinde derinlemesine inceleme yaptık, bunları farklı matris türlerinde test ettik ve sonuçlarını son teknoloji Superlu-dist doğrudan çözücüsü ile karşılaştırdık.
Sonuç olarak, seyrek hibrit çözücüler daha esneklerdir çünkü farklı bileşenlere sahiplerdir ve her bileşen mevcut probleme uyarlanabilecek şekilde diğerinin yerine geçirilebilir. İki seviyeli paralellik kullanmak, hibrit çözücülerin verimli olmasını ve bin adet işlemci sayısına kadar ölçeklenebilir olmasını sağlayabilir. Hibrit çözücüler, doğrudan veya iteratif yöntemlerden çok daha az miktarda bellek kullanılır

Büyük seyrek lineer sistemlerini çözmek sekansiyel olarak çalıştırıldığında, algoritmik hibrit yaklaşımlarla bile günler alabilir. Schur tamamlayıcı çerçeve paralel programlama için daha fazla tercih edilir. Donanım ve süper bilgisayarların gelişmesiyle birlikte, bu sistemler birkaç saniye içinde verimli bir şekilde çözülebilir. Ancak, var olan algoritmalar ölçeklenebilirlik ve yük dengesi gibi paralel çevresel kısıtlamalara uyum sağlayabilmek için modifiye edilmelidir.

PDSLin ve Maphys, mevcut en iyi hibrit çözücü tabanlı açık kod Schur-complement'dir. Bu çözücüler üzerinde derinlemesine inceleme yaptık, bunları farklı matris türlerinde test ettik ve sonuçlarını son teknoloji Superlu-dist doğrudan çözücüsü ile karşılaştırdık.
Sonuç olarak, seyrek hibrit çözücüler daha esneklerdir çünkü farklı bileşenlere sahiplerdir ve her bileşen mevcut probleme uyarlanabilecek şekilde diğerinin yerine geçirilebilir. İki seviyeli paralellik kullanmak, hibrit çözücülerin verimli olmasını ve bin adet işlemci sayısına kadar ölçeklenebilir olmasını sağlayabilir. Hibrit çözücüler, doğrudan veya iteratif yöntemlerden çok daha az miktarda bellek kullanılır

1. INTRODUCTION

Recent years witnessed huge and complex development in modern microprocessor architecture and hardware in general. Besides, the parallel computing methods and languages has shown a kind maturation in solving real-world problem especially, the development of Message Passing Interface(MPI). Consequently, many algorithms and software packages have emerged to exploit the new developments in hardware and software alike. For instance, the development in the hierarchy of memory and computation nodes leads to developing new blocking algorithms which accommodate the small cache memory of modern architecture and increase the floating point performance.

Similarly, solving large sparse linear systems becomes increasingly important as a kernel task for many scientific applications. Our focus in this thesis is the parallel software packages used for solving large sparse linear systems. We investigate on the various methods and techniques for solving sparse linear systems and mainly focus on the open source hybrid approaches as they are more flexible and well-suited the hierarchal structure of modern computers.

Key words: hybrid, hierarchal, matrix, parallel, sparse, linear, solver, pdslin , maphys , hips , pArms, Schur complement, additive schwarz , direct , iterative, LU factorization , Krylov methods, GMRES, CG, partitioning, preconditioning.

1.1 Literature Review

The problem of solving sparse linear systems was extensively studied in the last decade. There are many good parallel sparse linear packages developed over the years. In general, these packages are categorized according to the algebraic method used into direct and iterative. The direct solver packages are more mature than iterative solver packages and they gain more attention from the research community. Here we survey the most common distributed memory parallel packages in both methods starting with direct solvers.

PARDISO[1] is a left-right looking supernodal hybrid programming (MPI + threads) direct solver with dynamic scheduling of processes to tasks. It supports many types of symmetric and unsymmetric sparse matrices. Complete pivoting or Bunch and Kaufmann supernode pivoting is required. The main algorithm works as follows: First, ordering and symbolic factorization is done using ordering and symbolic factorization algorithms. Then, block numerical factorization is performed in the resulting elimination tree by factorizing groups of columns at a time. At supernode $\boldsymbol{l_i}$, an "external factorization" is performed for this supernode with left-looking by gathering contributions from previously factorized supernodes. The result is then gathered to the destination supernode. A set of optimization techniques are implemented in this solver such as assembly is separated from floating point operations, pivoting is restricted to supernode diagonal blocks and BLAS-3 pipelining parallelism during numerical factorization and out-of-core capability in which the disc is used as an extension of the main memory.

MUMPS[3] is a multifrontal direct method with dynamic pivoting(more about multifrontal methods on section(4.2)). Different ordering methods are supported in MUMPS such as AMD, QAMD, PORD, ND, METIS and AMF. User defined ordering can also be used. In this solver, first ordering and symbolic factorization is performed on the symmetrized matrix $\boldsymbol{A} + \boldsymbol{A}^{\boldsymbol{T}}$. The result is the elimination tree and a mapping of the multifrontal computation graph. According to the mapping computed in the previous step, numerical factorization is carried out on the multifrontal dense matrices. The factorized matrices are then used for finding the solution. Iterative refinement and backward error analysis are among the options. Similar to PARDISO, MUMPS allows dynamic scheduling of processors to tasks and out-of-core capability. The new versions of MUMPS support multithreading.

PASTIX[4] is a left-looking supernodal multithreading solver with static pivoting. It follows the same steps of the previous solvers for solving sparse linear systems(ordering, symbolic factorization, numerical factorization, solve, refinement). Metis is recommended for partitioning and halo approximate minimum degree algorithms is used if the subgraph is smaller than a specified threshold. Out-of-core capability is also supported. GMRES, CG or simple iterative refinement can be used to incease the precision of the solution. Newer verions of Pastix supports different low-rank compression techniques.

Developed by IBM, **WSMP**(Watson Sparse Matrix Package) is a hybrid programming (MPI + threads) multifrontal package for solving sparse symmetric/unsymmetric linear systems. Nested dissection ordering on the symmetrized matrix $\boldsymbol{A} + \boldsymbol{A^T}$ with some heuristics for more robustness. The method of this solver separates symbolic analysis from numerical factorization which experimentally gives better performance. Threshold pivoting is used with automatic option for the threshold value. The solver also has the capability of solving multiple linear systems with the same sparsity pattern using ordering and analysis of a single matrix.

PSPASES(Parallel SPArse Symmetric dirEct Solver) is a multifrontal solver for solving SPD sparse matrices with no pivoting required. This solver uses the same solution steps as the previous methods with subtree-to-subcube static mapping of processors to nodes in the elimination tree and cyclic mapping of rows and columns of the frontal matrices within each subgroup. ND of Metis is used for ordering and graph partitioning. Although this two-level processor mapping adds to the scalability of Cholesky factorization of this method; the expenses are load imbalance and communication overhead.

SUPERLU-DIST is a supernodal right-looking distributed sparse solver with shared memory capability for many core systems and GPU option. Sequential or parallel ordering options are available using multiple minimum degree(MMD), or nested discretion algorithm of Metis or Parmetis on $\boldsymbol{A^T A}$ or $\boldsymbol{A^T} + \boldsymbol{A}$ graph and static pivoting is used. Users can also provide their own ordering, overriding the defaults. Parallel symbolic factorization is an option with column permutation using Parmetis. Different factorization options are also available. For algorithmic stability, tiny diagonal pivot can be replaced by a small perturbation value and iterative refinement at the end of the solution for more accurate results. The latest version of SUPERLU-DIST can be compiled without Parmetis dependency.

Developed by Argonne National Lab, **PETSc** (Portable, Extensible Toolkit for Scientific Computation) is a set of sparse matrix tools written in C/Fortran and Python with object orientation programming support. The library includes a variety of Krylov subspace iterative methods, preconditioners, different orthogonalization schemes and refinement options.

WSMP (Watson Sparse Matrix Package) has an iterative version for shared memory environment only. The iterative package supports symmetric/ unsymmetric types of matrices. Different iterative solvers are supported like CG, GMRES, TFQMR and BiCGStab. The preconditioners supported include Jacobi, Gauss-Siedel and incomplete $\boldsymbol{LDL^T}$/ $\boldsymbol{LU}$ depending on the matrix type.

Although **pARMS** (Parallel Algebraic Recursive Multilevel Solver) can work as a hybrid direct/iterative solver(section 6.4), it is actually a set of iterative solvers and preconditioners for solving sparse linear systems in distributed memory environment. The main accelerator in pARMS is the preconditioned FGMRES. Three classes of preconditioners are available: Schwarz Preconditioners, single-level Schur Preconditioners and multi-level Schur Preconditioners.

Developed by Sandia National Laboratories, **Aztec** [5] is an iterative library for solving general sparse linear equations on distributed parallel systems. Written in ANSI-C standard, Aztec supports different types of iterative methods and preconditioners. Aztec supports two sparse matrix formats: Distributed Modified Sparse Row (DMSR) and Distributed Variable Block Row format (DVBR). The new version of Aztec supports simpler data formats allowing users to specify rows in a natural order.

Developed by Lawrence Livermore National Laboratory(LLNL), **HYPRE** (High Performance preconditioners) is set of multigrid algorithms and software for distributed and shared memory environment with emphasis on scalable parallel preconditioners. According to the problem provided by the user, different conceptual interfaces are available like structured grid, finite element and linear algebra interfaces. Multigrid preconditioners supported include semiconductor multigrd(SMG), BoomerAMG and ParaSail.

1.2 Related Work to Experimental Comparison of Hybrid Solvers

Unfortunately there is no much research comparing the performance of available hybrid solvers. We mention here the studies we found.

In [4], they compared PDSLin with Hips using **Tdr455k** matrix on Cray XT4 machine at NERSC. They mentioned that Schur complement got larger with increasing number of subdomains in Hips because of the one-level parallelism technique used.

Consequently, the convergence became slower and the number of iterations increased from 151 iterations on 16 processors, and it failed to converge with 1000 iterations on 32 processors. Authors in [5-7] mentioned this experiment in table (1.1). The results show that number of iterations did not increase much as in the case of Hips and PDSLin continues to converge with increasing number of processors.

Table 1.1 : Results optained in [4] for HIPS vs PDSLin using Tdr455k.

P	N_s	HIPS1.0 Sec (iter)	PDSLin Sec (iter)
8	13k	284.6 (26)	79.9
32	29k	55.4 (64)	25.3 (16)
128	62k	--	17.1 (16)
256	124k	--	21.9 (17)

In a Siam conference(not published), Maphys developers compared their solver's performance against PDSLin using **Audik-1**, **Haltere** and other matrices. The version of Maphys used in that experiment was one level parallelism (they did not add 2 level parallelism feature at that time) so they removed some features from PDSLin such that both solvers work on the same foot. The partitioner used in Maphys is sequential while the partitioner used in PDSLin is parallel. The results show a good scaling for PDSLin with increasing number of processors but the overall performance was approximately similar.

We repeated this experiment with the last versions of PDSLin and Maphys currently available on Sariyer Cluster using up to 16 nodes. The details of this experiment is shown in the result chapter.

An important study of hybrid solvers performance evaluation is the one mentioned in [8]. In their study, they ran a set of experiments on PDSLin and Maphys using **Matrix211** and **Tdr455k** matrices. In order to compare these solvers, they selected the following options:

- They used the same partitioning tool for both solvers.
- They used the same stopping criteria for both solvers. $\epsilon_p = \frac{||SX_\tau - \widehat{b_\tau}||}{||\, b \,||}$ where X_τ is the variables corresponding the interfaces τ and S, $\widehat{b_\tau}$ are Schur complement and its corresponding right hand side(see section 6.1 and   for Schur compelement and right hand side equations). **b** is the right hand side of

the original system **Ax = b** and ||.|| is the 2-norm. They also set $\epsilon_b = 10^{-10}$ as the error threshold.

- They set the other control parameters for the two solvers with aim to minimize the parallel time of the solution.
- The experiments were run on **Hopper** machine of Lawrence Berkeley National Laboratory.

These results show that the overall performance of Maphys (**M)** is better than that of PDSLin(**P**) in term of elapsed time. PDSLin scales better with increasing number of cores especially with **Tdr455k** matrix. The factorization time for PDSLin is larger than that of Maphys for both matrices but the solution time is smaller and this is an advantage for PDSLin in case of multiple right hand side. The best configuration of PDSLin is with 1536 cores for both matrices. With Maphys, 1536 gives the best result with **Matrix211** and 384 with **Tdr455k**.

In appendix(A) , table (A.1), we show some of the common matrices in literature used for evaluating those hybrid solvers along with some other information. Unfortunately, all publications use matrices not publically available except **Audik** matrix from Florida university collection. In column **Con.No**, we mention the hybrid solvers used with the corresponding matrix.

The remainder of this dessetation is devoted to different algorithms and techniques used in the components of the hybrid solvers. Chapter 2 is an introduction to sparse matrices and sparse matrix formats. Due to its crucial role in sparse matrix algorithms, we devoted Chapter 3 to the ordering and graph partitioning algorithms. Chapter 4 and Chapter 5 summarize the common direct and iterative methods respectively. In Chapter 6, we discuss the public domain hybrid solvers we found and the main algorithms used in each one of them. In Chapter 7, we discuss our experiments on the two best hybrid solvers we found; Maphys, PDSLin and compare results with Superlu-dist direct solver. Finally, the conclusion and future work in chapter 8.

2. SPARSE MATRICES

A matrix is called sparse if many of its entries are zero. Such types of matrices are generated from discretization problems of many fields like numerical simulations, Fluid Dynamics, Graph theory, Optimization problems, Signal processing, Finance, industry, LinearPprogramming, Electromagnetics, 2D/3D and many other real-world applications. Thus a reliable solution of such systems are more important than ever. In general, we call a matrix sparse if we could exploit the sparsity of its elements in terms of memory storage and computation [11]. In this section, we will briefly discuss the common data structures used for storing sparse matrices.

2.1 Sparase Matrix Storage Foratms

There are many varieties of sparse matrix formats, each tailored to specific application and matrix structure. In this section, we discuss the most familiar sparse matrix format.

2.1.1 Coordinate Format

Coordinate format (COO) stores the non-zeros value along with their row/column indices. No ordering constraints imposed on the coordinate format. Experiments show that this format is significantly slower by orders of magnitude[12].

2.1.2 Compressed stripe storage Format (CSR, CSC).

This format is considered the default sparse format by many sparse packages. In CSR, we put the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a non-symmetric sparse matrix A, we create three vectors:

- Floating point vector (val) for storing nonzero entries of the matrix A as they are traversed in a row-wise fashion.
- Integer vector col_ind which stores the column indexes of the elements in the val vector. That is, if val(k) = a_{ij}, then col_ind(k)=j.
- Integer vector row_ptr which stores the locations in the val vector that start a row; that is, if val(k) = a_{ij}, then row_ptr(i) $\leq$ k < row_ptr(i+1).

By convention, we define row_ptr(n+1) = nnz+1, where nnz is the number of nonzeros in the matrix A. Figure(2.1) shows an example of CSR for a nonsymetric matrix.

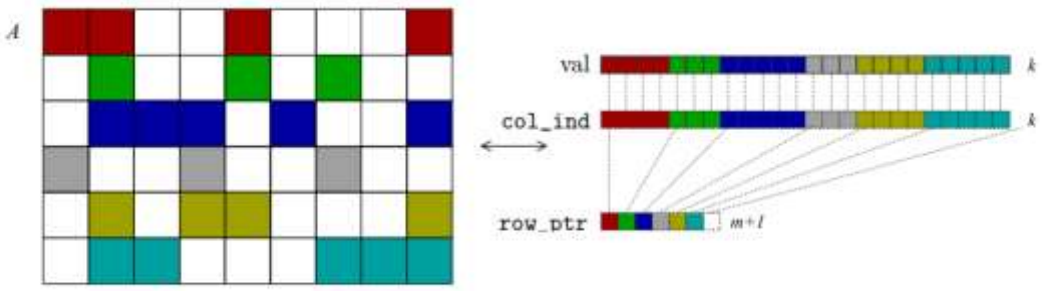


Figure 2.1 : Example of Compressed Sparse Row (CSR) format [12].

A slightly different variation is Modified Sparse Row format (MSR) in which the diagonal elements are stored in a separate array diag-val and the other off-diagonal elements are compressed using CSR.

A similar compression format is compressed sparse column (CSC) (also known as Harwell-Boeing) [3]. In this form, the columns are traversed instead of rows.Thus, CSC is the CSR of A^T. Similar to CSR,we use three arrays (val, row_ind, col_ptr), where row_ind stores the row indices of each nonzero, and col_ptr stores the index of the elements in val which start a column of A.

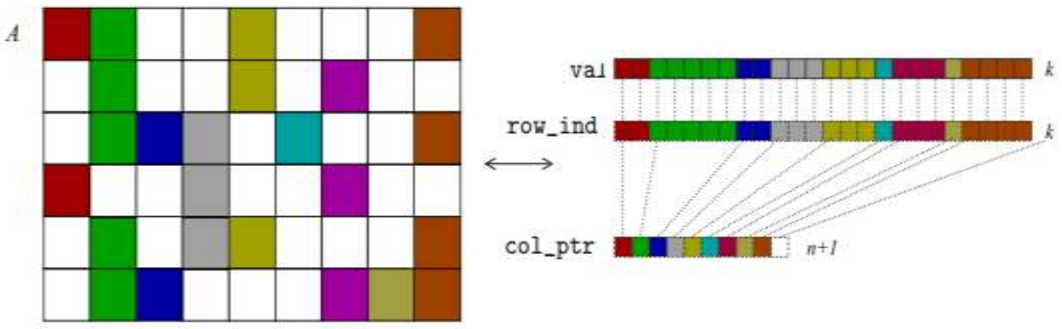


Figure 2.2 : An example of Compressed sparse Column (CSC) format [12].

The maximum storage requirement for CSC is $\boldsymbol{\theta}$(2nnz + n +1) and for CSR $\boldsymbol{\theta}$(2nnz + $\frac{\boldsymbol{n}}{\boldsymbol{2}}$+1). The maximum storage for uncompressed form is $\boldsymbol{\theta}$ ($\boldsymbol{n^2}$) which shows a huge saving of memory

2.1.3 Diagonal Format(DIAG)

This format is used for full nonzero diagonal matrices like those generated from stencil calculations. No need to store individual nonzero elements. Only diagonal indices need to be stored. According to the this numbering convention: the main diagonal is numbered zero, upper diagonals have positive numbers and lower diagonals have negative numbers. Nonzero entry at position(i,j) lies on diagonal number(j-i). Figure(2.3) shows an example of such format. Thus we need two data structures:

- A matrix **val** of size(m × s) for storing nonzero elements where **m** is the size of the diagonal and **s** is the number of diagonals. Since the diagonal carries the largest number of nonzero elements, there are padding places in **val** as shown in figure(2.3).
- An array **diag-num** of size **S** for storing the index of the first element of each diagonal.

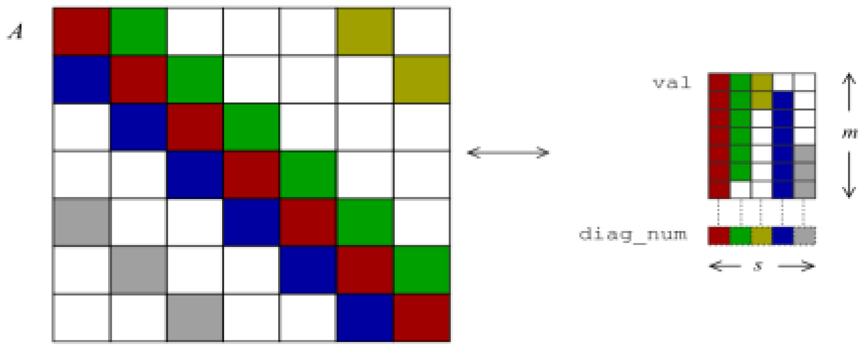


Figure 2.3 : An example of Diagonal Format (DIAG) format [12].

2.1.4 ELLPACK/ITPACK format(ELL)

This format was originally developed for ELLPACK and ITPACK sparse solvers and mostly suitable for matrices with almost same number of nonzero entries for each row. In ELL, two arrays are needed each of size $\boldsymbol{m} \times \boldsymbol{s}$ where m is the number of rows in A and s is maximum number of nonzero entries in any row in A. The first array val is used for storing nonzero values and the second **ind** for storing column indices for each row. Figure (2.4) shows an example.

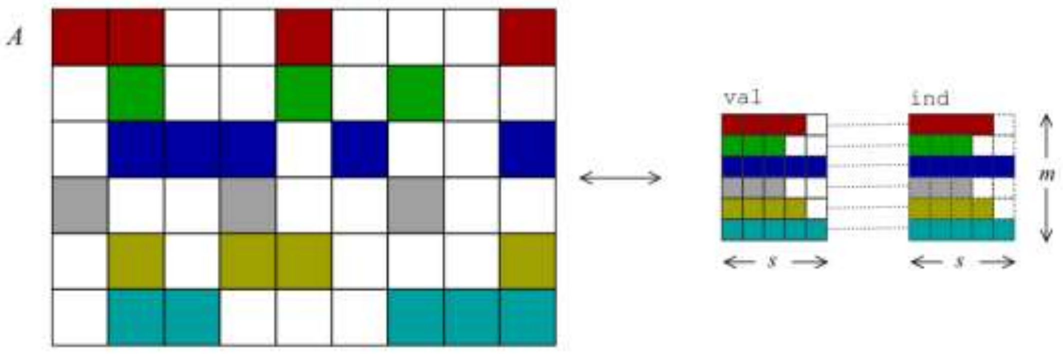


Figure 2.4 : An example of ELLPACK/ITPACK format (ELL) [12].

2.1.5 Block Compressed Stripe Formats

Block Compressed Stripe format is a generalization of compressed stripe format we discussed earlier. It is most suitable for sparse matrices with fixed block structures. As in compressed stripe format, three data structures are used **val**, **ind** and **ptr**. Unlike Compressed stripe format, **val** here is a matrix of size (rK) ×c where r × c is the block size and K is the number of blocks. The blocks are treated as full dense blocks so padding is required for filling any zero elements if necessary. Besides, blocking is not unique here. Figure(2.5) shows two different blocking arrangements for the same matrix.

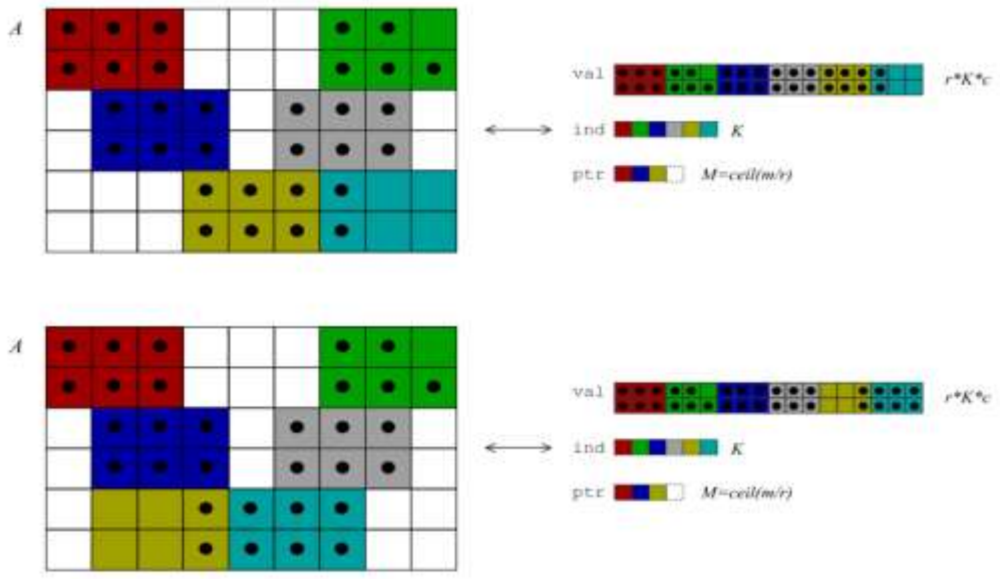


Figure 2.5 : Example of Block Compressed Stripe Format [12]

2.2 Experimental Comparison of the Basic Formats

Experiments in SpMV show that CSR and MSR give the best performance on a wide class of matrices and either one of them can be used as a default format if we do not know the structure of the matrix at hand[12]. In our experiments, we used RUA, MTX and IJV formats depending on the solver. For Maphys solver we used RUA and MTX. For PDSLin, we used RUA and IJV formats and for Superlu-dist we sued RUA format.

3. ORDERING AND GRAPH PARTITIONING SCHEMES

Graph theory is closely related to sparse matrices because of their sparsity structure. The best way to keep track of the nonzero elements in the sparse matrices is though graphs showing the connection between those nonzero entries. According to their structures, sparse matrices can be categorized into structured or unstructured. Structured matrices follow a certain regular pattern. Examples of such matrices are those generated from finite difference problems on rectangular grid. As we will see throughout this chapter, the main purpose ordering is to reduce fill-in and thus memry consumption during LU factorization.

In this chapter, we present the basic concepts, the most common algorithms, parallel implementation and the software packages for reordering and graph partitioning.

3.1 Basic Definitions

The graph **G** consists of two finite sets (V,E), where **V** is the set of vertices/nodes, them.

$$V = \{v_1, v_2, v_3, \dots, v_n\}$$

E is the set of edges connecting two ordered pairs of vertices where $v_i, v_j \in$ **V**. A graph is called undirected graph if for all where $v_i \; and \; v_j \in$ **V**:

$$(v_i, v_j) \in \mathbf{E} \Leftrightarrow (v_j, v_i) \in \mathbf{E}$$

Otherwise, the graph is directed(digraph). A degree of a vertix is the number of edges connecting to the node, denoted by deg(v_i)

A **path** in a graph is a sequence of vertices $v_1, v_2, v_3, \dots, v_k$ such that (v_i, v_{i+1}) is an edge in the graph. A closed path is known as a **Cycle**. A graph with no cycles is a **Tree**.

A graph $G'(V', E')$ is called subgraph of **G** if $V' \subseteq V$ and $E' \subseteq E$. A graph is called a **complete graph** (or strongly connected graph) if every pair of nodes are adjacent. A **clique** is a subgraph of the complete graph.

In sparse matrices, an edge a_{ij} is drawn between equation (**i**) and unknown (**j**) if its value is nonzero. i.e $a_{ij} \neq 0$. Moreover, if for each $a_{ij} \neq 0$, there exists $a_{ji} \neq 0$, then the matrix has a **symmetric nonzero structure(pattern symmetry)**. In such cases, undirected graph is used. For non-square matrices, **bipartite graph** is used. Bipartite graph consists of two independent sets of nodes(**U**,**V**) for which the edges connect node pairs $(u_i\,, v_j)$ where $u_i \in$ **U** and $v_i \in$ **V**. Figure(3.1) shows an example.

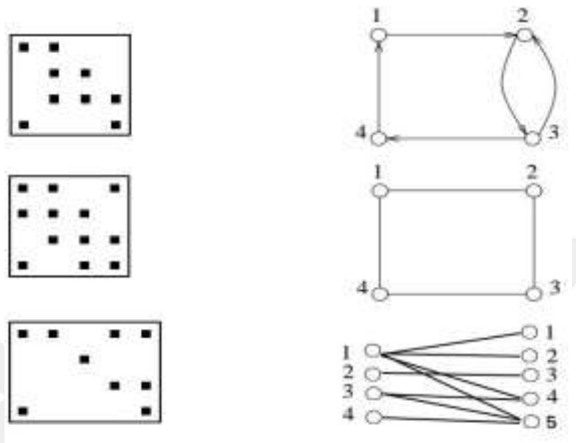


Figure 3.1 : Graph of sparse matrices (above)non-symmetric, (middle)symmetric, (down)bipartite for 4 x 5 matrix.

A common operation for solving sparse linear systems is by reordering rows and columns of the sparse matrix. This is a very important preprocessing step especially for parallel implementation. When the nonzero elements are cluttered near the main diagonal, this makes the variables more independent and thus less communication is required to find unknown values. This structure minimizes the **fill-in** problem of direct methods because of Gaussian elimination. Besides, the block diagonal structure maximizes locality and minimizes the solution time cost.

A **permutation matrix** is a matrix with its rows and/or columns interchanged. In order to interchange the rows of a matrix, we need to premultiply the matrix by a permutation matrix $\boldsymbol{P_r}$. Similarly, to change the columns of the matrix, we need to postmultiply the matrix by the permutation matrix $\boldsymbol{P_c}$. Permuting the rows of the matrix must also change the right hand side of the equation **AX=b** and permuting the columns of the matrix should change the order of the unknowns **X**. Furthermore, if $\boldsymbol{P_c} = \boldsymbol{P_r}^{\boldsymbol{T}}$,the permutation is known as symmetric permutation. Such permutation preserves the diagonal elements and the symmetric pattern of the original matrix. A matrix is called

reducible if its corresponding graph is undirected, otherwise, irreducible. A reducible matrix can be transformed into block upper triangular form using symmetric permutation [14].

3.2 Ordering and Graph Partitioning Techniques

In this section we present the graph partitioning problem and the most popular algorithms for solving them. In general, **a p-way partition** of a graph is a mapping **P : V[1 ... p]** vertices into subsets called partitions $S_1, S_2, S_3 .... , S_p$ such $\cup_i S_i = V$ and $S_i \cap S_j$ = 0. An **edge cut** E_c is a subset of **E** whose vertices lie on different partitions. The edge cut is known as **edge separator** because removing them split the graph into distinct partitions. **Vertex separator** is also possible in which the graph is partitioned along the vertices. It is this vertex separator that is used in the well-known nested dissection algorithm as we will see in section (3.2.4).

Graph partitioning problem is to partition the graph into roughly equal number of partitions with minimum separator size. This problem is known to be **NP-complete** problem [15,16]. However, many algorithms have been developed to give good partitioning. Table (3.1) shows the most common schemes for graph partitioning.

Ordering sparse matrices is mainly related to direct methods which use elimination during LU factorization. As a result, many nonzero elements can be generated. These new nonzero elements are called **fill**. Like that of dense matrices, ordering rows is used to make factorization more numerically stable. However for sparse matrices, there are other objectives : minimize fill-in and maximize parallelism. Finding the ordering that produces the fewest new entries in the LU factorization(the minimum fill-in) has already proven to be **NP-complete** problem [17,18] and thus many approaches are heuristics in nature. So fill-in is inevitable in LU factorization since most ordering algorithm are usually more expensive than the fill-in problem itself. Here we present four main strategies according to [19].

Table 3.1 : List of popular graph partitioning Schemes

Scheme	**Method**	**Run Time**	**Brief Info**
Geometric Techniques (coordinate information)	Coordinate Nested Dis- section (CND)	---	Splits the mesh in half, normal to its longest axis,fast, requires low memory and easy to par- allelize but the subgraphs are of low quality
	Recursive Inertial Bisec- tion (RIB)	---	Splits the mesh in half, normal to principal iner- tial axis of the mass distribution
	Sphare Filling Curve	---	Splits the mesh into *k* parts according to the po- sitions of the centers-of-mass elements along a space-filling curve,fast
	Sphare Cutting Approach	---	uses(a, k)-overlap graph to construct vertex sep- arator
Combinatorial Techniques (adjacency information)	Levelized Nested Dis- section (LND)	---	Numbering vertices in BFS manner until half vertices are numbered(one partition). The un- numbered vertices are in the other partition, sen- sitive ti initial vertex choice
	Kernighan- Lin/Fiduccia-Mattheyses (KL/FM)	---	Partition refinement of sub-optimal partitioning graphs, heuristic, naturally sequential
Spectral Techniques	Recursive Spectral Bi- section	---	Splits vertices according to eigenvector of the second smallest eigenvalue of L_G (Fiedler vec- tor), computationally expensive
	Multilevel Spectral Bi-section(MSB)	---	Uses multilevel approaches to reduce computa- tion , ,well-parallelized
Multilevel Schemes (Coarsening, Partitioning , Refinement	Multilevel Recursive Bi- section	O(|E|logk)[10]	Partitioning phase :A2-waypartition P_m, ver- tices in V_m are split in half, well-parallelized
	Multilevel k-way Parti- tioning	$O(\|E\|)$[10]	k-way multilevel partitioning,well-parallelized

3.2.1 Block traingular matrix ordering

We mentioned in section (3.1) that reducible matrices can be transformed into block triangular form. Such form is shown in equation (3.1).

$$PAQ = \begin{bmatrix} B_{11} & & & \\ B_{21} & B_{22} & & \\ B_{31} & B_{32} & B_{33} & \\ \dots & & \dots & \\ B_{N1} & B_{N2} & B_{N3} & & B_{NN} \end{bmatrix} \tag{3.1}$$

This form is appealing because it minimizes the cost of storage and solution time of the linear system since we can use forward substitution as follows.

$$B_{ii}\ y_i = (Pb)_i - \sum_{j=1}^{i-1} B_{ij}\ y_j \qquad \text{i = 1,2,3, ...} \tag{3.2}$$

The factorization is only done on the diagonal blocks B_{ii} and the off-diagonal blocks are used only for multiplication $B_{ij}\ y_j$. Thus fill-in occurs on the diagonal blocks only. Duff et.al in [14] divides the process of constructing block triangular form into three stages. Here we describe an algorithm with block matrix of size 3,i.e

$$\begin{bmatrix} B_{11} & & \\ B_{21} & B_{22} & \\ B_{31} & B_{32} & B_{33} \end{bmatrix} \tag{3.3}$$

- **Finding Row and Column Singletons**

  The purpose of this stage is to order the matrix into a form similar to (3.1). The idea is that any row singleton is moved to the first diagonal position of B_{11} block matrix. In the next step, the remaining submatrix is traversed for any other row singleton. If any, move it to the second diagonal position of B_{11}. The process continues until no singleton rows remaining. The result is a block lower triangular matrix B_{11}. Similarly, we construct the B_{33} matrix choosing one column singleton at a time from each remaining submatrix. The remaining portion of the matrix constitutes B_{22} block. An example of this algorithm is shown figure (3.2) .

$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & \times & & & \times & & & \\
2 & & \times & & \times & & \times & \\
3 & & \times & \times & & & & \times \\
4 & \times & & & \times & & & \\
5 & & \times & & & \times & & \\
6 & & & & & & & \times \\
7 & & & & & & \times & \times \\
\hline
\end{array}
\quad \text{becomes} \quad
\begin{array}{c|ccccccc|}
 & 7 & 6 & 1 & 4 & 2 & 3 & 5 \\
\hline
6 & \times & & & & & & \\
7 & \times & \times & & & & & \\
1 & & & \times & \times & & & \\
4 & & & \times & \times & & & \\
2 & & \times & & \times & \times & & \\
3 & \times & & & & \times & \times & \\
5 & & & & & \times & & \times \\
\hline
\end{array}$$

Figure 3.2 : Block Triangular matrix ordering through row and column singleton [14].

- **Permute Entries on the Diagonal(Transversal).**

  This stage concerns with placing nonzero entries on the diagonal of B_{22} of equation (3.1) so that all diagonal entries are nonzero at the end of this stage (otherwise the matrix is structurally singular). This problem is known as **assignment problem**[19,20] . The algorithm is based on depth-first search with look-ahead feature by seeking through rows or through columns with one row/column to examine at a time. So for example, to order the matrix into a sequence of columns c_1 , c_2 , c_3 , ... c_j with c_1 having **k** nonzero entries. Starting with the first entry in column **k**, we take its row number to indicate the next column and search through the first off-diagonal entry in each column as a subsequent column(DFS). In each column, we look for an entry in row k or beyond(Look-ahead).

- **Finding the Block Triangular Form by Symmetric Permutation.**

  The purpose of this stage is to find the symmetric permutation that will put the matrix into block lower triangular form. Digraphs are usually used since applying symmetric permutation does not change the digraph of the associated matrix except for relabeling of its nodes. Here we discuss two algorithms for relabeling.

  In **Sargent and Westerberg algorithm**, starting from a random node, we trace a path until we find a node from which the path does not leave. This last sinking node will be labeled first and we then delete all edges connecting to the node (deleting the row and column in the matrix). Continuing this way until no node remains. This process always works in the digraph as long as there are no cycles in the graph. Such cycles are called **strong components**. Cycles are collapsed into a single node called composite node and labeled separately. Figure (3.3) shows an example of the algorithm. Nodes in bold are labeled at this step.

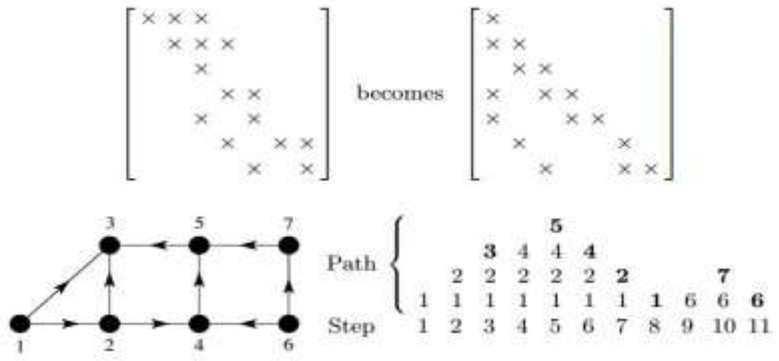


Figure 3.3 : Relabeling a matrix using Sargent and Westerberg algorithm [19].

Tarjan's algorithm is more efficient than the previous algorithm and uses stack for path tracking and node labeling. It avoids the excessive relabeling of the previous algorithm. The algorithm starts with a random selected node and moves through the unvisited edges pushing the node into the stack. If we find an edge connecting a node on top of the stack with a lower node, this will be a closed path and we do not label the nodes. A node is labeled and removed from the path if all its edges are visited. Figure (3.4) shows an example of this method.

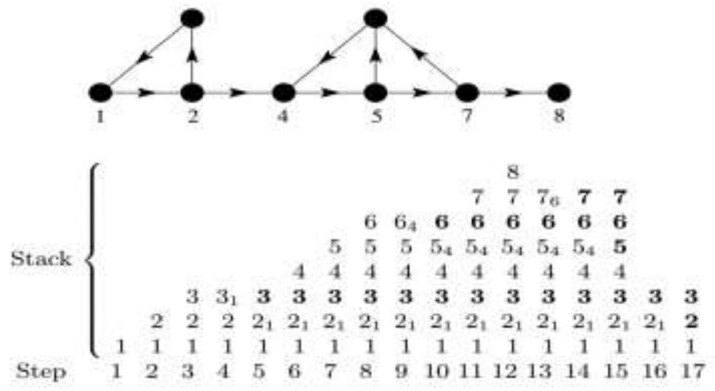


Figure 3.4 : Relabeling a matrix using Tartan's algorithm [19].

3.2.2 Local pivot ordering

These set of algorithms are based on Markowitz criterion [21] which chooses as a pivot the entry a_{ij} with row **i** and column **j** having the lowest number of nonzero elements.

More formally, for **(n-k+1) (n-k+1)** submatrix after applying (k-1) steps of the Gaussian elimination ,select an entry $a_{ij}{}^{k}$ that minimizes:

$$(r_i{}^{(k)} - 1)(c_j{}^{(k)} - 1) \tag{4}$$

Where $r_i{}^{(k)}$ and $c_j{}^{(k)}$ are the number of nonzero entries in row **i** and column **j** respectively. They are local greedy approaches because they select as a pivot the node with minimum degree without regard whether this selection affect the following steps. This is also very expensive since it requires knowledge of the sparsity pattern of the submatrix at each stage. Some variations of this algorithm are follows:

- **Minimum Degree Method**
  This method is applied to symmetric matrices. Thus the objective function of equation(4) is reduced to $min_t r_i{}^{(k)}$. It is called minimum degree because it chooses the node with the smallest degree in the associated graph of the submatrix as the next pivot row. To avoid the high cost of degree update at each step, an approximation is used such as **Column Approximate Minimum Degree (COLAMD)**[22] which proves to perform better than the minimum degree approach and applicable for unsymmetic matrices. A major drawback of this algorithm is its sensitivity to the ties: when more than two nodes have the same minimum degree value; which one to choose. Another variant of minimum degree is **Multiple Minimum Degree(MMD)**.
- **A Priori column ordering**
  The idea is to find a good column permutation for the normal symmetric positive definite matrix $\boldsymbol{N} = \boldsymbol{A^T A}$ and apply it as a column ordering to the zero-free diagonal matrix **A**. The justification is that finding a good column permutation for **N** will drastically reduce fill-in in its Cholesky factorization pattern **R**. Since the pattern of **A** is contained in **N** [20] .i.e if $a_{ij} \neq 0$ then $n_{ij} \neq 0$, such a column permutation will also reduce fill-in in $L_k\ U$ factorization of **A** provided that **N** is sparse and **A** is zero-free diagonal. This method is used in sequential and shared memory solvers.

3.2.3 Band and variable band ordering

Another set of approaches that rely on ordering the matrix such that the elements are cluttered along the main diagonal in a banded or variable band form. Examples are shown in figure (3.5). In this section we introduce some basic terms and the common algorithms for this type of ordering. A matrix has a **bandwidth** 2m + 1 and a **semibandwidth** m if $a_{ij} = 0$ whenever |i - j| > m. A **profile** (also called skyline or envelope) is the number of nonzero entries in the variable banded matrix. It is clear that Gaussian elimination without interchanges does not affect the banded or variable banded structure of the matrix since no fill is created outside the band. However, any zeros within the band will be filled totally. Thus the smaller the bandwidth and the profile the less fill can occur.

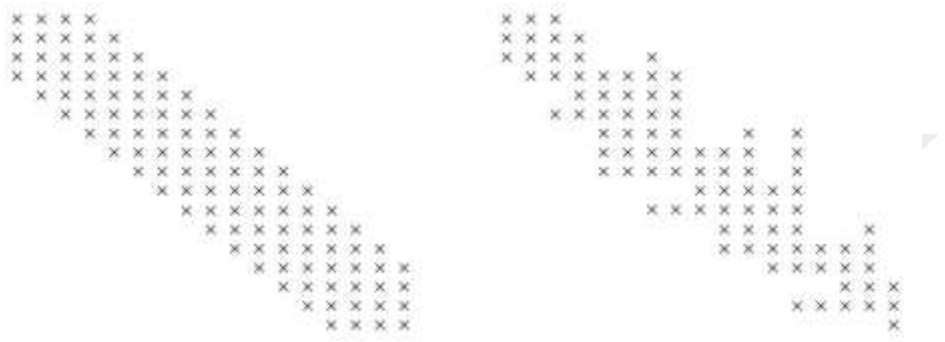


Figure 3.5 : Band and variable band form [19].

An important variant of this method is Cuthill-McKee and Reverse Cuthill-McKee algorithm(CM/RCM). This algorithm constructs a level set of nodes via breadth-first search starting with a node of minimum degree at level set S_1. Level set S_2 consists of all nodes neighboring to the node in S_1. Level set S_3 contains all nodes neighboring to the nodes in S_2 that are not in S_1 and S_2. In general, level set S_i consists of all neighboring nodes in S_{i-1} that are not in S_{i-1} and S_{i-2}. The ordering then starts by traversing all nodes from the lowest level to the highest level as shown in the example of figure(3.6). Reversing this order (RCM) was proven to give better performance when the neighbors of a node at level S_i have occurred in the previous levels or the node has no neighbors at level S_{i+1}.

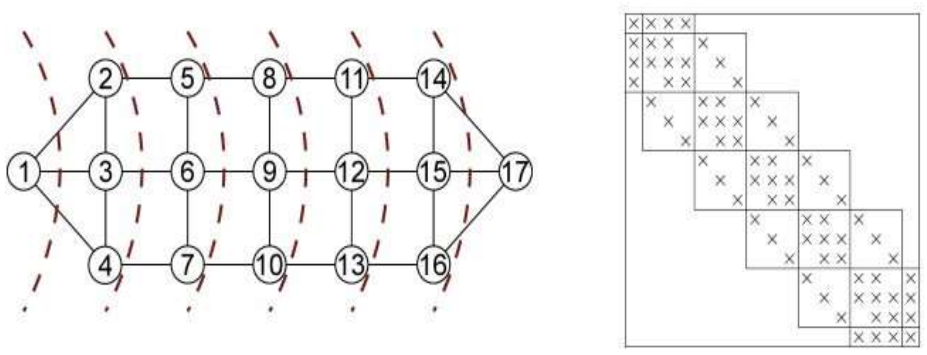


Figure 3.6 : Example of Cuthill-McKee ordering [19].

3.2.4 Dissection methods

These algorithms are based on finding a set of nodes/edges whose removal result in splitting the graph into independent subgraphs. Those nodes/edges are called node/edge separators. Ordering the separators last results in a bordered block diagonal form. An important variant of this algorithm is **the Nested Discetion method**. This is a divide and conquer approach and was originally developed for partitioning regular graphs generated from finite element systems of equations [23] but recently it has been found to do well for large 3D problems. In this method, we split the graph into roughly two equal parts using separators. After getting the first partition, we continue with the subgraphs using the same steps. An example of this ordering is shown in figure (3.7). The efficiency of this algorithm highly depends on the size of the separator and thus works well with problems of regular and planer graphs since they have smaller separator size [24]. There are several methods for finding the separator nodes with the objective to find the smallest possible separator size. One way is by constructing the level set discussed in section (3.2.3) and taking the middle set as the separator set.

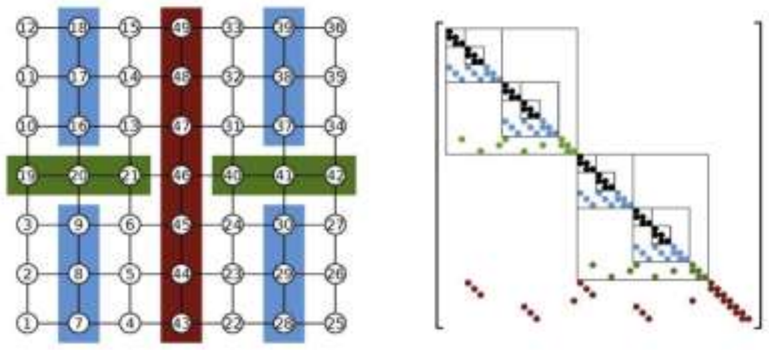


Figure 3.7 : Example of Nested Dissection ordering [8].

3.3 Ordering Unsymmetric Matrices

Unlike symmetric matrices, partitioning unsymmetic matrices can not be done directly using undirected graphs but rather many algorithms borrow techniques from partitioning symmetric matrices. Usually the graph of $\boldsymbol{A^T + A}$, $\boldsymbol{A^T A}$ and $\boldsymbol{A\,A^T}$ are used help partitioning and ordering the unsymmetric matrix **A** because the sparsity structure of these matrices is a superset of the sparsity structure of **A**. Obviously, if **A** is symmetric the structure of $\boldsymbol{A^T + A}$ is the same as the structure of **A** and $\boldsymbol{A^T A}$ and $\boldsymbol{A\,A^T}$ will be the identical. Furthermore, the structure of $\boldsymbol{A^T + A}$ will be nearly symmetric if **A** is so and will be full if **A** is unsymmetirc. Both $\boldsymbol{A^T A}$ and $\boldsymbol{A\,A^T}$ are symmetric positive definite and thus Cholesky factorization can be used. As discussed in section (3.2.2), $\boldsymbol{A^T A}$ is used to find a column permutation Q such that the sparsity pattern in the Cholesky factorization of $\boldsymbol{Q^T\,A^T A\,Q}$ is preserved. For numerical ordering, row permutation is used which does not affect the structure of $\boldsymbol{A^T A}$ since $\boldsymbol{(PA)^T PA} = \boldsymbol{A^T A}$ [19,20]. In this case, **P** is $\boldsymbol{Q^T}$. For convenience, we present the suggested solution steps of (George/Ng '87) for solving sparse linear system **AX = b** in algorithm (1) below where A is large sparse unsymmetric matrix. This is the original paper that proposed Nested Dissection ordering discussed previously.

A different approach for partitioning unsymmetric matrices is by using **Hypergraph**. Hypergraphs are generalizations of graphs in which edges(called hyperedges or nets) can connect arbitrary number of vertices(called pins). In matrices, rows represent vertices and columns represent nets. A set of vertices(rows) is called a part and a net is connected to a part if it has at least one pin in the part. Hypergraphs can partition the unsymmetric matrix into any number of parts[19]. The most popular packages for partitioning unsymmetric matrices using hypergraph is **PaToH** and **Zoltan**.

Algorithm 1 ND-algorithm[20]

```
• Find a permutation matrix Q so that QA has a zero-free diagonal.
• Determine the structure of $B = (QA)^T(QA) = A^T A$.
• Find a symmetric permutation $P_c$ so that ${P_c}^T B P_c$ has a sparse Cholesky
  factor. Denote the Cholesky factorization by ${P_c}^T B P_c = \widehat{R^T} \hat{R}$.
• Determine the structure of Cholesky factor $\hat{R}$ of ${P_c}^T B P_c$, and set up a
  storage scheme that exploits the sparsity of $\hat{R}$ and $\widehat{R^T}$.
• Input the numerical values of A, storing it as ${P_c}^T QA P_c$.
• Compute the LU-decomposition of ${P_c}^T QA P_c$ using Gaussian
  elimination with partial pivoting. Store the triangular factors in the
  storage structure for $\hat{R}$ and $\widehat{R^T}$.
• Solve $({P_c}^T QA P_c){P_c}^T = {P_c}^T Qb$ using the LU-decomposition.
```

3.4 Sparsity Preservation vs Numerical Stability

Preserving sparsity and numerical stability can be conflicting issues in sparse matrices. As we will see in the next chapter, pivoting is used during LU factorization to put large entries on the diagonal and void tiny pivots. However, this can destroy sparsity if the new pivot row, for instance, contains more nonzero elements than the original pivot row and will cause more fill-in in the following steps.

One technique to alleviate this problem is through **threshold pivoting**. Suppose that $\boldsymbol{a_{ii}}$ is the diagonal entry and $\boldsymbol{a_{mi}}$ is the largest entry on a partially factored matrix $\boldsymbol{A}$ up to column $\boldsymbol{i}$. Depending on a threshold value $\boldsymbol{0 < u \leq 1}$, defined by the user, $\boldsymbol{a_{mi}}$ will be chosen as new pivot if $|\boldsymbol{a_{ii}}| < \boldsymbol{u}\, |\boldsymbol{a_{mi}}|$. otherwise no changing is done. When $\boldsymbol{u}$ = 1, this is equivalent to the classical partial pivoting and when $\boldsymbol{u}$ = 0, the diagonal entries on the pivot will be chosen, unless they are zero so pivoting is required [19,25].

For symmetric positive definite matrices, such numerical stability is not a concern because pivots has no growth during Gaussian elimination. A detailed study about this is in [26]. There are other types of matrices which have special numerical stability issues like diagonally dominant matrices, symmetric indefinite matrices which is out of scope of this study.

3.5 Software Packages for Ordering and Graph partitioning

The most important parallel partitioning libraries used are Ptscotch and Parmetis. Table (3.2) shows the most common packages for graph partitioning.

Table 3.2 : Popular Packages for Ordering Matrices

Name	Method	Type
Chaco	Multilevel spectral bisection approaches & unsymmetric & sequential and parallel	sequential / parallel
Jostle	Multilevel k-way partitioning and diffusive load-balancing & unsymmetric & sequential and parallel	sequential / parallel
PARTY	Multilevel k-way partitioning & unsymmetric & sequential and parallel	sequential / parallel
Metis/Parmetis	Multilevel recursive bisection, multilevel k-way, KL/FM refinement	sequential / parallel
Sotch/Ptscotch	Multilevel recursive bisection with KL/FM refinement	sequential / parallel
PaToH	Multilevel hypergraph partitioning	sequential
Zotan	Hypergraph partitioning	sequential
hMeTiS	Hypergraph partitioning	sequential

4. DIRECT METHODS

In this chapter we talk about solution of sparse linear systems using the most important factorization techniques for solving large sparse linear systems. Before proceeding with the discussion of the factorization methods,we define some basic concepts.

Elimination tree is a compact data structure that shows the node dependencies and the order in which the variables must be eliminated. Thus it is related to ordering and Gaussian elimination. A vertex **j** depends on a vertex **i** such that **i** < **j** (written **i** → **j**) iff eliminating element a_{ki} affects the value of a_{kj} for **k** < **i**. More formally, i →j iff ∃ k ∈ [i+1,n] such that $L_{ki}U_{ij} \neq 0$ [27]. If there is such a relation (**i** → ***j***) , we say **j** is an ancestor of **i** in the elimination tree. Figure (4.1) shows an example of elimination tree.

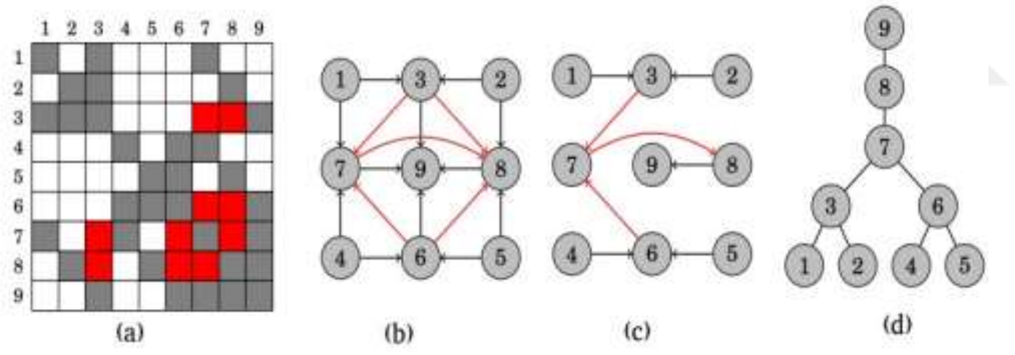


Figure 4.1 : (a)matrix ordered by ND(red shows fill-in) (b) directed filled graph (c) transitive reduction (d) elimination tree[27].

Directed filled graph is the graph representation of the matrix and, shows vertex dependencies(including fill-in). Because the dependency relation is not symmetric since (i → j) can occur only if i < j, the filled graph must be a directed graph. Figure (4.1b) shows the directed filled graph of matrix shown in (4.1a) with red arrows showing the fill-in that can occur during elimination. Figure (4.1c) shows the transitive reduction graph after eliminating dependencies. If the matrix is symmetric, the undirected variant of this transitive reduction graph is a spanning tree (**elimination**

tree)[27, 28]. If the matrix is unsymmetric, approaches of section (3.3) are used to symmetrize the matrix. Figure(4.1d) shows the elimination tree of this example.

In the elimination tree, two or more nodes(columns) can be grouped together if they have the same sparsity structure. The resulting grouped node is called **supernode** and the process is known as **amalgamation**. The solvers that use this approach are **Supernodal method** solvers. For example, in figure (4.1), nodes 7,8,9 have the same sparsity structure and thus can be combined to form a supernode. Among the common Solvers that use this method are Superlu-dist and Pastix.

Another important tree structure is the **assembly tree** which assembles the contributions of the variables in the form of a dense block matrix called **contribution block** of the variable. For example, assembly tree is used in **frontal method** which was originally developed for solving finite element problems in limited memory systems. Figure(4.2) shows an example.

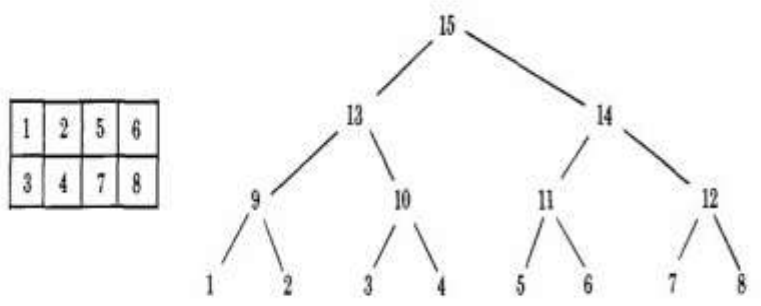


Figure 4.2 : The proposed steganography method for extraction[29].

In this approach, each finite element variable form a block dense matrix called **frontal matrix** and the whole system is solved by partial factorization and elimination of matrices of the form.

$$A = \sum_{l} B^{(l)} \qquad (5)$$

That is, when applying Gaussian elimination, there is no need to wait for all the assembly steps of equation(5) to complete, i.e assembly and factorization can be done simultaneously. The variable can be chosen as a pivot only if it is fully summed i.e, no further contribution to come to the variable. In other words, a column is chosen for elimination only when all its descendants have been eliminated. In Gaussian elimination, this corresponds to passing the Schur complement to the parent in the elimination tree [29,30].

Depending on the properties of the matrix, we distinguish the following types of matrices [31]:

- n x n symmetric positive definite matrix for which Cholesky A = $L\,L^T$ factorization is used where **L** is a lower triangular matrix. A = LDL^T is also possible where **D** is a diagonal matrix and **L** is a unit lower triangular matrix. This later formula voids using square roots. As duscussed in section (3.4), such matrices simplify pivoting strategies..
- n x n unsymmetric matrix for which A = L U factorization is used where **L** is a unit lower triangular matrix and **U** is an upper triangular matrix.
- n x n symmetric indefinite matrix for which A = LDL^T factorization is used where **D** is a block diagonal matrix. Diagonal pivoting is required.
- m x n rectangular matrix where m $\neq$ n for which A = QR factorization is used where **Q** is n x n orthogonal matrix and **R** upper trapezoidal matrix. LU factorization is also possible where **L** is n x n lower triangular matrix and **U** is m x n upper trapezoidal matrix since $U_{ij} = 0$ when i > j.

In int following sections, we will investigate these factorization methods in detail.

4.1 Cholesky Factorization

Due to its importance, Cholesky factorization has gained a lot of attention from research community. The major steps for solving the linear system where the matrix is symmetric positive definite are the following:

- **Ordering**. This type of matrices simplify ordering since the matrix is numerically stable. The purpose of ordering is to find a permutation matrix **P** such that P^TAP has small fill-in.
- **Symbolic factorization**. The nonzero structure of **L** can be determined independently of the numerical factorization. Elimination tree is the result of this step.
- **Numerical factorization.** Applying Cholesky factorization to find L.

- **Triangular Solution**. Forward substitution to find y in **Ly = b** and back substitution to find x in $\boldsymbol{L^T}$**x = y**

4.2 LU Factorization

The most common method for matrix factorization is LU Factorization where L is lower triangular and U is upper triangular. In general, there are four major steps for direct solution of sparse linear system Ax=b using LU factorization [24,33,34]:

- **Ordering**. Reorder rows and columns to reduce fill-in as discussed in chapter (3). Our experimental results show that row permutation can enhance the solve step tıme by orders of magnitude.

- **Symbolic factorization**. As we have seen previously, from ordering, we can predict the upper bound of fill-in that can occur during numerical factorization. Thus a static data structure can be accommodated for **L** and **U** beforehand. This added to the efficiency because dynamic memory allocation for the fill-in takes time. Besides, this allows us to determine which compression scheme to use(CRS, CSC ...etc). This process is known as **Symbolic factorization**. For example, steps (3-6) in algorithm (1) represents one way of symbolic factorization. Another approach is the symbolic analysis of the symmetric matrix $A + A^T$. However, Unlike $A^T A$, row permutation affects the ordering found and thus symbolic analysis will not be so good.

  In many cases, row permutation $C = AQ + Q^T A^T$ is performed to get large diagonal entries and thus reduce the number of pivots that are delayed in this way[35]. Such a permutation matrix **Q** can be found using **maximum weighted bipartite matching** algorithm. Developers of This way of prepivoting large elements to the diagonal is called **Static Pivoting**. To further ensure stability, tiny pivots are replaced with a small perturbation $\sqrt{\epsilon}$||A||, where ϵ is machine precision) and iterative refinement is used at the end to get more accurate results.

- **Numeric factorization**. To compute **L** and **U** factors and store results in the data structure assigned in the previous step. This is the most time consuming part of the solution process. For unsymmetric matrices, numerical and symbolic factorization interleave.

To construct LU factors, **Gaussian elimination** is used. For clarity, we illustrate Gaussian elimination with 3x3 linear system example taken from[19] shown in equation (6).

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{6}$$

The first step is to eliminate a_{21} and a_{31} by multiplying the first equation first by l_{21} = a_{21}/a_{11} and subtract it from the second equation and then multiply it by a_{31} = a_{31}/a_{11} and subtract it from the third equation. The equivalent system is:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & a^{(2)}{}_{22} & a^{(2)}{}_{23} \\ l_{31} & a^{(2)}{}_{32} & a^{(2)}{}_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b^{(2)}{}_2 \\ b^{(2)}{}_3 \end{bmatrix} \tag{7}$$

where

$a^{(2)}{}_{22} = a_{22} - l_{21}a_{12}$ $\quad a^{(2)}{}_{23} = a_{23} - l_{21}a_{13}$ $\quad b^{(2)}{}_2 = b_2 - l_{21}b_1$

$a^{(2)}{}_{32} = a_{32} - l_{31}a_{12}$ $\quad a^{(2)}{}_{33} = a_{33} - l_{31}a_{13}$ $\quad b^{(2)}{}_3 = b_3 - l_{31}b_1$

Finally, we have to eliminate $a^{(2)}{}_{32}$ by multiplying the new second equation by l_{32} = $a^{(2)}{}_{32} / a^{(2)}{}_{22}$ and subtract it from the third equation.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & a^{(3)}{}_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b^{(2)}{}_2 \\ b^{(3)}{}_3 \end{bmatrix} \tag{8}$$

where

$$a^{(3)}{}_{33} = a^2{}_{33} - l_{32}a^{(2)}{}_{23} \quad , \quad b^{(3)}{}_3 = b^{(2)}{}_3 - l_{32}b^{(2)}{}_2$$

The entry $a^{(k)}{}_{ii}$ (called the **pivot**) should be nonzero for any iteration step **k** and if its value is too small, it will cause instability in the elimination process because the updated element will be very large. This problem propagates in the following steps so it is crucial to choose the pivot element. The factors l_{ik} are called the **multipliers** and they constitute the unit lower triangular matrix **L** and the final matrix of **A** of equation

(8) constitutes the upper triangular matrix **U**. Thus the elimination process can be summarized in the following important equation:

$$a^{(k+1)}{}_{ij} = a^{(k)}{}_{ik} - l_{ik}a^{(k)}{}_{kj} \qquad \text{i,j} > 0 \qquad (9)$$

where $l_{ik} = a^{(k)}{}_{ik} / = a^{(k)}{}_{kk}$.

This way of Gaussian elimination is also called **Right-looking Gaussian Elimination** because at each elimination step, we always use entries of the right of the pivot and modify entries at the lower right part of the submatrix. It is also called **submatrix-based Gaussian elimination** for obvious reason. It is also called "data-driven" , "fan-out" or "eager" approach.

A variant of Gaussian elimination is to delay modifying a_{ij} until column **j** is pivotal. Using this method with the above example,we eliminate the a_{21} and a_{31} using the same method and use this information to update the second column as follows:

$$\begin{bmatrix} u_{11} & a^{(2)}{}_{12} & a_{13} \\ l_{21} & a^{(2)}{}_{22} & a_{23} \\ l_{31} & a^{(2)}{}_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad (10)$$

where

$$u_{12} = a^{(2)}{}_{12} = a_{12} \quad , \quad a^{(2)}{}_{22} = a_{22} - l_{21}u_{12} \quad , \quad a^{(2)}{}_{32} = a_{32} - l_{31}u_{12}$$

In the next step, we modify the third column,

$$\begin{bmatrix} u_{11} & u_{12} & a^{(3)}{}_{13} \\ l_{21} & u_{22} & a^{(3)}{}_{23} \\ l_{31} & l_{32} & a^{(3)}{}_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad (11)$$

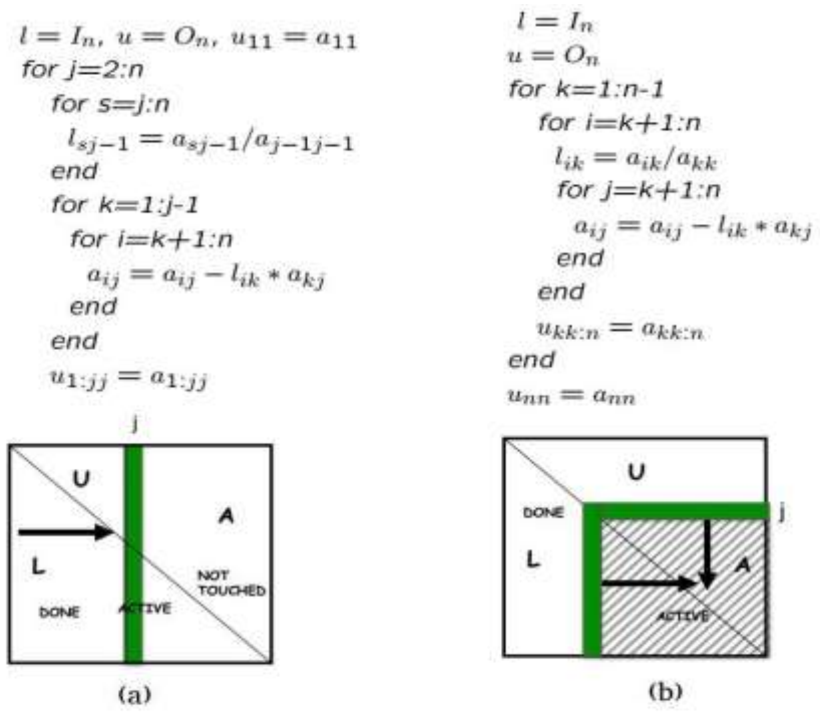
where

$$a^{(3)}{}_{13} = u_{13} = a_{13} \quad a^{(3)}{}_{23} = a_{23} - l_{21}u_{13} \quad a^{(3)}{}_{33} = a_{32} - l_{31}u_{13} - l_{32}u_{23}$$

which yields the same results as the previous example.Thus the elimination equation will be [19]:

$$a^{(j)}{}_{ij} = a_{ij} - \sum_{k=1,k<j}^{j-1} l_{ik}\, u_{kj} \qquad (12)$$

This way of Gaussian elimination is called **Left-looking Gaussian elimination** because, as shown in the example, while modifying the current column, we always use the information on the left of the pivotal column. It is also called **Column-based Gaussian elimination** for obvious reason. It is also called "demand-driven" , "fan-in" or "lazy" approach. A C implemetation code of this these two varaints for factorizing dense matrices is shown in figure(4.3).



```
$l = I_n,\ u = O_n,\ u_{11} = a_{11}$
for j=2:n
   for s=j:n
     $l_{sj-1} = a_{sj-1}/a_{j-1j-1}$
   end
   for k=1:j-1
     for i=k+1:n
       $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
     end
   end
   $u_{1:jj} = a_{1:jj}$
```

```
$l = I_n$
$u = O_n$
for k=1:n-1
   for i=k+1:n
     $l_{ik} = a_{ik}/a_{kk}$
     for j=k+1:n
       $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
     end
   end
   $u_{kk:n} = a_{kk:n}$
end
$u_{nn} = a_{nn}$
```

Figure 4.3 : (a) left approach (b) right approach of Gaussian elimination[8,36].

An important variant of the right-looking Gaussian elimination is the **Multifrontal method** which makes use of the concepts of elimination and assembly trees discussed earlier. In the multifrontal method, each node of the assembly tree is associated with a matrix called **frontal matrix** that shows the elimination contribution of the node. For example, eliminating node **1** of figure(4.2) contributes to variables **3** and **7**($1 \rightarrow 3$, $1 \rightarrow 7$). This contribution is represented by the 2×2 Schur complement(partial factorization) of the frontal matrix F_1 and is passed to the parent(node **3**) in this case). Similarly, the elimination of node **2** contributes to the variables **3** and **8** and results in 2×2 Schur complement that is passed to node **3**. Now, when eliminating variable **3**, we should first assemble the contributions of all its children and add them to the original contribution of node **3**. We do the same thing with the other variables in the elimination tree. Figure (4.4) shows the assembly tree corresponding to the elimination tree of figure(4.2)

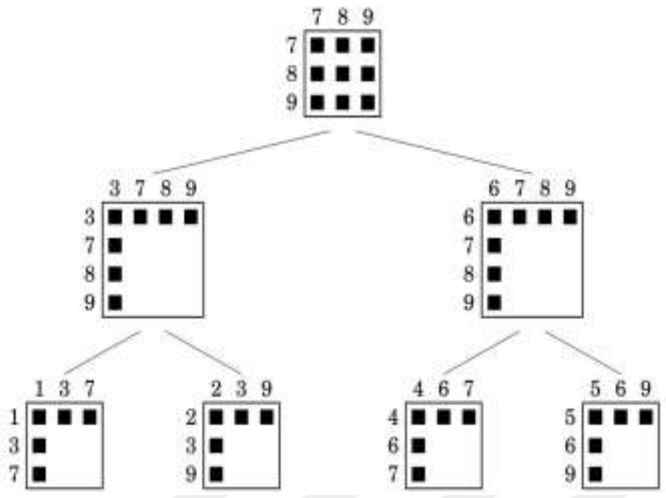


Figure 4.4 : assembly tree of the elimination tree of figure(4.3) with the supernode combining 7,8,9 [27]

- **Solution.** solve the system using forward and backward substitution. Once the LU factorization is obtained, the linear system solution consists of two steps:
    1. The forward substitution that solves the triangular system Ly = b;
    2. The backward substitution that solves Ux = y.

  The solution of successive linear systems using the same matrix but with different right-hand sides, often arising in practice, is then relatively cheap .

5. ITERATİVE METHODS AND PRECONDİTİONERS

Iterative methods have the advantage of using less memory and being easier to parallelize than direct methods. The main problem is that rate of convergence depends on the properties of the matrix. In this section, we discuss the basic iterative methods and preconditioners.

5.1 Basic Iterative Methods

These methods are also called stationary (fixed point) methods. They are based on the relaxation of the coordinates. We start with an initial approximation values of the solution and modify them through successive iterations until convergence. Basically these methods are not used by their own because they are not so efficient and the convergence is never guaranteed for all types of matrices. However, variations are used as preconditioners or combined with other methods. Table (5.1) shows the basic iterative methods along with the vectorization form of each method where **D** is the diagonal matrix with nonzero entries in the diagonal and **-F** and **-E** are strict lower and upper triangular matrices respectively such that:

$$A = D - L - U \tag{5.1}$$

Jacobi iteration method updates approximate solution locally at the end of each iteration. Thus it is easily parallelizable but the convergence is very slow. Besides, the convergence is only guaranteed if the matrix is diagonally dominant.

Gauss-Seidel iteration method Updates approximate solution on the same vector immediately after the new component is available. Thus it is faster that Jacobi and more economical in terms of memory but difficult to parallelize.

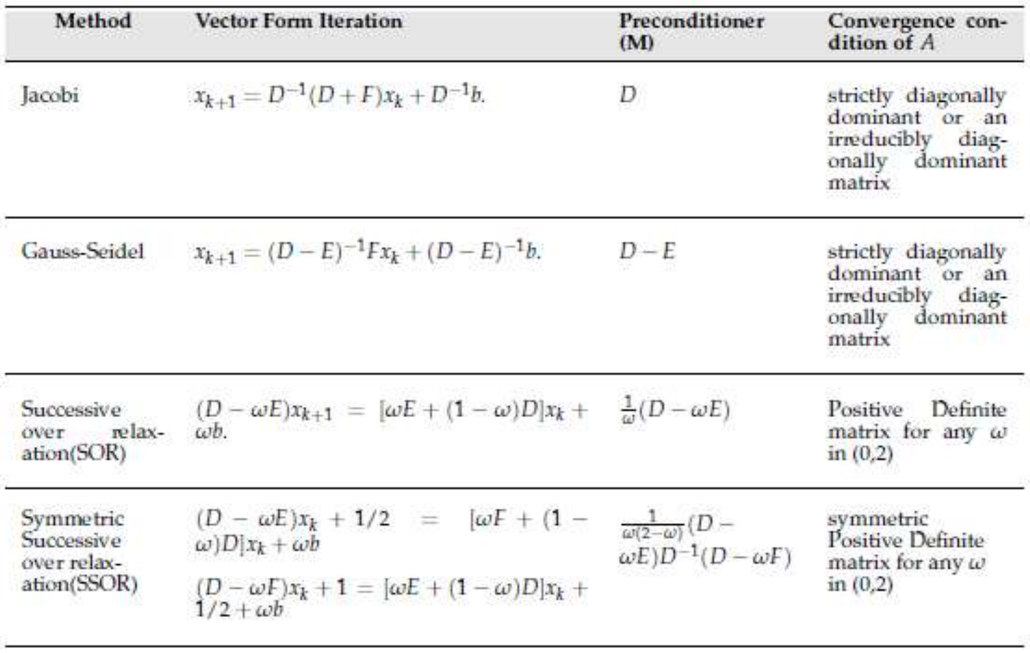
Successive over relaxation (**SOR**) is faster than the previous approaches when an appropriate value of $\boldsymbol{\omega}$ is chosen. Here, $\boldsymbol{\omega}$ is positive nonzero value called the **relaxation factor**. A variant of SOR is symmetric successive over relaxation (**SSOR**) used a preconditioner for non-stationary methods.

Most of the iterative techniques converge faster with **preconditioners**. The role of the preconditioners is to enhance the spectrum and thus the convergence characteristics of the coefficient matrix. Hence a preconditioning matrix **M** is multiplied by the coefficient matrix such that

$$M^{-1}Ax = M^{-1}b \tag{5.2}$$

There are block extensions of these algorithms which increase the locality and speed-up.

Table 5.1: List of Basic Iterative Schemes.



Method	Vector Form Iteration	Preconditioner (M)	Convergence condition of A
Jacobi	$x_{k+1} = D^{-1}(D+F)x_k + D^{-1}b.$	D	strictly diagonally dominant or an irreducibly diagonally dominant matrix
Gauss-Seidel	$x_{k+1} = (D-E)^{-1}Fx_k + (D-E)^{-1}b.$	$D-E$	strictly diagonally dominant or an irreducibly diagonally dominant matrix
Successive over relaxation(SOR)	$(D-\omega E)x_{k+1} = [\omega E + (1-\omega)D]x_k + \omega b.$	$\frac{1}{\omega}(D-\omega E)$	Positive Definite matrix for any ω in (0,2)
Symmetric Successive over relaxation(SSOR)	$(D-\omega E)x_k + 1/2 = [\omega F + (1-\omega)D]x_k + \omega b$ $(D-\omega F)x_k + 1 = [\omega E + (1-\omega)D]x_k + 1/2 + \omega b$	$\frac{1}{\omega(2-\omega)}(D-\omega E)D^{-1}(D-\omega F)$	symmetric Positive Definite matrix for any ω in (0,2)

5.2 Krylov Subspace Methods

Krylov subspace methods are the non-stationary iterative methods and the most important iterative techniques. Krylov subspace methods solve the linear system of equations **Ax = b** by extracting an approximate solution x_m from the affine subspace $x_0 + \kappa_m$ of dimension m by imposing the Petrov-Galerkin condition

$$x_m \in x_0 + K \qquad \text{such that} \qquad b - Ax_m \perp L_m$$

where $\boldsymbol{L_m}$ is a subspace of dimension m and r_0 = b - Ax_0 for some initial guess of solution $\boldsymbol{x_0}$ [14]. In other words, the approximate solution can be defined as

$$\widehat{x} = x_0 + \delta \qquad\qquad \delta \in K$$

$$(r_0 - A\delta, W) - 0 \qquad \forall\, w \in L$$

Figure (5.1) depicts the orthogonality condition of the krylov subspace method.

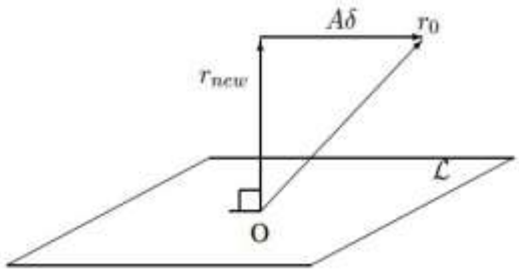


Figure 5.1 : Orthogonality condition of the Krylov subspace[14].

5.2.1 Generalized minimum residual method (GMRES)

GMRES is a projection method where **K** = $\mathbf{K}_m$ and L = A$\mathbf{K}_m$. This method is based on minimizing the residual norm over a Krylov subspace at each iteration by computing a sequence of orthogonal vectors. Algorithm(2) shows GMRES with householder orthogonalization scheme. GMRES converges in at most **n** steps. however, **n** might be large and the memory required for storing the orthonormal bases and the computational cost of the orthogonalization scheme is getting larger. To cope with these problems, GMRES is restarted after **m** iterations if the desired convergence is not achieved with $\boldsymbol{x_0} = \boldsymbol{x_m}$. This variant of GMRES is know as restarted GMRES(m) with projection size **m**. Another important variant of GMRES is Flexible GMRES (FGMRES) in which a right preconditioner is changing at each step. The cost of this flexibility is that an extra memory is required to store a set of vectors $\{\boldsymbol{Z_j}\}_{\boldsymbol{j=1\ldots m}}$ [14].

Algorithm 2 GMRES with Householder orthogonalization [9]

```
procedure GMRES
    Compute $r_0$ = b - A$x_0$, z := $r_0$.
    For j = 1,. . . ,m,m + 1
       Compute the Householder unit vector $w_j$ such that
          $(wj)_i$ = 0,i = 1,... ,j - 1 and
          $(Pjz)_i$ = 0,i = j + 1,... ,n where $P_j$ = I - 2$w_j w_j^T$ ;
       $h_{j-1}$ := $P_j$z ;
       If j = 1 then let $\beta := e_1^T\ h_0$.
       v := $P_1 P_2 \ldots P_j e_j$.
       If ≤ m then compute z := $P_j\ P_{j-1} \ldots P_1$ Av.
    EndFor
    Define $\bar{H}_m$ = the (m + 1) x m upper part of the matrix [$h_1$ ... ,$h_m$].
    Compute ym = $Argmin_y||\beta e_1 - \bar{H}_m||_2$ . Let $y_m = (\eta_1,\eta_2,\ldots , \eta_m)^T$ .
    z := 0
     For j = m,m - 1,... , 1 :
       z := $P_j$ ($\eta_j e_j$ + z),
    EndFor
    Compute $x_m = x_0$ + z
```

5.2.2 Conjugate gradient method (CG)

This method works fine with symmetric positive definite matrices. We approximate the solution by minimizing a quadratic functional of the form.

$$F(x) = \frac{1}{2}\, x^T A\, x - \, x^T\, b \tag{5.3}$$

by taking the orthogonal(conjugate) gradients of equation (5.3). They are also residuals of the iterates. Variants of this method for symmetric but not positive definite are minimal residual (MINRES) and symmetric LQ (SYMMLQ). BiConjugate Gradient(BiCG) generates two CG, one is based on the original coefficient matrix A and the other on $\boldsymbol{A^T}$ and are made orthogonal with each other "bi-orthogonal". This method can work with unsymmetric matrices but they require multiplication with the A and $\boldsymbol{A^T}$ at each iteration [38].

5.3 Preconditioning Methods

A preconditioner is an operator **M** that transforms the original matrix **A** into another matrix such that the new matrix is faster to solve. There are many constraints on the preconditioners. A good preconditioner should be inexpensive to compute and store. In the parallel environment, there are two more constraints; parallelisibility and scalability; the communication should be minimized during preconditioner setup and performance should be enhances with increasing number of processors.

From application point of view, there are two broad classes of preconditioners; problem-specific preconditioners and general-purpose algebraic preconditioners. The former can give optimal solution for specific type of problems but require a complete knowledge of the problem and usually sensitive to the input parameters. The second type is not optimal for specific problems but gives reasonable solution in many cases. The later type use only information contained in the coefficient matrix and they are easy to apply [40].

If preconditioner **M** is a non-singular matrix that approximates $\boldsymbol{A^{-1}}$, then the left preconditioner is defined as:

$$MAx = Mb$$

Right preconditioner is also possible:

$$AMt = b$$

Once the **t** vector is obtained, we can get the solution vector by x=Mt.

6. HYBRID SOLVERS

As mentioned in the previous sections, direct and iterative methods are the most popular algorithms for solving sparse linear systems. Direct methods can give high accurate results($\approx 10^{-15}$) and are more suitable for small(by small we mean several millions of equations) problems because of the memory consumption they require(for example, they can not solve large sparse 3D problems which contain hundereds of millions of equations). Besides, They have limited parallel scalability. Preconditioned iterative methods, on the other hand, are more robust, require less memory and easier to parallelize. However, they are problem dependent and can converge faster with good preconditioning methods.

In an effort to find the best algorithm for solving large sparse linear systems efficiently, various hybrid methods have been developed using both direct and iterative techniques. In general, there are five major steps these hybrid solvers follow to solve the linear systems:

Step 1: Algebraic domain decomposition

Step 2: Factorization

Step 3: Preconditioning

Step 4: Solve

All hybrid solvers discussed here are for solving linear systems of the form:

$$Ax = b \tag{6.1}$$

where **A** is a large sparse general purpose matrix, b is a dense vector and x is the vector of unknowns to be found.

6.1 PDSLin

The first Hybrid solver we are going to discuss is called PDSLin (Parallel Domain decomposition Schur complement based LINear solver) developed by Ichitaro

Yamazaki and X. Sherry Li at Lawrence Berkeley National Laboratory(LBNL). It can be downloaded from here[40]. In this solver, the global system is first reordered using any parallel ordering algorithm like Pt-scotch[41] or Parmetis[42]. Next, the system is divided into a set of subdomains that are only connected through separators as shown in figure(6.1).

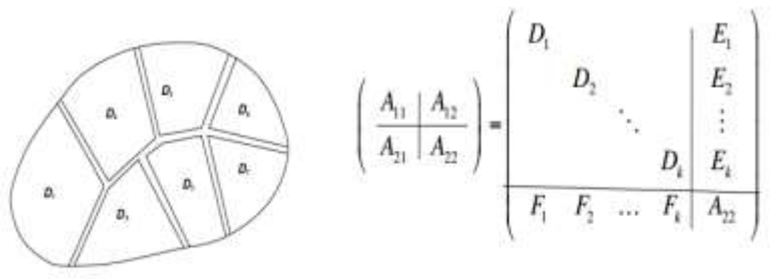


Figure 6.1 : Domain Decomposition of PDSLin[43].

$\boldsymbol{A_{11}}$ is diagonal matrix and always has a block diagonal structure because vertices in each subdomain are either connected to other vertices in the same subdomain or their interfaces. $\boldsymbol{A_{22}}$ are the separators and $\boldsymbol{A_{12}}$ and $\boldsymbol{A_{21}}$ are the interfaces between them. The internal unknowns are then eliminated to form the Schur complement for each subdomain [43]:

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma\mathrm{I}} & A_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ b_\Gamma \end{pmatrix} \qquad (6.2)$$

$$\begin{pmatrix} A_{II} & A_{I\Gamma} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ \hat{b}_\Gamma \end{pmatrix} \qquad (6.3)$$

Thus S and $\boldsymbol{\hat{b}_\Gamma}$ can be defined as,

$$S = A_{\Gamma\Gamma} - A_{\Gamma\mathrm{I}}\, A^{-1}{}_{\mathrm{II}}\, A_{I\Gamma} \qquad (6.4)$$

$$\hat{b}_\Gamma = b_\Gamma - A_{\Gamma\mathrm{I}}\, A^{-1}{}_{\mathrm{II}}\, b_I \qquad (6.5)$$

Matrix $\boldsymbol{A^{-1}{}_{\mathrm{II}}}$ can be factorized using either superlu-dist[44] or MUMPS[3] for each subdomain. The approximate $\tilde{S}$ is used as a preconditioner as follows:

$$\begin{aligned} \mathrm{S} \quad &= \quad A_{\Gamma\Gamma} - (A_{\Gamma\mathrm{I}}\, U^{-1}{}_{\mathrm{II}})(\, L^{-1}{}_{\mathrm{II}} A_{\mathrm{I}\Gamma}\,) \\ &\approx \quad A_{\Gamma\Gamma} - \tilde{G}\, \widetilde{W} \end{aligned}$$

$\approx \quad A_{\Gamma\Gamma} - \tilde{T}$ (global approximation of S as a preconditioner).

$\approx \quad \tilde{S} \quad , \; M = \tilde{S}^{-1}$

Because most of the fill occur in the Schur complement, matrices $\tilde{\boldsymbol{G}}$, $\widetilde{\boldsymbol{W}}$, $\tilde{\boldsymbol{T}}$ and $\tilde{\boldsymbol{S}}$ are approximated by a predefined dropping threshold values in order to enforce the sparsity of the resulting matrices and thus reduce matrix-matrix multiplication and memory consumption. Furthermore, if A is SPD, S inherits this property and thus Conjugate Gradient method can be employed [14]. Another advantage of this method is that Schur complement need not be defined explicitly i.e no need to reserve memory space for the whole Schur matrix and thus multiplication can be done on the fly because this Schur matrix is used only once. S is solved using Krylov subspace method of PETSc[43,45].The unknowns on the interface $\boldsymbol{x}_{\boldsymbol{\Gamma}}$ are then found as:

$$x_\Gamma = \quad S^{-1}\hat{b}_\Gamma = S^{-1}\,(b_\Gamma - \, A_{\Gamma\mathrm{I}}\, A^{-1}{}_{\mathrm{II}}\, b_I)$$

The above multiplication is matrix vector multiplication. The next step is to solve interior unknowns in parallel using already factored subdomains:

$$x_I = \quad A^{-1}{}_{\mathrm{II}}\,(b_I - \, A_{\mathrm{I}\Gamma}\, x_\Gamma)$$

Our experimental results show that choosing appropriate values of sparsifying tolerances can drastically reduce the preconditioning time . However, this also can increase the number of iterations and thus the solution time especially in PDSLin.

To tackle the problem of large scale systems, PDSLin uses two levels of parallelism: in first level, the internal domains are factored concurrently and independently, in the second level, each internal subdomain is assigned to different processors. This ensures constant number of subdomains, Schur size and convergence rate regardless of the number of processors used. See figure(6.2).

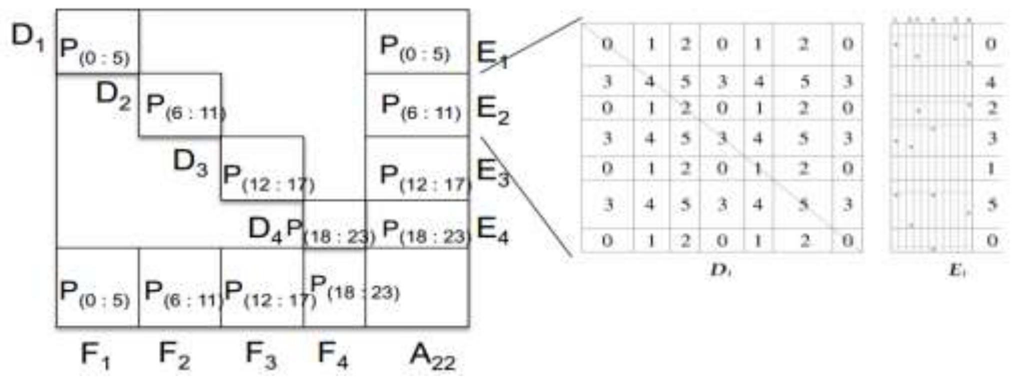


Figure 6.2 : parallel subdomain calculation of PDSLin[43].

To summarize the main solution steps and the options we used for using the two level parallelism in PDSLin :

- Read and scatter the matrix among processors
- .matrix ordering and partitioning. We used Parmetis for subdomain extraction. Some refinement algorithms are used here to load balance subdomains and interfaces.
- Using input options for factorizing internal subdomains, we used superlu-dist with $\boldsymbol{n_{gl}}$ processors to factorize each subdomain concurrently.
- Computation of an approximate schur complement. Using sparsifying tolerances defined in the input file, equation (6.4) is calculated by doing approximate matrix-matrix multiplication on $(\boldsymbol{A_{\Gamma I}}\, \boldsymbol{U^{-1}}_{\mathbf{II}})$ (called G matrix) and ($\boldsymbol{L^{-1}}_{\mathbf{II}}\boldsymbol{A_{I\Gamma}}$) (called W matrix). The result Schur matrix is further preprocessed and sparified.
- This sparsified Schur matrix is then factorized(exact LU factorization) using another instance of superlu-dist and used as a precoditioner for the Schur matrix itself. Krylov method(GMRES, FGMRES or BiCGstab) of PETSc is used to solve the preconditioned Schur matrix. At this stage the solution vector of the interface is found.

- Using solution vector of the interface and the already factored subdomains, triangular solve is used to find the unknowns corresponding to the internal subdomain variables.

6.2 Maphys

Maphys solver [46] is similar to PDSLin. This solver first partitions the global matrix into 2×2 block matrices using non-overlapping graph partitioner and then solves the generated Schur complement. The internal subdomains $\boldsymbol{A_{II}}$ are factored using sparse direct solver either Pastix [47] or MUMPS[3]. To solve the preconditioned Schur matrix, we can use CG, GMRES , FGMRES. The partitioner used here is a sequential one like Metis[42] or Scotch[41] and they consider parallel partitioners as future work. The preconditioning criterion is also different from that of PDSLin. Each local subdomain $\boldsymbol{\Omega_i}$ can be represented as a local matrix $\boldsymbol{A_i}$:

$$A_i = \begin{pmatrix} A_{I_iI_i} & A_{I_i\Gamma_i} \\ A_{\Gamma_iI_i} & A_{\Gamma_i\Gamma_i} \end{pmatrix} \qquad (6.6)$$

Thus the Schur matrix of equation (6.4) can be defined as:

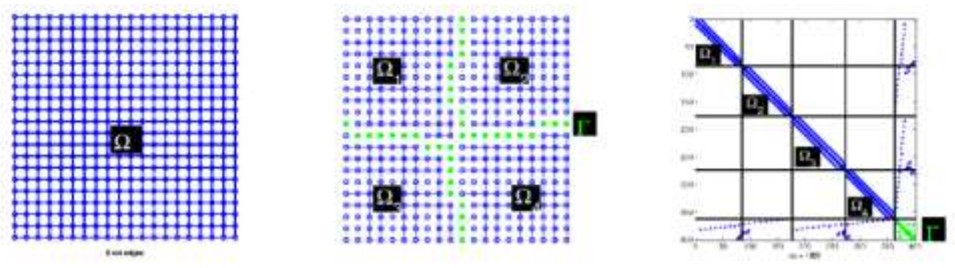
$$S = \sum_{i=1}^{n} R^T{}_{\Gamma\mathrm{i}} \, S_i \, R_{\Gamma i} \qquad (6.7)$$

where

$$S_i = A_{\Gamma_i\Gamma_i} - A_{I_i\Gamma_i} A^{-1}{}_{I_iI_i} A_{\Gamma_iI_i} \qquad (6.8)$$

Since the interior unknowns are no longer considered, a new restriction $\boldsymbol{R_{\Gamma\mathrm{i}}}$ should be devised because of the non-overlapping between the neighboring subdomains at the interfaces($\boldsymbol{\Gamma_i} \cap \boldsymbol{\Gamma_j} = \emptyset$). Thus $\boldsymbol{R_{\Gamma\mathrm{i}}}$ is $\boldsymbol{\Gamma} \rightarrow \boldsymbol{\Gamma_i}$ be the canonical point-wise restriction which maps full vectors defined on $\boldsymbol{\Gamma}$ into vectors defined on $\boldsymbol{\Gamma_i}$ [45].

Equation (6.7) is the global Schur complement obtained by summing the contributions over the subgraphs, and equation (6.8) is the local Schur complement corresponding to each local subdomain $\boldsymbol{\Omega_i}$ of the equation (6.6).



(a) Graph representation. (b) Domain decomposition. (c) Block reordered matrix.

Figure 6.3 : Domain decomposition of Maphys[48].

Figure (6.3) is similar to figure (6.1) but we elaborate on the subdomains because the preconditioner relies on the local assembled Schur complement rather than global Schur complement of PDSLin. The assembled local Schur complement is constructed from local Schur complement by assembling their local blocks. Each box in matrix **M** corresponds to an assembled local Schur complement $\bar{\boldsymbol{S}}_{\boldsymbol{i}}$ as depicted in figure (6.3). This preconditioner is known as **Algebraic Additive Schwarz Preconditioner**. Such preconditioner is similar to Neumann- Neumann preconditioner but with the difference is that it is SPD if the original matrix is SPD which is not always true in Neumann-Neumann preconditioner[10].

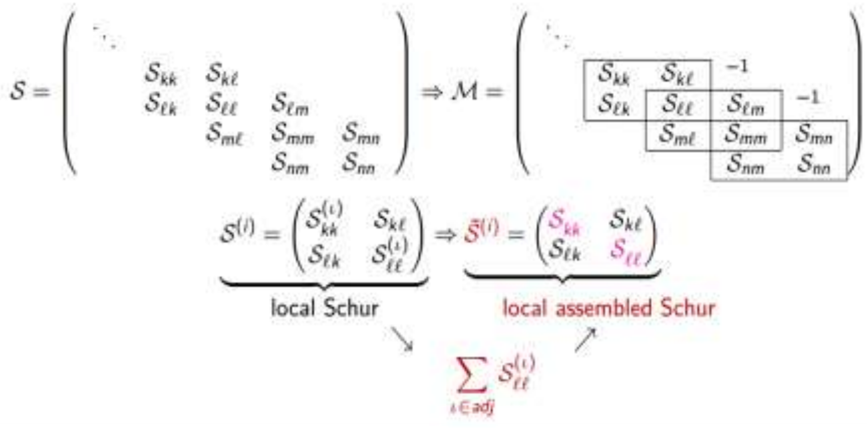

$\mathcal{S} = \begin{pmatrix} \ddots & & & & \\ & S_{kk} & S_{k\ell} & & \\ & S_{\ell k} & S_{\ell\ell} & S_{\ell m} & \\ & & S_{m\ell} & S_{mm} & S_{mn} \\ & & & S_{nm} & S_{nn} \end{pmatrix} \Rightarrow \mathcal{M} = \begin{pmatrix} \ddots & & & & \\ & S_{kk} & S_{k\ell} & -1 & \\ & S_{\ell k} & S_{\ell\ell} & S_{\ell m} & -1 \\ & & S_{m\ell} & S_{mm} & S_{mn} \\ & & & S_{nm} & S_{nn} \end{pmatrix}$
$\underbrace{\mathcal{S}^{(i)} = \begin{pmatrix} S_{kk}^{(i)} & S_{k\ell} \\ S_{\ell k} & S_{\ell\ell}^{(i)} \end{pmatrix}}_{\text{local Schur}} \Rightarrow \underbrace{\bar{\mathcal{S}}^{(i)} = \begin{pmatrix} S_{kk} & S_{k\ell} \\ S_{\ell k} & S_{\ell\ell} \end{pmatrix}}_{\text{local assembled Schur}}$
$\searrow \quad \sum_{i \in adj} S_{\ell\ell}^{(i)} \quad \nearrow$

Figure 6.4 : Algebraic Additive Schwarz preconditioner[49].

Similar to PDSLin, in order to get inexpensive preconditioner, Maphys relies on the sparsifying techniques. However, the sparsfication mechanism is different from that of PDSLin. It uses the following criterion to sparsity the assembled preconditioning matrix:

$$\bar{\bar{S}}_{li} = \begin{cases} 0 & if\,|\,\bar{S}_{li}\,| \le \xi\ (\bar{S}_{ll} + \bar{S}_{ii}) \\ \bar{S}_{li} & otherwise \end{cases} \qquad (6.9)$$

This criterion preserves the symmetry of the symmetric problems[10,50].

Here we summarize the solution phases of Maphys followed in our experiments:

- Partitioning step. Maphys uses nested dissection algorithm of Metis or Scotch for graph partitioning. In our experiments we used scotch-custom option.
- Factorization of the interior subdomains $\boldsymbol{A_{I_iI_i}}$ and the local Schur complement $\boldsymbol{S_i}$. Maphys uses either the supernodal method of Pastix or the multifrontal method of Mumps for this purpose and let the user choose between them. In order to use the 2-level parallelism, Pastix with multithreading option should be compiled and linked with Maphys. We chose Pastix in all our experiments except for serial time, we chose Mumps.
- Preconditioning. As dicussed before, Maphys uses Additive Schwarz preconditioner which consists of two steps: assembly of the local Schur matrices $\boldsymbol{\bar{S}_i}$. The next step is factorization of this assembled Schur matrix. Because $\boldsymbol{\bar{S}_i}$ is a dense matrix, the preconditioner can be expensive. Thus dropping is used as discussed before. This preconditioner is called sparse preconditioner. The user has to define the dropping threshold $\boldsymbol{\xi}$ in order to activate this preconditioner. There is another preconditioner type in Maphys based on ILU(t,p) factorization which is already implemented in pARMS. In our experiments, we used sparse preconditioner.
- Solve step. Similar to PDSLin, There are two consecutive phases here. First solving the interface unknowns and then backsolve of the interiors.

6.2.1 Multithreading in Maphys

In order to implement the two level parallelism in Maphys. Maphys developers use three types of binding and let the user decide which binding to choose. The first type is not to use any kind of binding and let the operating system handle the binding between processes and threads(Figure (6.5a)). The second type is to bind processes to cores and let the operating system handle the placement of threads(Figure (6.5b)). The third type is to handle the placement of threads and processes(Figure (6.5c)). Since such bindings depend on the operating system and architecture of the target machine, Maphys relies on hwloc software [51] for getting these information from the machine. In [10], they tested the three types of binding and concluded that the second type gives the best results so we used this type of binding in all our experiments when evaluating the two level parallelism of Maphys. More about this can be found in the result section.

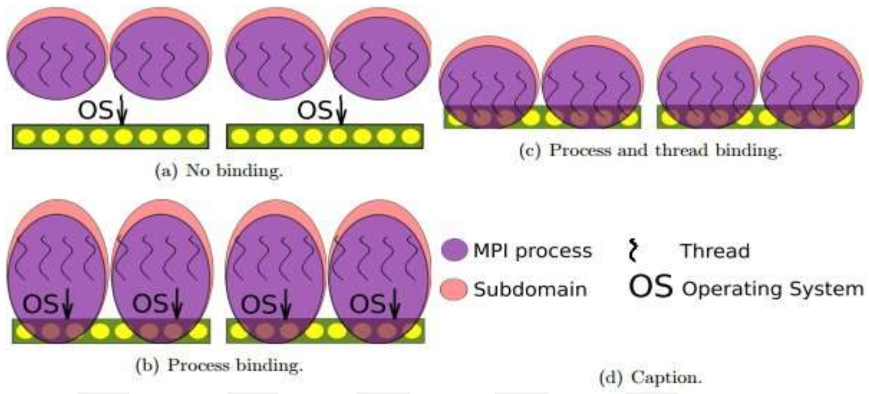


Figure 6.5 : The Three binding mechanisms of Maphys[10].

6.3 HIPS

Similar to the previous solvers, HIPS (Hierarchical Iterative Parallel Solver) uses direct/iterative methods through Schur complement. GMRES is used for solving the Schur complement S preconditioned by incomplete LU of S; i.e M ≈ $\bar{L}_s\bar{U}_s$ as follows:

The local block matrices of A_{II} of equation(6.2) are factorized independently with exact factorization using supernodal right looking algorithm. In the next step, an approximate computation of the global $\bar{L}_s\bar{U}_s$ using left looking ILUT(τ) algorithm where:

$$\bar{L}_s\bar{U}_s = \bar{S} - (A_{\Gamma I}\, U^{-1})(L^{-1}{}_{II}A_{I\Gamma}) \qquad (6.10)$$

HIPS uses a different graph partitioning and reordering algorithm called Hierarchical Interface Decomposition Algorithm (HID) as well as Metis or Scotch for ordering internal unknowns. Here we briefly outline the algorithm of HID. Reader should consult [52] for further details .

HID is an edge-based partitioning algorithm ,i.e, the overlaps are over vertices rather edges. The global domain is partitioned into levels; each level has a set of subgraphs called connectors such that connectors of a given level are 'separators' of the levels below. Figure (6.6) shows an example of partitioning 5-point mesh of 2D Poisson equation into 9 subdomains using this scheme. In this example, we can distinguish three types of points: interiors which are the lowest level. The interface nodes which are the separators of the interior nodes and the cross points; separators of the interface nodes. Such a hierarchal decomposition is appealing for parallel processing since factorization can proceed independently starting from the lowest level using ILU[52,53] . However, we should keep this HID structure as intact as possible and minimize fill-in between connectors.

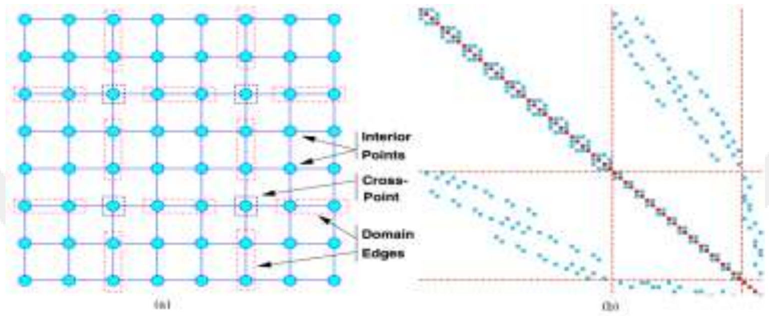


Figure 6.6 : (a)Partition of an 8× 8 5-point mesh into 9 subdomains and the corresponding HID structure (b) Matrix associated with an 8× 8 5-point mesh reordered according to HID [52].

In order to keep the HID structure, developers of HIPS suggested two dropping strategies: locally consisted strategy and strictly connected strategy.

In **strictly consisted strategy**, fill-in is allowed only in places that will not destroy the HID block diagonal structure. **Locally consistent strategy**, fill-in is allowed in any block matrix.

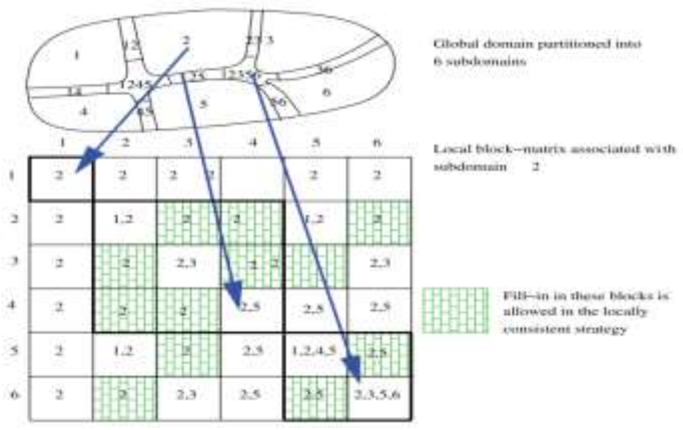

Global domain partitioned into 6 subdomains
Local block—matrix associated with subdomain 2
Fill–in in these blocks is allowed in the locally consistent strategy

Figure 6.7 : block partition of subdomain(2) [52].

Figure (6.7) shows the block matrix of subdomain (2) as an example. The diagonal elements show the levels: level labeled (2) is the lowest level (level 0) and takes the label of the subdomain itself. connectors of the second level(level one): (1,2), (2,3), (2,5) which are interfaces of level 0 (previous level) and connectors of level 2: (1,2,4,5),(2,3,5,6) or the cross points. The figure also shows places where fill-in is allowed using locally consistent strategy.

Saad in [53] states that there are two important ingredients of this method: (1) good levelization(few levels). (2) good combination of incomplete factorization algorithm and dropping strategy.

As opposed to PDSLin, HIPS uses one level mapping of processors to subdomain ,i.e, each processor is assigned to multiple subdomain. This ensures a good load balancing but global size of the Schur complement increases with increasing number of subdomain [6]. In our experiments, we used one level parallelism with Maphys using ASIC680ks matrix.

6.4 pARMS

This solver is based on a multilevel recursive algorithm called Algebraic Recursive Multilevel Solver (ARMS) developed by Yousef Saad[54]. So it is a multilevel approach, in the first level, the global system is ordered into 2×2 blocks of matrices similar to that of equation(6.2). Nested dissection algorithm of Metis or row and column permutation, i.e, PAP^T can be used for such ordering. For convenience, we are going to rewrite the equation (6.2) with superscript denoting the level number as follows:

$$\begin{pmatrix} A^{(l)}{}_{II} & A^{(l)}{}_{I\Gamma} \\ A^{(l)}{}_{\Gamma \mathrm{I}} & A^{(l)}{}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x^{(l)}{}_{I} \\ x^{(l)}{}_{\Gamma} \end{pmatrix} = \begin{pmatrix} b^{(l)}{}_{I} \\ b^{(l)}{}_{\Gamma} \end{pmatrix} \qquad (6.11)$$

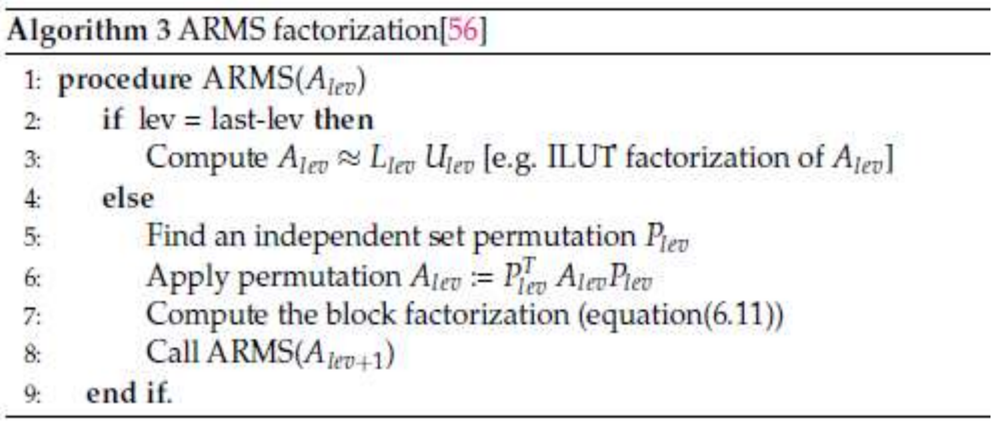
The above equation is then approximately factored like this:

$$\begin{pmatrix} A^{(l)}{}_{II} & A^{(l)}{}_{I\Gamma} \\ A^{(l)}{}_{\Gamma \mathrm{I}} & A^{(l)}{}_{\Gamma\Gamma} \end{pmatrix} \approx \begin{pmatrix} L^{(l)} & 0 \\ b^{(l)}{}_{\Gamma \mathrm{I}}\, U^{-1^{(l)}} & I \end{pmatrix} \times \begin{pmatrix} U^{(l)} & L^{-1^{(l)}} A^{(l)}{}_{I\Gamma} \\ 0 & S^{(l+1)} \end{pmatrix} \qquad (6.12)$$

where I is the identity matrix, $\boldsymbol{U^{(l)}}$ and $\boldsymbol{L^{(l)}}$ are the LU (or ILU) factors of $\boldsymbol{A^{(l)}{}_{\Gamma\Gamma}}$ and $\boldsymbol{S^{(l+1)}}$ is an approximate of the Schur complement of equation(6.3).

$$S^{(l+1)} = A^{(l)}{}_{I\Gamma} - \left(A^{(l)}{}_{\Gamma I} \;\; U^{-1^{(l)}}\right)\left(L^{-1^{(l)}} \, A^{(l)}{}_{I\Gamma}\right) \qquad (6.13)$$

All matrices and matrix multiplications of equation(6.13) are approximated with certain dropping threshold values in a manner similar to that of PDSLin and Maphys. In the next step, we repeat this same process with resultant S matrix. ARMS algorithm is shown at (3)below


Algorithm 3 ARMS factorization[56]

```
1: procedure ARMS($A_{lev}$)
2:     if lev = last-lev then
3:         Compute $A_{lev} \approx L_{lev} U_{lev}$ [e.g. ILUT factorization of $A_{lev}$]
4:     else
5:         Find an independent set permutation $P_{lev}$
6:         Apply permutation $A_{lev} := P_{lev}^T A_{lev} P_{lev}$
7:         Compute the block factorization (equation(6.11))
8:         Call ARMS($A_{lev+1}$)
9:     end if.
```

For LU(or ILU) factorization of the last level at step(3), different approximations can be used. Pivoting is not performed during the block factorization except for the last level where a pivoting factorization technique like ILUTP or GMRES preconditioned with ILUT can be used. Saad in[56] divides ARMS into three phases: the first phase(called the forward or restriction) phase as analogous to the coarsening phase of AMG method. Various number of methods may be used in this phase like VARMS and WARMS.

6.5 ABCD

Augmented Block Cimmino Distributed Solver(ABCD) is develpoed from a PhD thesis of Mohamed Zenadi. The official website of this solver is [56] and the github account is [57]. This solver is based on an iterative method using block-row projections. called block Cimmino method[58]. For convenience, we present the original algorith below. The original system (Ax = b) is partitioned into p blocks of rows such that,

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix} \qquad (6.14)$$

PaToH Hypergraph partitioner is used for row permutation. The justification for this row-wise partitioning is that an ill-conditioned matrix A has some linear combination of rows almost equal to the zero vector. After such row partitioning, these may occur within the blocks or across the blocks. If, in the block Cimmino algorithm, we assume that we compute the projections on the subspaces exactly, the rate of convergence of the method will depend only on the conditioning across the blocks[59].

From initial estimates of $\boldsymbol{x^{(0)}}$, the algorithm approximates the solution iteratively according to

$$u_i = A^+{}_i\,(b_i - A_i\,x^{(k)}) \qquad \mathrm{i} = 1, \ldots, \mathrm{p} \qquad (6.15)$$

$$x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} u_i \qquad (6.16)$$

where $\boldsymbol{A^+{}_i}$ is the Moore–Penrose pseudo-inverse(explicitly $\boldsymbol{A^+{}_i} = \boldsymbol{A^T{}_i(A_i\,A^T{}_i)^{-1}}$) of the matrix $\boldsymbol{A_i}$ and $\boldsymbol{\omega}$ is the relaxation parameter. Similar to Jacobi iteration method, equations in (6.15) are independent and can be solved in parallel. We can write the above equations like:

$$x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} A^+{}_i\,(b_i - A_i\,x^{(k)}) \qquad (6.17)$$

$$= (I - \omega \textstyle\sum_{i=1}^{p} A^+{}_i\,A_i)\,x^{(k)} + \omega \textstyle\sum_{i=1}^{p} A^+{}_i\,b_i \qquad (6.18)$$

$$= Q\,x^{(k)} + \omega \sum_{i=1}^{p} A^+{}_i\,b_i \qquad (6.19)$$

The iteration matrix for the block Cimmino method is (H = I - Q) which corresponds to a sum of projectors (H = $\boldsymbol{\omega} \sum_{i=1}^{p} \boldsymbol{P_{R(A^T{}_i)}}$ where $\boldsymbol{R(A^T{}_i)}$ is a projection matrix onto the range of $\boldsymbol{A^T{}_i}$. If $\boldsymbol{\omega} > 0$, H will be symmetric positive semidefinite and it is symmetric positive definite if A is square and of full rank. So we can solve equation (6.20):

$$Hx \;=\; \omega \sum_{i=1}^{p} A^{+}{}_{i}\, b_i \qquad (6.20)$$

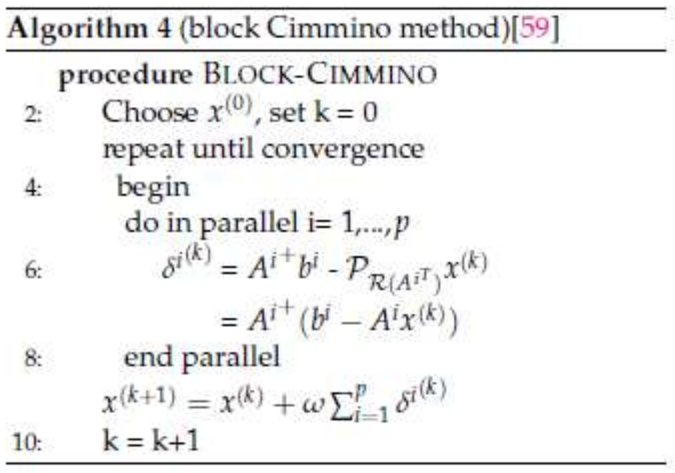
using conjugate gradient or block conjugate gradient methods. As the relaxation scalar $\boldsymbol{\omega}$ appears on both sides of the equation, we can set it to one. At each step of the conjugate gradient algorithm, we must solve for the p projections

$$A_i\, u_i \;=\; r_i\ ,\ \left(r_i \;=\; b_i \;-\; A_i\, x^{(k)}\right),\quad i \;=\; 1\,,2\,\ldots p. \qquad (6.21)$$

To solve the subproblems of (6.15, 6.21) system, the following augmented system is used:

$$\begin{pmatrix} I & A^{(T)}{}_{i} \\ A_i & 0 \end{pmatrix}\begin{pmatrix} u_i \\ v_i \end{pmatrix} \;=\; \begin{pmatrix} 0 \\ r_i \end{pmatrix} \qquad .. \quad (6.22)$$

The reason for this augmented system approach is that it is more stable and less sensitive to ill-conditioning within the blocks caused by this row-wise partitioning and thus accelerate the convergence of the block Cimmino method. Second it helps using the ellipsoidal norms and the corresponding oblique projectors used.


Algorithm 4 (block Cimmino method)[59]

```
procedure BLOCK-CIMMINO
    Choose $x^{(0)}$, set k = 0
    repeat until convergence
      begin
       do in parallel i= 1,...,$p$
           $\delta^{i(k)} = A^{i+}b^{i} - \mathcal{P}_{\mathcal{R}(A^{i^T})}x^{(k)}$
                $= A^{i+}(b^{i} - A^{i}x^{(k)})$
       end parallel
    $x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} \delta^{i(k)}$
    k = k+1
```

The sparse direct multifrontal solver MUMPS is used to solve this augmented system. At each iteration to get $\boldsymbol{u_i} \;=\; \boldsymbol{A^{+}{}_{i} r_i}$, the projection is needed for each partition $\boldsymbol{A_i}$.

According to [60], there are two ways that ill-conditioning can affect the solution using block Cimmino:

- Within the blocks where the systems being solved are symmetric indefinite problems, ill-conditioning can cause any sparse direct method to behave poorly or unpredictably.
- Across the blocks where there can be problems if the subspaces are far from orthogonal.

7. EXPERIMENTAL SETUP AND RESULTS

In this chapter we discuss the experimental method and environment we used for comparing those methods. The hybrid solvers we evaluate are Maphys and PDSLin. We compare the results with state-of-art Superlu-dist direct solver.

7.1 Experimetal environment and Optimization Details

Our experiments are conducted on a linux cluster with Slurm workload manager called Sariyer cluster at Ulusal Yüksek Başarımlı Hesaplama Merkezi Projesi (UHeM). Sariyer nodes consist of two sockets with 14-core Intel(R) Xeon(R) CPU

E5-2680 v4 @ 2.40GHz processors in each socket (total of 28 cores per node).

All the libraries are compiled using Openmpi, Intel MKL library and gcc@7.1.0 compiler with 03 optimization flag. Due to the large number of libraries and their dependencies, we could not compile the whole libraries with Intel compilers and consider that as a future work. With superlu-dist and PDSLin, we used one debugging and print level, namely, -DDEBUGlevel=1 -DPRNTlevel=1. All the libraries are compiled with 32 bit arithmatic and matrix indices (integer limit 2^{31}-1) and we consider 64 bit as a future work. The versions of the libraries appear in Appendix(A), table(A2).

We used the matrix format IJV, MTX and HB and Matlab for converting matrices from Matlab format into IJV/HB. Because MTX takes explicit zeros into account, for the sake of fair comparison, we took those explicit zeros into account during conversion in all types of matrices too. The reason of taking multiple matrix formats is that different solvers support different matrix formats and a solver sometimes fails on a certain matrix format and works with other format. For example, Maphys fails to run (not fails to give a solution) with atmosmodl MTX matrix format and works with HB format. PDSLin works better when using IJV format for symmetric matrices. Because Matlab automatically drops zeros from sparse matrices, we substitute the explicit zero values by $\mathbf{10^{-30}}$.

Finally, following the conventions in [61], we distinguish three types of failur: $\boldsymbol{F_C}$ indicates an abnormal or no termination failure, $\boldsymbol{F_M}$ indicates that the solver runs out of memory, $\boldsymbol{F_N}$ indicates that the results were not accurate.

7.2 Matrix Description

The four matrices we selected for evaluation are all from Florida University collection. These are of moderate sizes. The description of the matrices is shown in table (7.1). The criteria of choosing these matrices are the size of the matrix, number of nonzero elements, source of the matrix, the sparsity structure and degree of diffcultiy. The condition number of the matrices is taken from Matlab function condest(). According to matlab documentation, condest() is based on the 1-norm condition estimator of Hager and a block-oriented generalization of Hager's estimator. The heart of the algorithm involves an iterative search to estimate $||\boldsymbol{A}^{-1}||_1$without computing this inverse[62].

Table 7.1: List of matrices used for evaluating hybrid solvers**.**

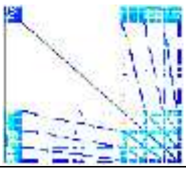
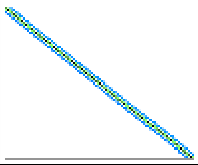
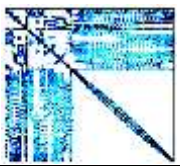
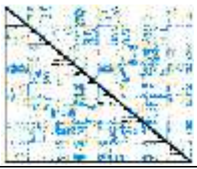
Name	**Freescale1**	**Atmosmodl**	**Audikw-1**	**ASIC_680ks**
n	3.4M	1.5M	934K	682.7K
nnz	17M	10.31M	77.7 M	1.7 M
Cond No	1.074941e+10	1.472850e+03	6.952866e+10	9.474649e+19
Explicit Zeros	1.9M	0	0	0.6M
Sym (pattern Sym%)	unsymmetric (0%)	unsymmetric (100%)	symmetric (100%)	unsymmetric (100%)
Type	real	real	real	real
Source	Circuit Simulation	Computational Fluid Dynamics	3D Structur	Circuit Simulation
Maphys Serial Time	34.81	321.6	199.4	69.03
PDSLin Serial Time	72.397	659.40688	684.700	8.0897
Superlu-dist Serial Time	F_N	676.79	622.1	F_N
Sparsity Pattern				

Table 7.2: Processor Distribution and Sparsifying Tolerance Setting on the Selected Matrices

			Audik					**Freescale**						**ASIC**					atmosmodl				
#nodes			**1**	**2**	**8**	**8**	**11**	**1**	**1**	**1**	**2**	**8**	**16**	**1**	**1**	**8**	**8**	**16**	**2**	**2**	**8**	**8**	**16**
#cores			**16**	**32**	**64**	**128**	**256**	**4**	**8**	**16**	**32**	**64**	**256**	**4**	**16**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
#subdomains	MaPHyS		16	16	16	16	16	4	4	8	4	16	16	4	16	64	128	256	8	8	16	16	16
	PDSLin		4	8	8	8	8	4	4	4	4	16	16	4	8	8	8	8	4	4	8	8	16
# cores	MaPHyS threads		1	2	4	8	16	1	2	2	8	4	16	4	16	64	128	256	2	4	4	8	16
	PDSLin	n_{gl}	2	4	16	32	64	1	2	4	8	4	16	1	2	8	16	32	2	4	4	8	16
		n_{gs}	16	16	16	4	64	4	4	4	8	16	8	4	4	32	8	4	4	4	8	16	16
preconditioner	MaPHyS		10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}
	PDSLin	τ_G	10^{-4}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-4}	10^{-20}	10^{-20}	10^{-20}	10^{-20}	10^{-20}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}
		τ_W	10^{-1}	10^{-2}	10^{-2}	10^{-2}	10^{-2}	dense	dense	dense	dense	dense	dense	10^{-20}	10^{-20}	10^{-20}	10^{-20}	10^{-20}	dense	dense	dense	dense	dense
		τ_S	10^{-5}	10^{-5}	dense	dense	dense	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	*dense*	10^{-20}	10^{-20}	10^{-20}	10^{-20}	10^{-20}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}
#iterations	MaPHyS		76	97	96	88	68	17	17	17	46	46	46	2	2	--	--	--	20	20	25	25	25
	PDSLin		43	156	8	10	10	2	2	2	2	2	8	4	2	2	4	4	16	16	18	18	18

7.3 Evaluation Metrics

For evaluating the solvers, we extract the information from the generated output files. We used the same stopping criteria for all solvers($\mathbf{10^{-12}}$) the error measurement for the three solvers is shown in table(7.3) below.

Table 7.3: List Error Measurements

Solver	Error Measurement	
Superlu-dist	FERR = $\|\|\boldsymbol{x_{true}} - \boldsymbol{x}\,\|\|_\infty$ / $\|\|\boldsymbol{x}\,\|\|_\infty$	$\|\|\boldsymbol{x}\,\|\|_\infty = \boldsymbol{max_i}\,\|\boldsymbol{x_i}\|$
PDSLin	Err.nrm = $\sqrt{\|\|\boldsymbol{x_{true}} - \boldsymbol{x}\,\|\|_2}$ / $\sqrt{\|\|\boldsymbol{x}\,\|\|_2}$	$\|\|\boldsymbol{x}\,\|\|_2$ Euclidean norm
Maphys	Norm.res = $\|\|\boldsymbol{Ax} - \boldsymbol{b}\,\|\|_2$ / $\|\|\boldsymbol{b}\,\|\|_2$	

Since time and memory are the major concerns in sparse matrices, we studied those parameters here. The time measurement is the MPI wall-clock time function; MPI_Wtime(). For memory measurement, different solvers use different tools to extract these information.

Finally, we used Speedup measurement according to the definition below:

Definition The speedup(**S(p)**) is defined as : S(p) =T(1)/T(P). where T(1) is the serial time; we ran the code with a single processor and T(P) is the parallel time using P number of processors.

The control parameters for solving the selected matrices are show in table (7.2) above. As mentioned before, the criteria of selecting these values is for best performance of the solver. We should mention here that except for ASIC680KS, the matrices are simple matrices and that is why sparse preconditioner is used as shown in table(7.2). By setting the number of subdomains to 4,8 and 16 in PDSLin and Maphsy, we increased the number of threads/processors from 16 to 256 in each solver. We chose a power of 2 for subdomains since nested dissection method is used for graph partitioning. The speedup is show in figure(6.8) below.

Table7.4: Superlu-dist options selected

Name	Audik-1	Freescale1	ASIC680ks	atmosmodl
Equil	YES	YES	YES	YES
ColPerm	PARMETIS	PARMETIS	PARMETIS	PARMETIS
RowPerm	NO	LargeDiag	LargeDiag	LargeDiag
SymbFact	ParSymbFact	ParSymbFact	ParSymbFact	Serial on $A^T + A$
SymPattern	YES	NO	NO	NO
ReplaceTinyPivot	YES	YES	YES	YES
IterRefin	NO	NO	NO	NO
Trans	NO	NO	NO	NO
SolveInitialized	NO	NO	NO	NO
RefineInitialized	NO	NO	NO	NO

The serial/Parallel time in Maphys is the total execustion time RINFO(21). For PDSLin, the parallel time is the time to factorize local subdomains + time to compute approximate Schur + time preprocessing approximate Schur + time factorizing Schur matrix + solve time. The serial time in PDSLin is the factorization time + the solve time. The parallel/serial time in Superlu-dist is equal to equilibration time + row permutation time(if exists) + column permutation time + symbolic factorization time + distribute time + numerical factorization time + solve time.

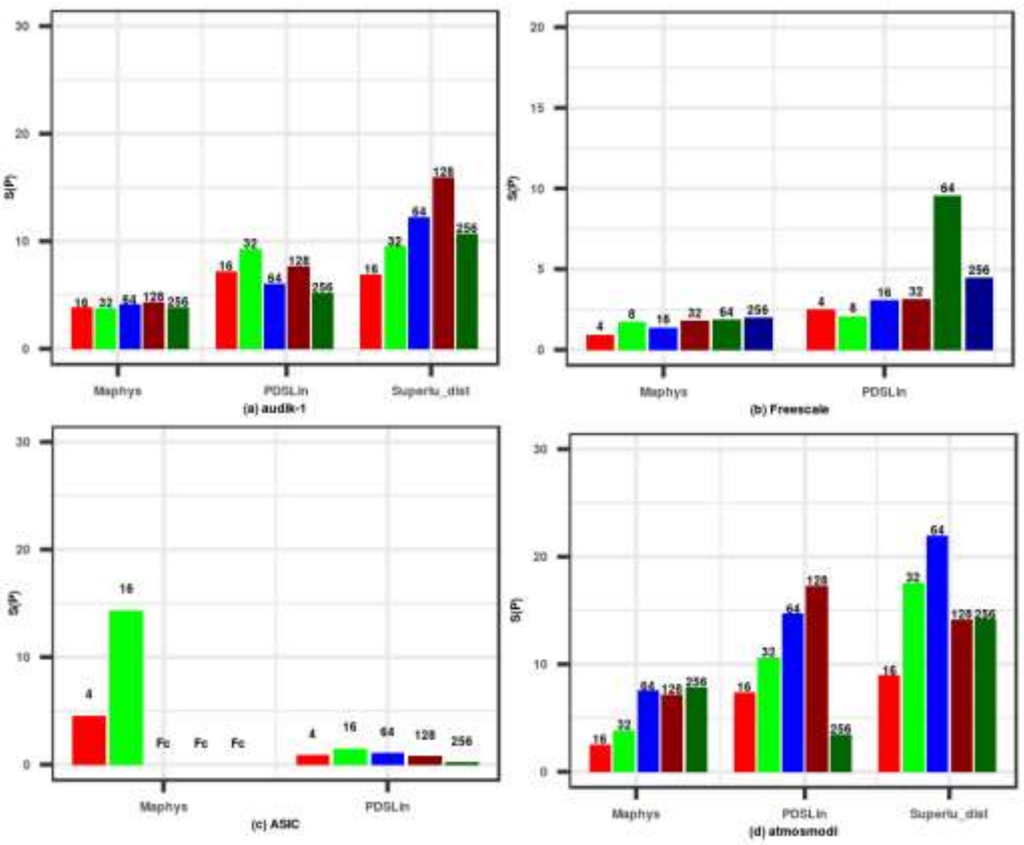


Figure 7.1: Speedup of Maphys, PDSLin, and Superlu on Audik(a), Freescale(b)ASIC680ks(c) and Atmosmodl(d).

As shown in table (7.1) above, In Audik-1 matrix, the serial time for the three solvers is approximately (199.4, 684.700, 622.1) seconds for Maphys, PDSLin and Superlu-dist restrictively. It is significanet to see that serial time in Maphys is three times faster than Superlu-dist and PDSLin. Because we are using a single subdomain, Maphys and PDSLin work as a direct solver. Maphys uses MUMPS and PDSLin uses Superlu-dist in this case.

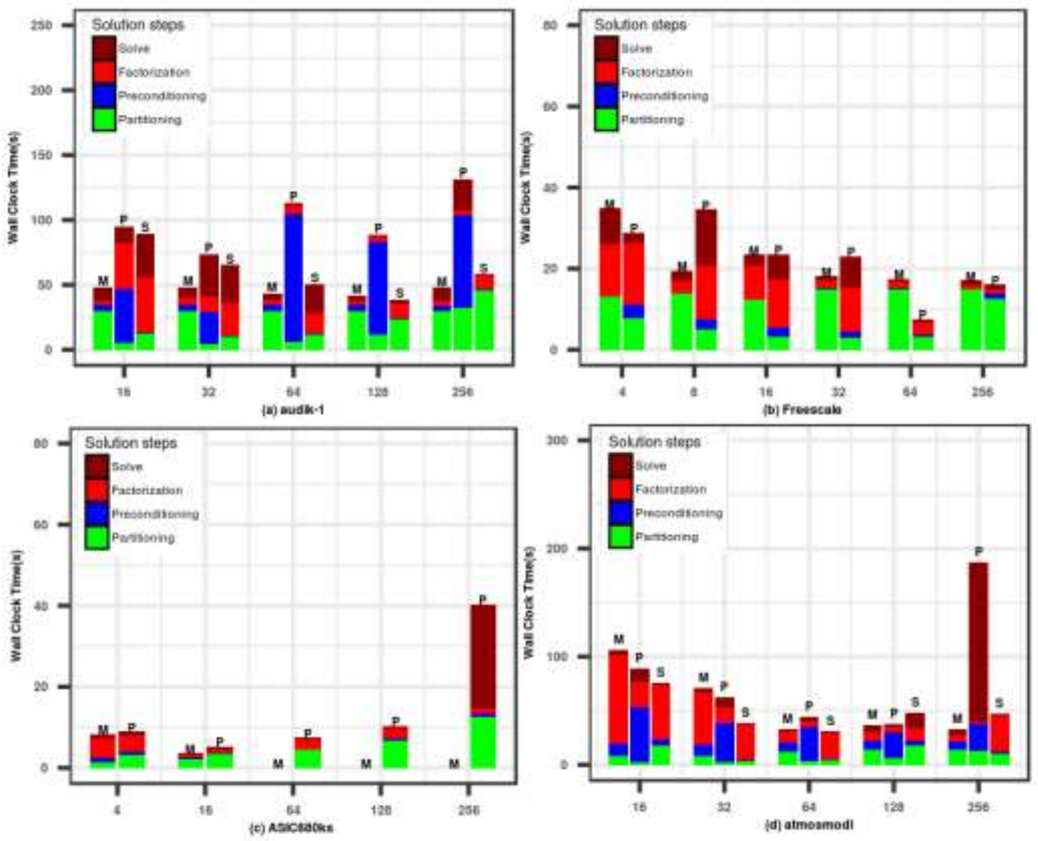


Figure 7.2: Solution steps of Maphys(M), PDSLin(P) and Superlu-dist(S) on Audik(a), Freescale(b)ASIC680ks(c) and Atmosmodl(d).

In Freescale and ASC680ks, Superlu-dist fails so it is not included in the graphs(The estimated forward error is approximately one even with itrative refinement). As shown in Table(7.1), Freescale1 matrix is an arrow matrix so with a good column permutation, small amount of fill-in can occur. As shown in Figure(7.1), Maphys scales better than PDSLin using such input parameters. Both solvers make use of increasing number of processors and scales very well in Freescale matrix.

The two level parallelism continues working in PDSLin with increasing number of processors. The two level parallelism in Maphys in ASIC680KS fails. We used one level parallelism which fails at 64 processors. In this one level parallelism, number of processors is equal to the number of subdomains and this makes the Schur complement matrix size increases rapidly. This might be the reason on Maphys failure at 64 processors and beyond.

In atmosmodl matrix, all PDSLin and Maphys scales very well upto 128 processors. Again the facotization and solution time in Maphys is twice than that of Superlu-dist and PDSLin.

7.4 Schur Complement Processing

The following tables show Schur matrix number of entries after applying dropping threshold along with number of subdomains for both solvers PDSLin and Maphys.

Tables(5 - 12) shows the Schur complement information of Audik on PDSLin and Maphys respectively. In PDSLin, when the dropping threshold $\boldsymbol{\rho_2}= 10^{-5}$around 25% entries are kept using 4 and 8 subdomains and when $\boldsymbol{\rho_2}$= 0, a dense preconditioner is used. In that case, number of iterations and consequently solution time is getting smaller.

Table7.5: Schur Matrix Info in Audik using PDSLin Solver

P	#subdom	ρ_2	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
16	4	1.000000e-05.	12539	26100057	25,0916	34
32	8	1.000000e-05.	29177	38836822	28,1691	156
64	8	0.000000e+00.	27874	304736476	100	8
128	8	0.000000e+00.	27570	301717802	100	10
256	8	0.000000e+00.	26906	293136106	100	10

Maphys uses a local preconditioner, the Schur matrix size is much smaller. When ρ= $\mathbf{10^{-4}}$ using 16 subdomains, 9% of the entries are kept. This gives us an average of 85 iterations. The solution time scales well with these settings up to 128 processors.

Table7.6 Schur Matrix Info in Audik using Maphys Solver

Threads	#subdom	ρ	Schur Size(avg)	Nnz($\bar{S}$) (avg)	Nnz Kept (%)	Niter
16	16	0,0001	6304	43240000	9.062	76
32	16	0,0001	6304	43240000	9.062	97
64	16	0,0001	6304	43240000	9.062	96
128	16	0,0001	6304	43240000	9.062	88
256	16	0,0001	6304	43240000	9.062	68

Table7.7: Schur Matrix Info in Freescale using PDSLin Solver

P	#subdom	ρ_2	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
4	4	1.000000e-10.	1003	14195	16,2206	2
8	4	1.000000e-10.	989	13955	15,6357	2
16	4	1.000000e-10.	1237	18926	15,9283	2
32	4	1.000000e-10.	937	11208	14,7863	2
64	16	1.000000e-10.	1832	22814	16,1188	2
256	16	0.000000e+00.	1671	3799	100	8

Table(7.7) shows the Schur complement information of Freescale. When the dropping threshold $\boldsymbol{\rho_2} = \mathbf{10^{-10}}$, around 15% entries are kept using 4 and 16 subdomains and when $\boldsymbol{\rho_2}$ = 0, a dense preconditioner is used.

In Maphys, ρ= $\mathbf{10^{-6}}$, only 1% and 1.5% of nnz entries are kept in Freescale matrix as shown in table(7.8) below.

Table7.8 Schur Matrix Info in Freescale using Maphys Solver

Threads	#subdom	ρ	Schur Size(avg)	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
4	4	1E-06	577.2	338000	1	17
8	4	1E-06	577.2	338000	1	17
16	4	1E-06	577.2	338000	1	17
32	16	1E-06	344.2	133700	1.5	46
64	16	1E-06	344.2	133700	1.5	46
256	16	1E-06	344.2	133700	1.5	46

The entry values of ASC680ks are very small so even with , $\boldsymbol{\rho_2} = \mathbf{10^{-20}}$, almost all entries are kept and a dense preconditioner is used as shown in table(7.9) below.

Table7.9 Schur Matrix Info in ASIC680KS using PDSLin Solver

P	#subdom	ρ_2	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
4	4	1.000000e-20.	781	1249	99,3636	4
16	8	1.000000e-20.	1124	1851	100	2
64	8	1.000000e-20.	1155	1837	99,7827	2
128	8	1.000000e-20.	1086	1707	99,8246	4
256	8	1.000000e-20.	1099	1777	99,7754	4

As a contrast, using $\boldsymbol{\rho_2} = \mathbf{10^{-6}}$, thrshold, only 1% of the nnz entries are kept in the local Schur complement as shown in ytable(7.10) below.

Table7.10 Schur Matrix Info in ASIC680ks using Maphys Solver

Threads	#subdom	ρ	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
4	4	1E-06	5032	28300000	1	2
16	16	1E-06	2771	9574000	1	2
64	64	1E-06	0	0	0	0
128	128	1E-06	0	0	0	0
256	256	1E-06	0	0	0	0

For Atmosmodl matrix, we fixed the dropping threshold $\boldsymbol{\rho_2} = \mathbf{10^{-5}}$ In PDSLin and $\boldsymbol{\rho_2} = \mathbf{10^{-6}}$ in Maphys. The results are shown below.

Table7.11 Schur Matrix Info in Atmosmodl using PDSLin Solver

P	#subdom	ρ_2	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
16	4	1.000000e-05.	14886	6250461	6,41983	16
32	4	1.000000e-05.	14753	6006205	6,1622	16
64	8	1.000000e-05.	29351	11962169	4,49453	18
128	8	1.000000e-05.	29437	11717100	4,271	18
256	16	1.000000e-05.	43456	17654959	4,39066	18

Table7.12 Schur Matrix Info in Atmosmodl using Maphys Solver

Threads	#subdom	ρ	Schur Size	Nnz($\bar{S}$)	Nnz Kept (%)	Niter
16	8	1E-06	7467	58800000	14,25	20
32	8	1E-06	7467	58800000	14,25	20
64	16	1E-06	5641	33500000	18,5	25
128	16	1E-06	5641	33500000	18,5	25
256	16	1E-06	5641	33500000	18,44	25

In the next figure, we elaborate further on time spent for Schur complement setup. The red color (schurSolvemax) shows the total maximum time spent for sparse traingular solve(both lower and upper triangular). The green color (schurSymbolmax) shows the symbolic computation of approximate Schur. The blue color shows sparse matrix-matrix multiply of $(\boldsymbol{A}_{\boldsymbol{\Gamma}\mathbf{I}}\,\boldsymbol{U}^{-\mathbf{1}}{}_{\mathbf{II}})$ (called G matrix) and ($\boldsymbol{L}^{-\mathbf{1}}{}_{\mathbf{II}}\boldsymbol{A}_{\mathbf{I}\boldsymbol{\Gamma}}$) (called W matrix) of section (6.1). The yellow color time spent for MPI communication setup(send, recv etc). The brown color shows approximate Schur computation setup. Finally, the black color shows the time of updating with messaging. All these values are the maximum values. It is clear that decreasing the sparifing tolerance , $\boldsymbol{\rho}_{\mathbf{2}}$ which consequnetly increases the nnz in the approximate Schur will lead to an incease in MM multiply as shown in some cases below(the blue color). The traingular solve is the most time consuming in almost all other cases here as shown in figure(7.3) below.

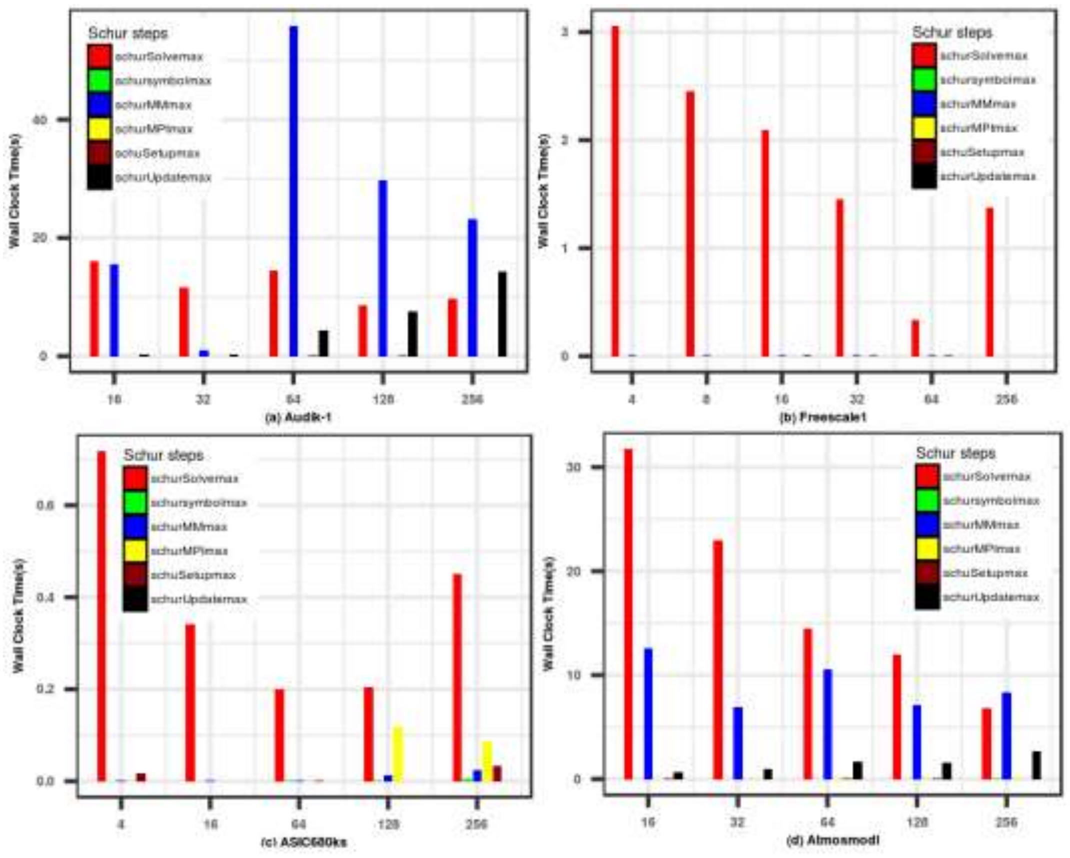


Figure 7.3: Schur Complement Steps of PDSLin for solving Audik(a), Freescale(b)ASIC680ks(c) and Atmosmodl(d).

7.5 Memory Vs Time

The following graphs show time agains memory for Maphys, PDSLin and Superlu-dist on Audik-1, Freescale, ASIC680ks and Atmosmodl matrices respectively.

The Figure (7.4) below shows total memory consumption during solution of Audik-1 matrix using Superlu-dist solver. Because most of the time is spent in factorization, the total time spent takes the shape of factiorization time shape. The first graph (Figure(7.4 a)) shows the actual memory consumption. The summation of all memory is about 44GB and increasing from 16 to 256.

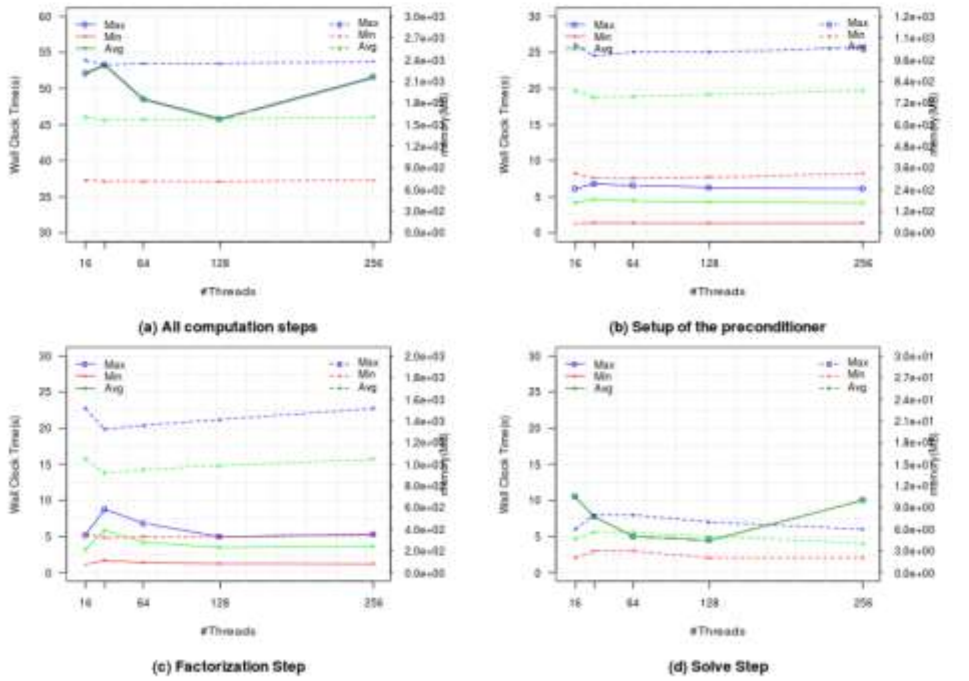


Figure 7.4 : Wall clock time and memory usage of Maphys solver on Audik-1 matrix

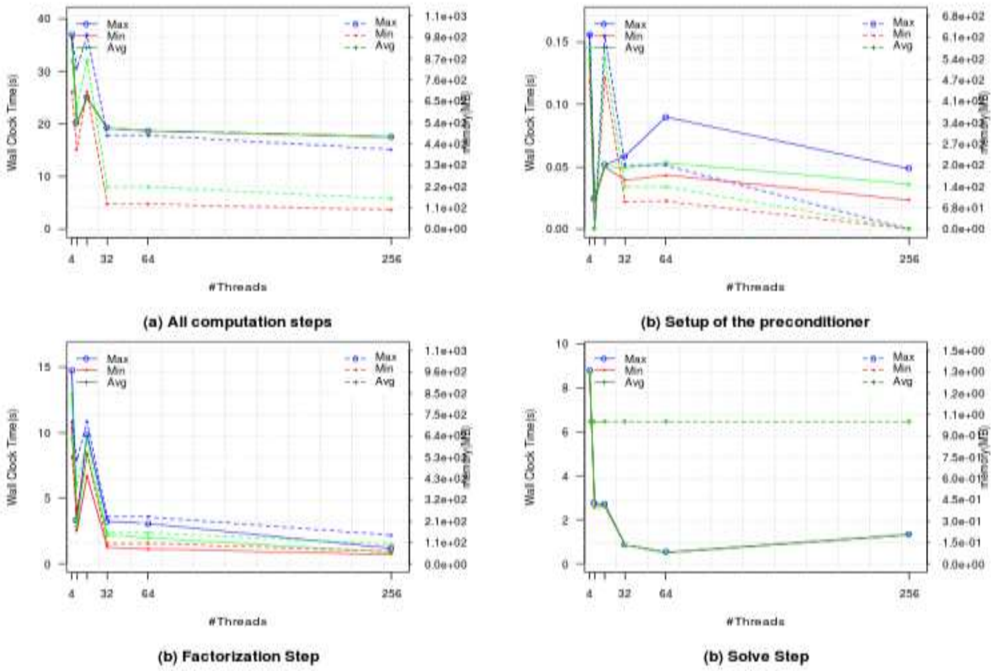


Figure 7.5: Wall clock time and memory usage of Maphys solver on Freescale1 matrix

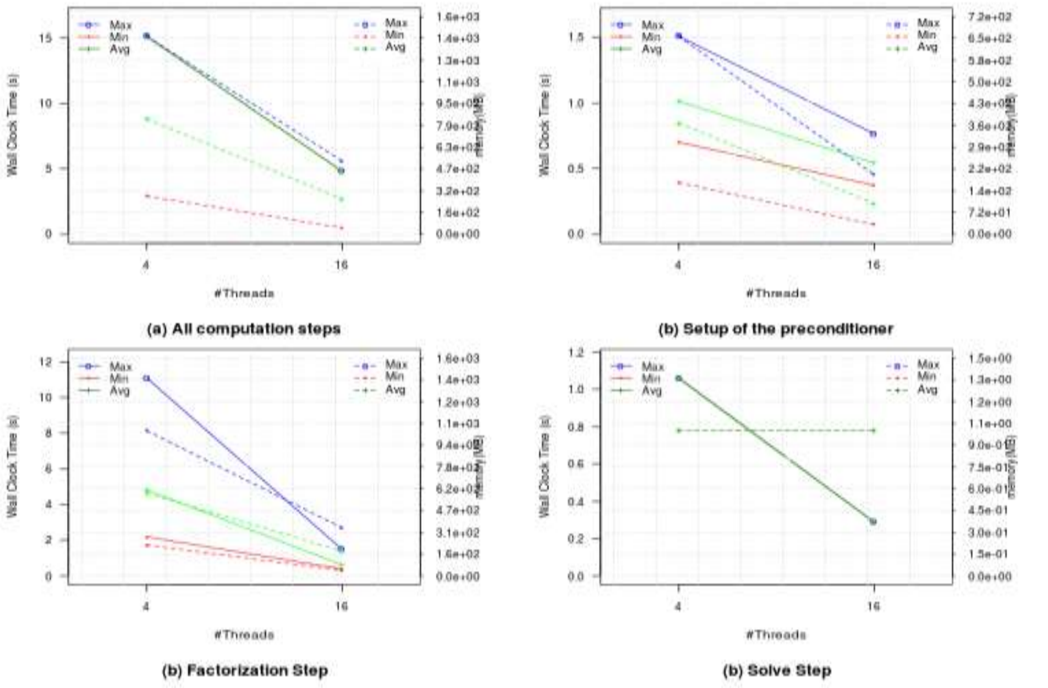


Figure 7.6: Wall clock time and memory usage of Maphys solver on ASIC680ks matrix

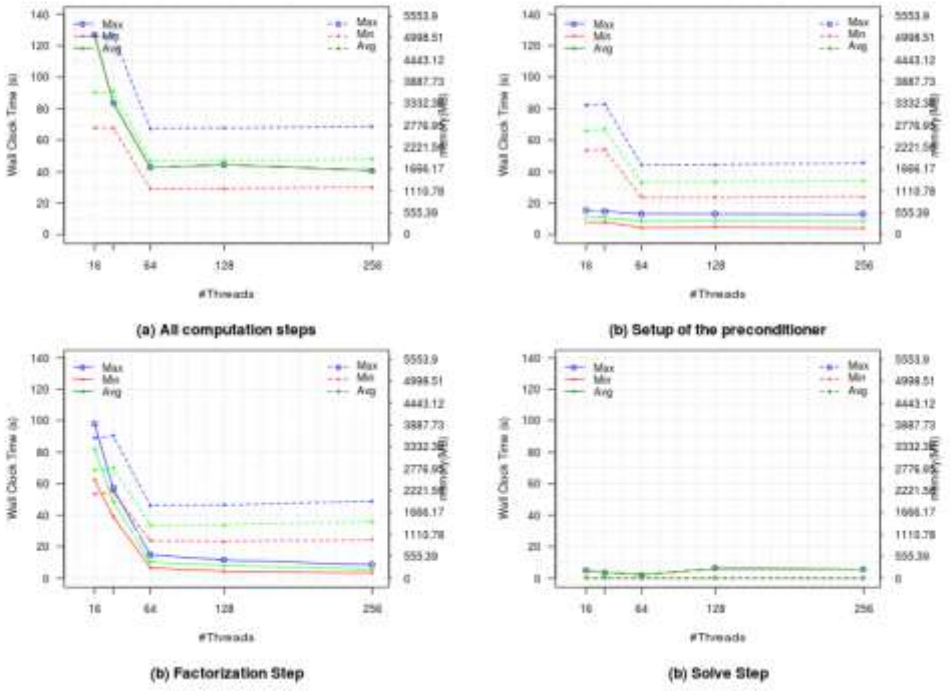


Figure 7.7: Wall clock time and memory usage of Maphys solver on Atmosmodl matrix

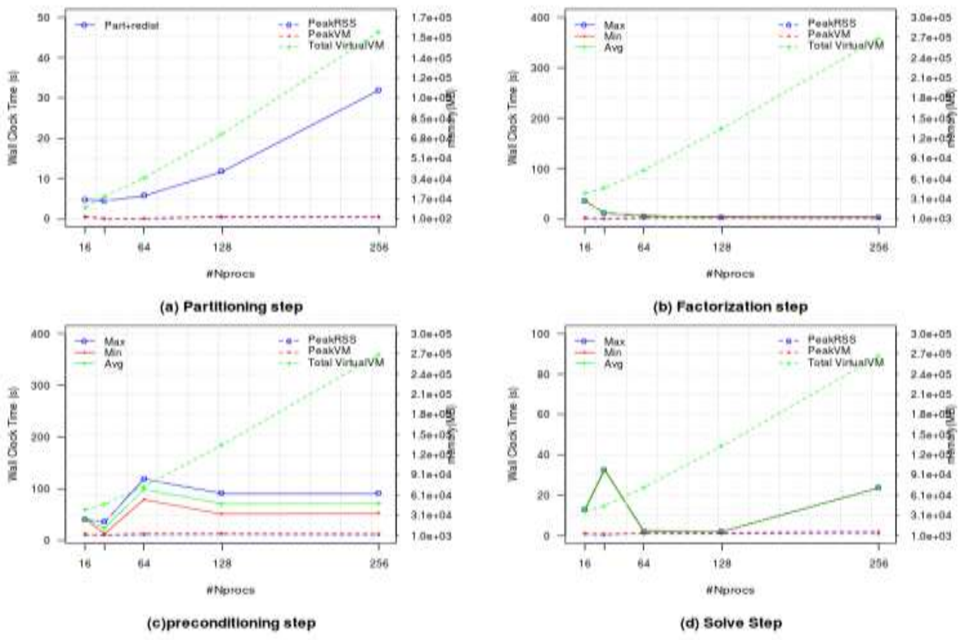


Figure7.8 : Wall clock time and memory usage of PDSLin solver on Audik-1 matrix

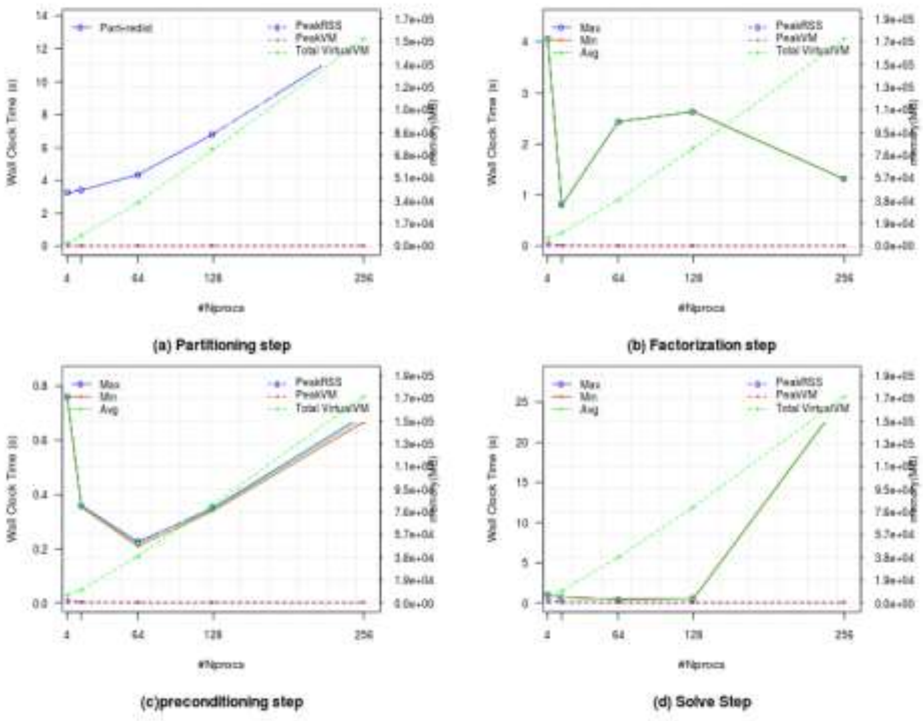


Figure 7.9: Wall clock time and memory usage of PDSLin solver on ASIC680ks matrix

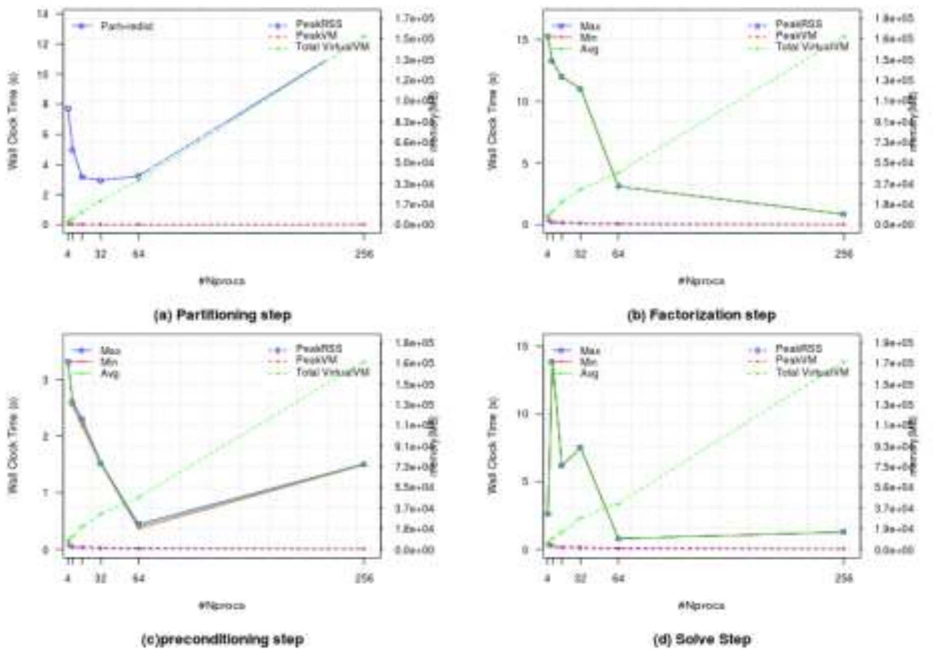


Figure 7.10: Wall clock time and memory usage of PDSLin solver on Freescale1 matrix

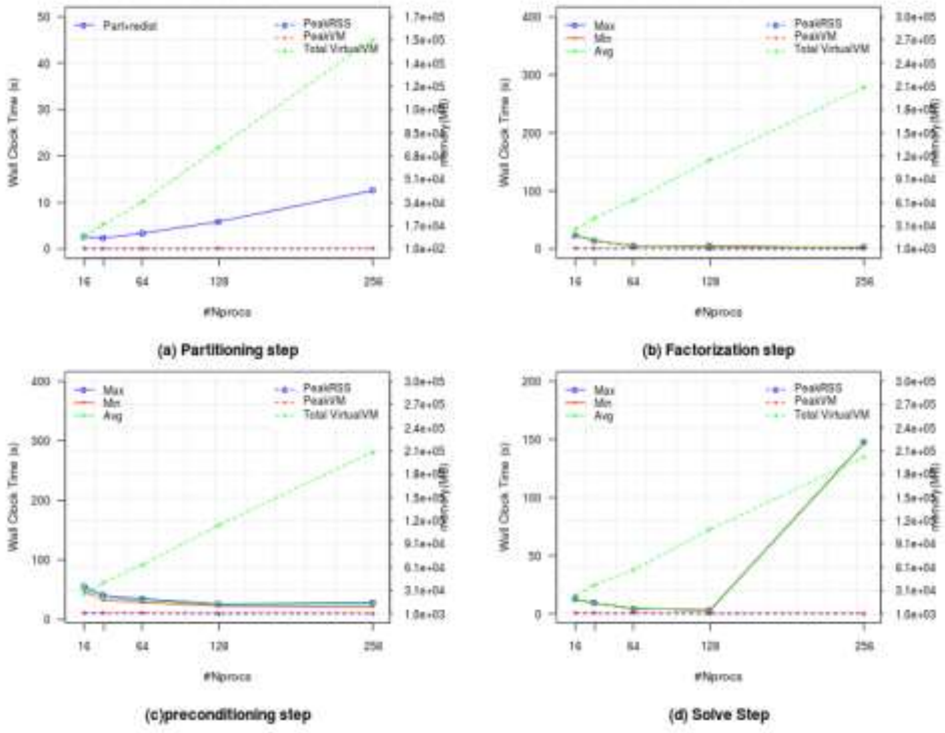


Figure 7.11 : Wall clock time and memory usage of PDSLin solver on Atmosmodl matrix

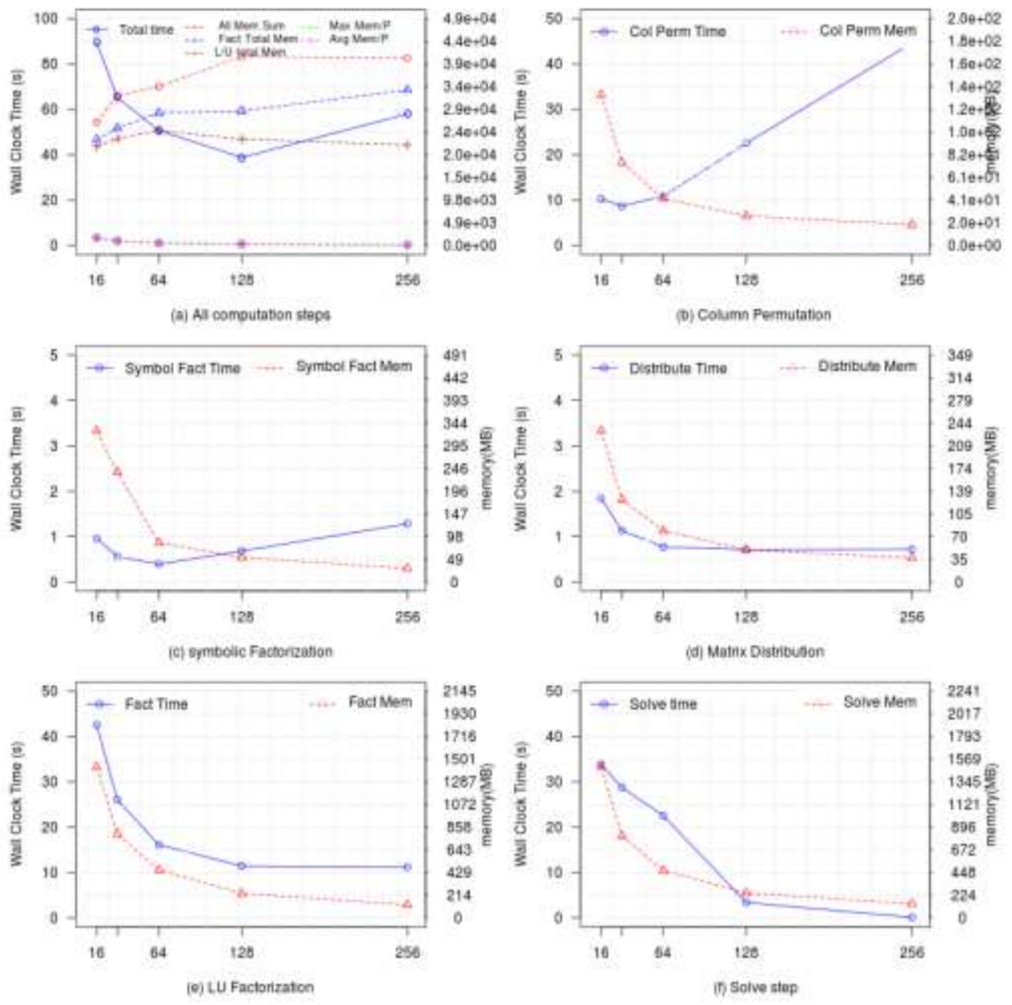


Figure 7.12: Wall clock time and memory usage of Superlu-dist solver on Audik-1 matrix

For atmosmodl matrix, Superlu-dist shows a strange behavour at 128(process grid 8 * 16) processors. With parallel symbolic factorization using Parmetis, it gave a termination failure at 128 processors and the slove time step((f) part of the figure) was large by orders of magnitude. When we changed serial symbolic facotrization on $\boldsymbol{A^T} + \boldsymbol{A}$, the solution time decreased and the solver works with 128 processor but the column permutation, symbolic factorization and total memory consumption are large as shown in Figure(7.12) below.

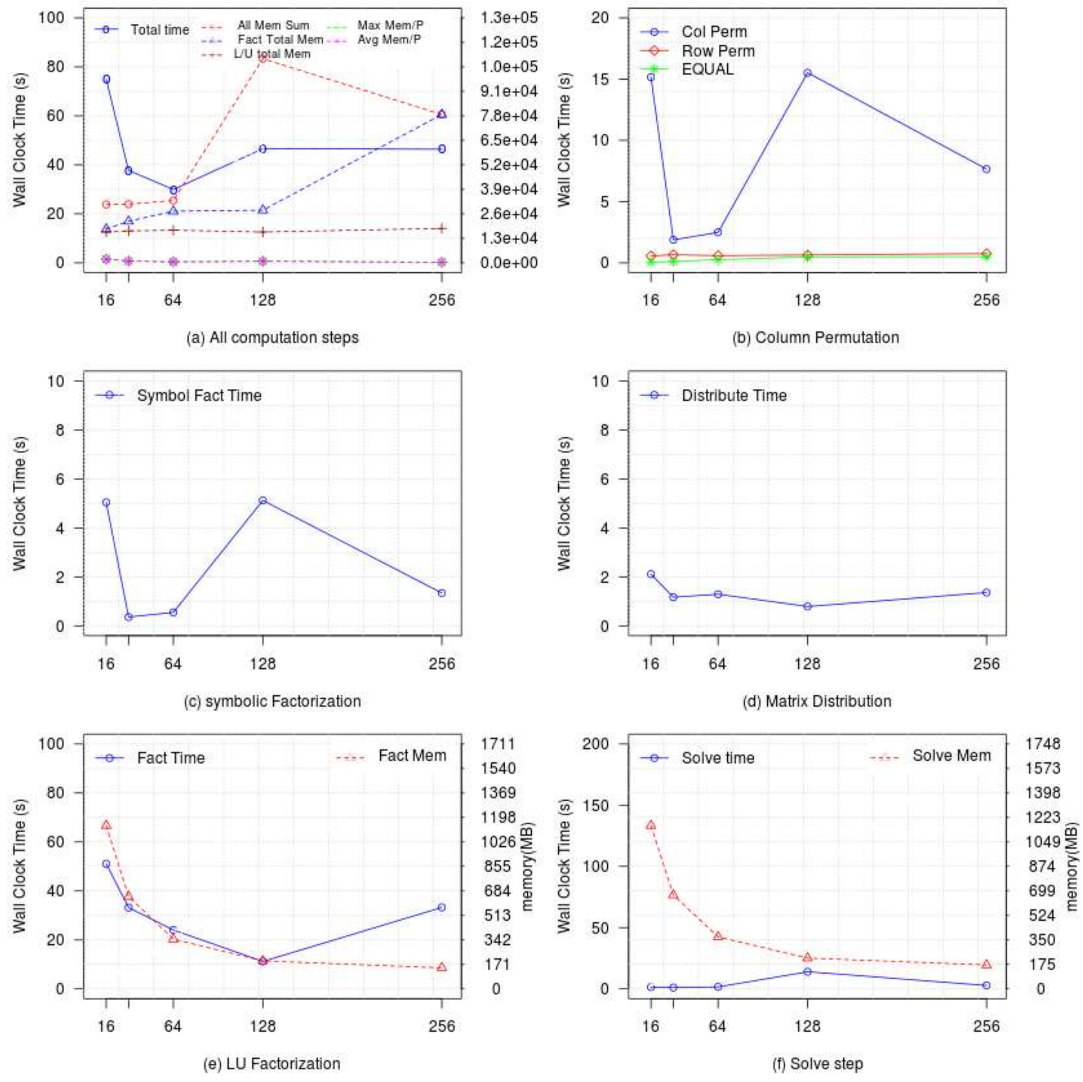


Figure 7.13: Wall clock time and memory usage of Superlu-dist solver on Atmosmodl matrix

8. CONCLUSION AND FUTURE WORK

This is an attempt to address the effect of Sparse hybrid solvers on various matrix types. Our intuition is that it is not easy to evaluate a hybrid solver. There are large number of dependencies for the best hybrid solvers known; Maphys and PDSLin. It is not also easy to compare these solvers fairly due to the large and different number of input values.

Our experience show that there are many failure in the solvers like division by zero due to sparsity or memory address out of range error. The solvers may also behave differently with different matrix format like IJV, HB or MTX even when using the same matrix. This is because the ordering is different. The pure-MPI two level parallelism of PDSLin is more robust than the multithreading of Maphys. Some matrices like ASIC680ks fails with two level parallelism of Maphys. The roots of PDSLin are based on sparse matrix factorization, namely superlu-dist, and the roots of Maphys are based on domain decomposition. This may explain that the factorization time in PDSLin is low whereas the preconditioning in Maphys is low. That is why larger number of subdomains are used in Maphys as input parameter.

Development process in Maphys is progressing and there are active contributions to Maphys. This may also explain the good performance of Maphys in the result section. PDSLin still needs work and there are many bugs in the solver. Superlu-dist is more mature and robust.

We have already considered HIPS and pARMS solvers but because of the limited time, we consider them as a future work. Preconditioners of pARMS like ILU(p,τ) are already implemented in Maphys and the graph partitioning of HIPS(HID) can also be used in PDSLin. Shylu is another hybrid solver and it would be interesting to compare it with these solvers. Besides, one can easily compare these solvers with the Kryolv methods of PETSc. PETSc has a large community of support and active contribution. Its installation and debugging is also easy. There are also many open source direct solvers that are worthwhile to study and compare like MUMPS and Pastix. One can also compile the libraries with 64 bit integer and 64 bit matrix indices and test the solvers with very large matrices. For example we tried to test HV15R matrix but the solvers failed due to matrix index overflow problems.

We believe that there are many ways to improve these solvers such as reducing the dependencies like graph repartitioners or developing a hybrid solver without dependencies at all because dependencies are always constraints since there must be backward compatiblity if not continue developing at the same rate(naming conflicts, etc). On the other hand, dependencies make hybrid solvers work as modules. We can change and choose the suitable package as the problem required. For example, there are two different implementations of LU factorization in Maphys, supernodal using Pastix and Multifronatal using MUMPS. This makes the hybrid solver more flexible for different problem types.

Finally, we created a framework that can extract information automatically from the output files of the solvers, create a database in csv formats, select from these results according to certain queries and send results to R for drawing. The format can draw results related to speedup, solution steps and Schur stages and create tables related to Schur complement matrix. This framework can be further automated by using other tools such as as data analysis, machine learning and simulation tools to suggest the best values of the input parameters and further analysis steps. Here we emphasis on two things: first is the importance of data analysis tools that will greatly enhance the performance of students doing such reseaches. The second thing is the importance of a well organzied input/output files of the solvers so that analysis programs extact information automatically without errors. Maphsy output and input files are well-organzied and information can be extracted easily. This was not the case in PDSLin or any other solver we examined.

REFERENCES

[1] https://www.pardiso-project.org/.

[2**] Schenk, O.; Gärtner, K.; Fichtner, W. BIT** Numerical Mathematics 2000, 40, 158–176.

[3] http://mumps.enseeiht.fr/.

[4**] Ramet Pierre Casadei Astrid, F. M.L. X.** Pastix users guide; tech. rep.; gforge INRIA, November 29, 2013.

[5] http://www.cs.sandia.gov/CRF/aztec1.html.

[6**] Yamazaki, I.; Li, X. S.** On techniques to improve robustness and scalability of the Schur complement method In Proc. VECPAR, 2010, pp 421–34.

[7] **Yamazaki, I.; Li, X. S.; Rouet, F.-H.; Uçar, B.** Combinatorial problems in a parallel hybrid linear solver In Abstracts of 5th SIAM Workshop on Combinatorial Scientific Computing, 2011, pp 87–89.

[8] **Li, X. S.; Shao, M; Yamazaki, I; Ng, E.** Factorization-based sparse solvers and preconditioners In Journal of Physics: Conference Series, 2009; Vol. 180, p 012015.

[9] **Yamazaki, I.; Li, X. S.; Rouet, F.-h.; Uçar, B.** Partitioning, ordering, and load balancing in a hierarchically parallel hybrid linear solver In International Journal of High Performance Computing Applications, 2012.

[10] **Nakov, S**. On the design of sparse hybrid linear solvers for modern parallel architectures., Ph.D. Thesis, Bordeaux, 2015.

[11] **DUFF, I. S.; ERISMAN, A. M.; REID, K.,** Direct Methods for Sparse Matrices; Oxford University Press: 2017; Vol. 72.

[12] **Vuduc, R. W.; Demmel, J. W.,** Automatic performance tuning of sparse matrix kernels; University of California, Berkeley: 2003; Vol. 1.

[13] **Duff, I. S.; Grimes, R. G.; Lewis, J. G.** ACM Transactions on Mathematical Software (TOMS) 1989, 15, 1–14.

[14] **Saad, Y.,** Iterative methods for sparse linear systems; SIAM: 2003; Vol. 82.

[15] **Schloegel, K.; Karypis, G.; Kumar, V.,** Graph partitioning for high performance scientific simulations; Army High Performance Computing Research Center: 2000.

[16] **Karypis, G.; Kumar, V**. A fast and high quality multilevel scheme for partitioning irregular graphs SIAM Journal on scientific Computing 1998, 20, 359–392.

[17] **Yannakakis, M**. Computing the minimum fill-in is NP-complete. SIAM Journal on Algebraic Discrete Methods 1981, 2, 77–79.

[18] **Duff, I. S.; Uçar, B**. Combinatorial problems in solving linear systems. Combinatorial Scientific Computing 2009.

[19] **Duff, I. S.; Erisman, A. M.; Reid, J. K.,** Direct methods for sparse matrices; Oxford University Press: 2017.

[20] **George, A.; Ng, E**. An implementation of Gaussian elimination with partial pivoting for sparse systems. SIAM journal on scientific and statistical computing 1985, 6, 390–409.

[21] **Markowitz, H. M.** The elimination form of the inverse and its application to linear programming Management Science 1957, 3, 255–269.

[22] **Amestoy, P. R.; Davis, T. A.; Duff, I. S**. An approximate minimum degree ordering algorithm. SIAM Journal on Matrix Analysis and Applications 1996, 17, 886–905.

[23] **George, A**. Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis 1973, 10, 345–363.

[24] **Heath, M. T.; Ng, E.; Peyton, B.W**. Parallel algorithms for sparse linear systems. SIAM review 1991, 33, 420–460.

[25] **Demmel, J. W**. SuperLU users' guide; tech. rep.; 1999,2016.

[26] **Wilkinson, J. H.** Error analysis of direct methods of matrix inversion. Journal of the ACM (JACM) 1961, 8, 281–330.

[27] **Mary, T**. Block low-rank multifrontal solvers: complexity, performance, and scalability., Ph.D. Thesis, Ph. D. dissertation, Toulouse University, Toulouse, France, 2017.

[28] **Celebi, M. S.; Duran, A.; Tuncel, M**.; **Akaydin**, **B** Scalable and improved SuperLU on GPU for heterogeneous systems. PRACE PN 2012, 283493.

[29] **Duff, I. S.; Reid, J. K.** The multifrontal solution of indefinite sparse symmetric linear. ACM Transactions on Mathematical Software (TOMS) 1983, 9, 302–325.

[30] **Zitney, S. E.** Sparse matrix methods for chemical process separation calculations on supercomputers. In Proceedings of the 1992 ACM/IEEE conference on Supercomputing, 1992, pp 414–423.

[31] **Duff, I.; Uçar, B**. Scholarpedia 2013, 8, revision #153309, 9700.

[32**] Li, X. S.** Sparse Gaussian elimination on high performance computers; tech. rep.; California University, Graduate Dev, 1996.

[33] **Li, X** In http://crd. lbl. gov/˜ xiaoye/SuperLU/SparseDirectSurvey. pdf, 2006.

[34] **Duran, A.; Celebi, M. S.; Tuncel, M.; Oztoprak, F**. In Advances in Applied Mathematics; Springer: 2014, pp 153–160.

[35] **Duff, I. S.; Koster, J.** The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM Journal on Matrix Analysis and Applications 1999, 20,889–901.

[36] **Benzi, M.; Tuma, M**. A robust preconditioner with low memory requirements for large sparse least squares problems SIAM Journal on Scientific Computing 2003, 25, 499–512.

[37] **Celebi, M. S.; Duran, A.; Oztoprak, F.; Tuncel, M.; Akaydin, B**. On the improvement of a scalable sparse direct solver for unsymmetrical linear equations. The Journal of Supercomputing 2017, 73, 1852–1904.

[38] **Celebi, M. S.; Duran, A.; Tuncel, M.; Akaydına, B.; Oztoprak, F**. Performance Analysis of BLAS Libraries in SuperLU_DIST for SuperLU_MCDT (Multi Core Distributed) Development. 2013.

[39] **Barrett, R.; Berry, M. W.; Chan, T. F.; Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; Van der Vorst, H.,** Templates for the solution of linear systems: building blocks for iterative methods; Siam: 1994; Vol. 43.

[40] **Haidar, A**. On the parallel scalability of hybrid linear solvers for large 3D problems., Ph.D. Thesis, Institut National Polytechnique de Toulouse-INPT, 2008.

[41] http://portal.nersc.gov/project/sparse/pdslin/.

[42] https://www.labri.fr/perso/pelegrin/scotch/.

[43] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[44] **Li, X. S.; Shao, M; Yamazaki, I; Ng, E**. Pdslin user's Guide 1.2; tech. rep.; 2011.

[45] http://crd-legacy.lbl.gov/~xiaoye/SuperLU/.

[46**] L. Giraud, R. S. T.** Algebraic Domain Decomposition Preconditioners, Technical Report ENSEEIHT-IRIT RT 2006, 24.

[47] https://gitlab.inria.fr/solverstack/maphys.

[48] https://pastix.gforge.inria.fr/files/README-txt.html.

[49] **Agullo, E.; Giraud, L.; Zounon, M. On the resilience of parallel sparse hybrid solvers,** In High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on, 2015, pp 75–84.

[50] **Carvalho, L. M.; Giraud, L.; Meurant, G**. Numerical linear algebra with applications 2001, 8, 207–227.

[51] **Agullo, E.; Giraud, L.; Zounon, M**. Maphys user's guide 0.9.5.0; tech. rep. 3; Mar. 2015, pp 1–47.

[52] https://www.open-mpi.org/projects/hwloc/.

[53] **Hénon, P.; Saad, Y.** A parallel multistage ILU factorization based on a hierarchical graph decomposition, SIAM Journal on Scientific Computing 2006, 28, 2266–2293.

[54] **Barth, T. J.; Griebel, M.; Keyes, D. E.; Nieminen, R. M.; Roose, D.; Schlick, T.,** Lecture Notes in Computational Science and Engineering; Springer: 2005; Vol. 49.

[55] http://www-users.cs.umn.edu/~saad/software/pARMS/.

[56] **Saad, Y.; Suchomel, B.,** ARMS: An Algebraic Recursive Multilevel Solver for general sparse linear systems; Numer. Linear Alg. Appl: 1999.

[57] http://abcd.enseeiht.fr/.

[58] https://github.com/zeapo/abcd/blob/master/doc/old_doxygen / installation.md.

[59] **Arioli Mario; Duff, I. N.-J.R. D.** A Block Projection Method for Sparse Matrices, SIAM Journal on Scientific and Statistical Computing Jan. 1992, 13.

[60] **Duff Iain S.; Guivarch, R. R.D.Z. M.** The Augmented Block Cimmino Distributed Method, SIAM Journal on Scientific Computing Jan. 2015, 37.

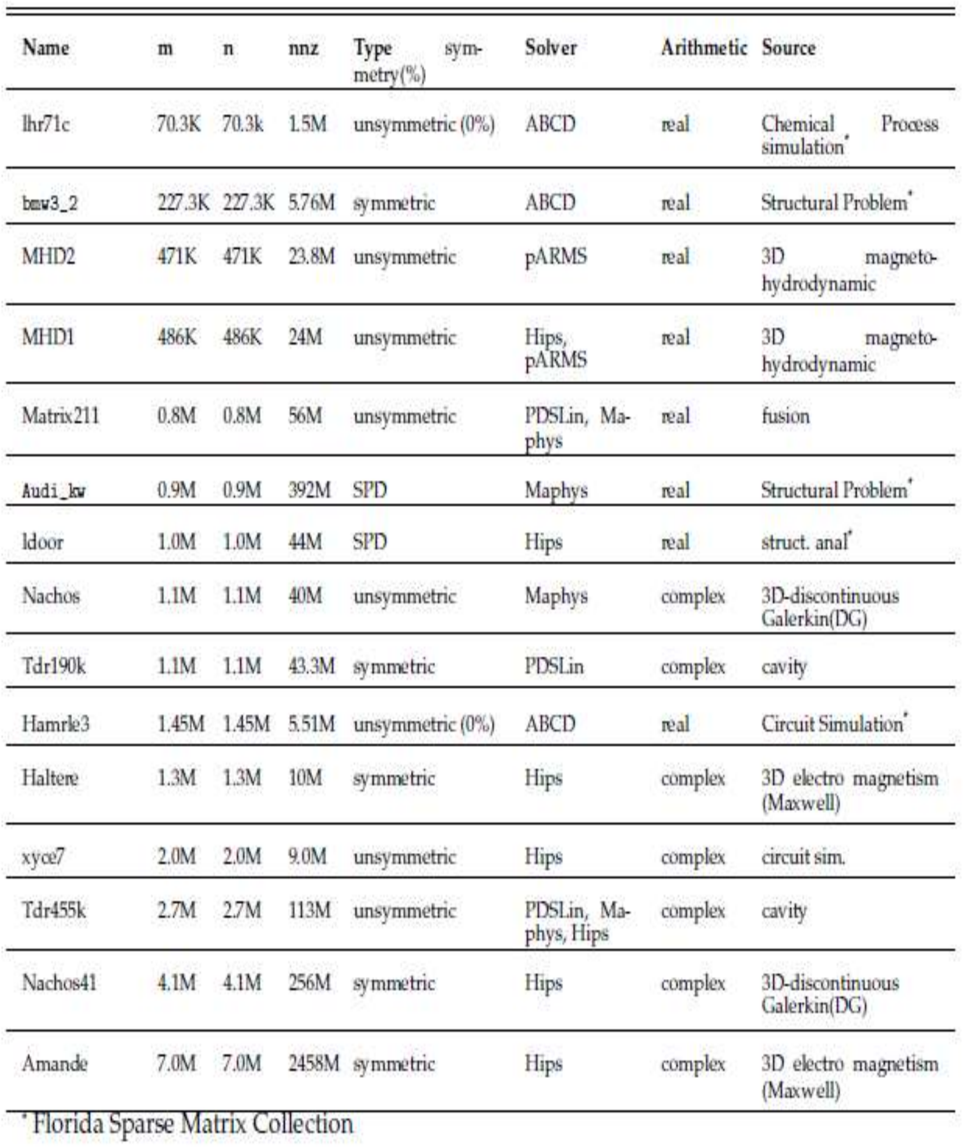
[61] **Gupta, A**. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations, ACM Transactions on Mathematical Software (TOMS) 2002, 28, 301–324.

[62] https://www.mathworks.com/help/matlab/ref/condest.html.

APPENDICES

APPENDIX A: Matrices and Package Versions

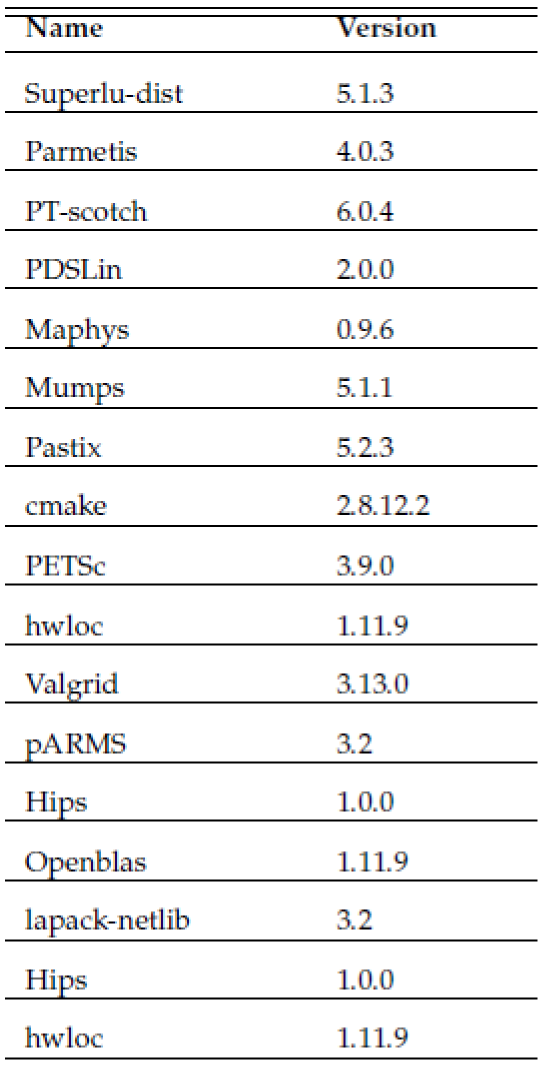
Table A.1 : List of Matrices used in Literature for Evaluating Hybrid Solvers



Name	m	n	nnz	Type symmetry(%)	Solver	Arithmetic	Source
lhr71c	70.3K	70.3k	1.5M	unsymmetric (0%)	ABCD	real	Chemical Process simulation*
bmw3_2	227.3K	227.3K	5.76M	symmetric	ABCD	real	Structural Problem*
MHD2	471K	471K	23.8M	unsymmetric	pARMS	real	3D magneto-hydrodynamic
MHD1	486K	486K	24M	unsymmetric	Hips, pARMS	real	3D magneto-hydrodynamic
Matrix211	0.8M	0.8M	56M	unsymmetric	PDSLin, Maphys	real	fusion
Audi_kw	0.9M	0.9M	392M	SPD	Maphys	real	Structural Problem*
ldoor	1.0M	1.0M	44M	SPD	Hips	real	struct. anal*
Nachos	1.1M	1.1M	40M	unsymmetric	Maphys	complex	3D-discontinuous Galerkin(DG)
Tdr190k	1.1M	1.1M	43.3M	symmetric	PDSLin	complex	cavity
Hamrle3	1.45M	1.45M	5.51M	unsymmetric (0%)	ABCD	real	Circuit Simulation*
Haltere	1.3M	1.3M	10M	symmetric	Hips	complex	3D electro magnetism (Maxwell)
xyce7	2.0M	2.0M	9.0M	unsymmetric	Hips	complex	circuit sim.
Tdr455k	2.7M	2.7M	113M	unsymmetric	PDSLin, Maphys, Hips	complex	cavity
Nachos41	4.1M	4.1M	256M	symmetric	Hips	complex	3D-discontinuous Galerkin(DG)
Amande	7.0M	7.0M	2458M	symmetric	Hips	complex	3D electro magnetism (Maxwell)

* Florida Sparse Matrix Collection

Table A.2 : Versions of libraries we have considered in this work



Name	Version
Superlu-dist	5.1.3
Parmetis	4.0.3
PT-scotch	6.0.4
PDSLin	2.0.0
Maphys	0.9.6
Mumps	5.1.1
Pastix	5.2.3
cmake	2.8.12.2
PETSc	3.9.0
hwloc	1.11.9
Valgrid	3.13.0
pARMS	3.2
Hips	1.0.0
Openblas	1.11.9
lapack-netlib	3.2
Hips	1.0.0
hwloc	1.11.9

APPENDIX B: Dependecies of Hybrid Solvers

Note: Design is quoted from Spack developers

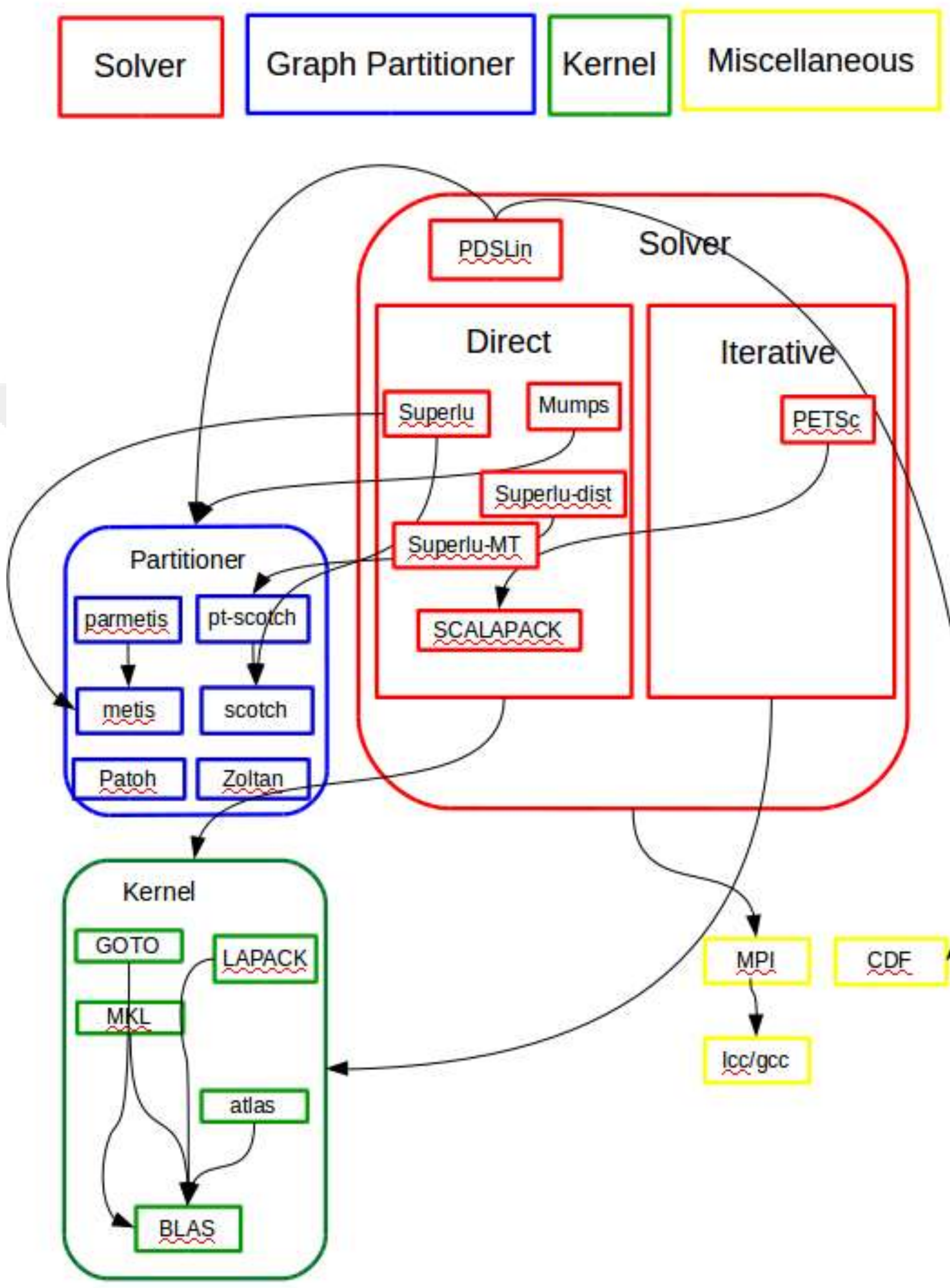


Figure A.1 : PDSLin Hybrid Solver Installation Dependencies

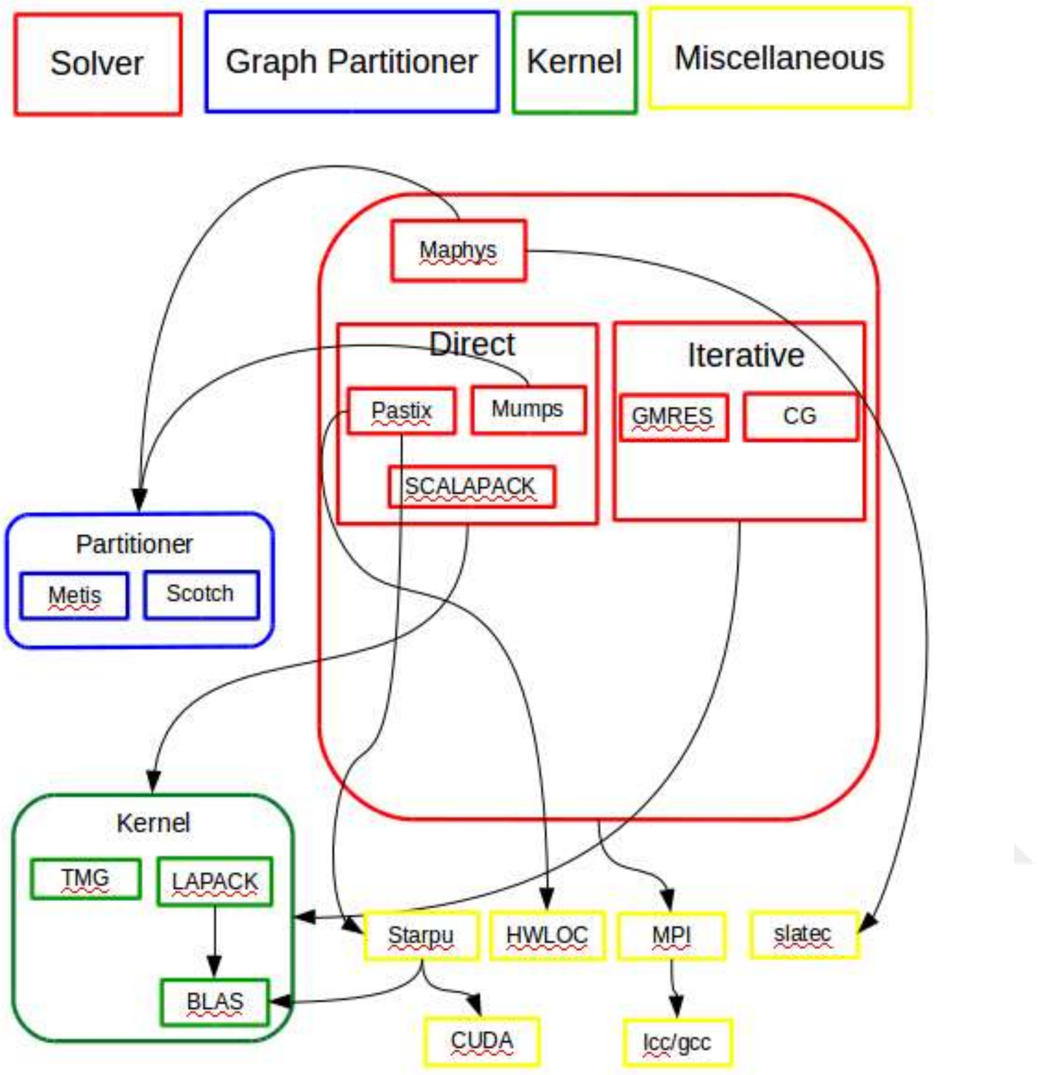


Figure A.2 : Maphys Hybrid Solver Installation Dependencies

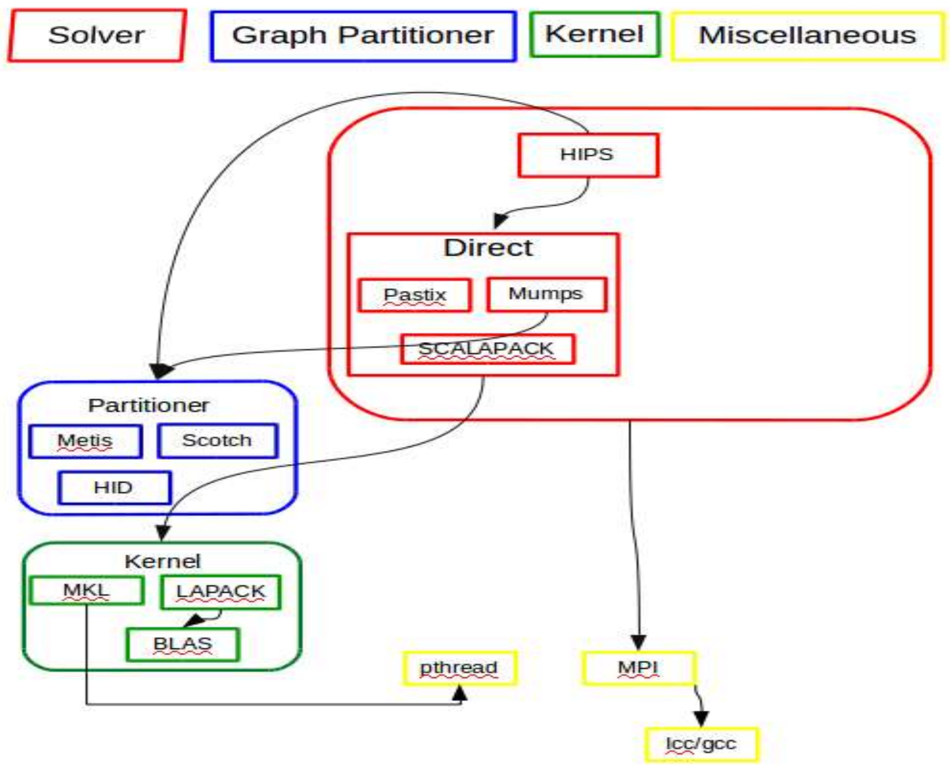


Figure A.3 : HIPS Hybrid Solver Installation Dependencies

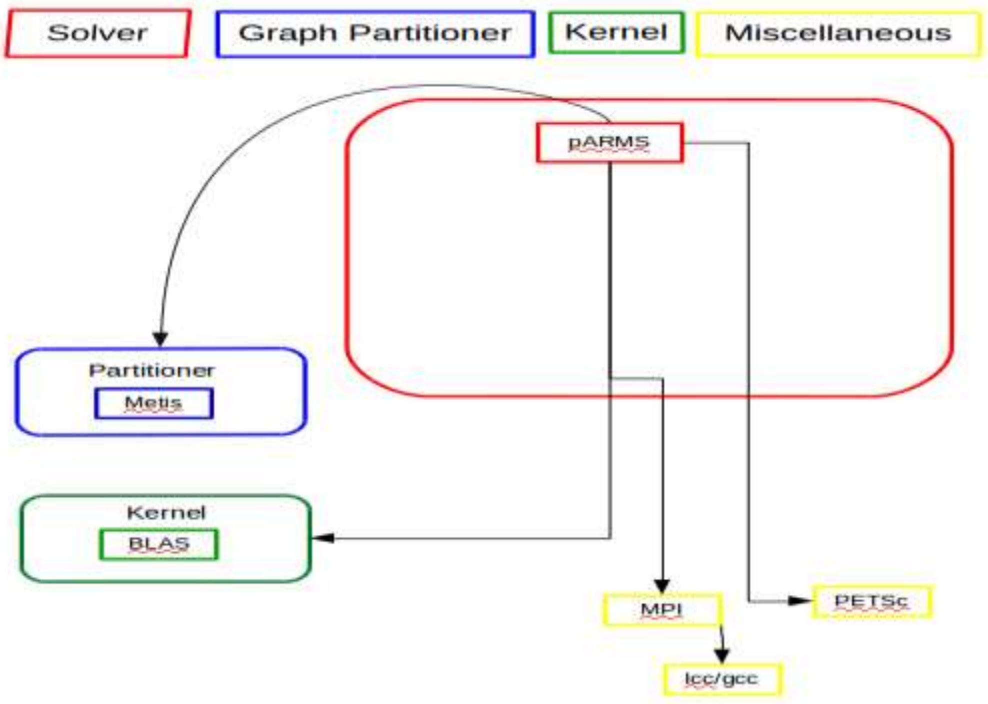


Figure A.4 : pARMS Hybrid Solver Installation Dependencies

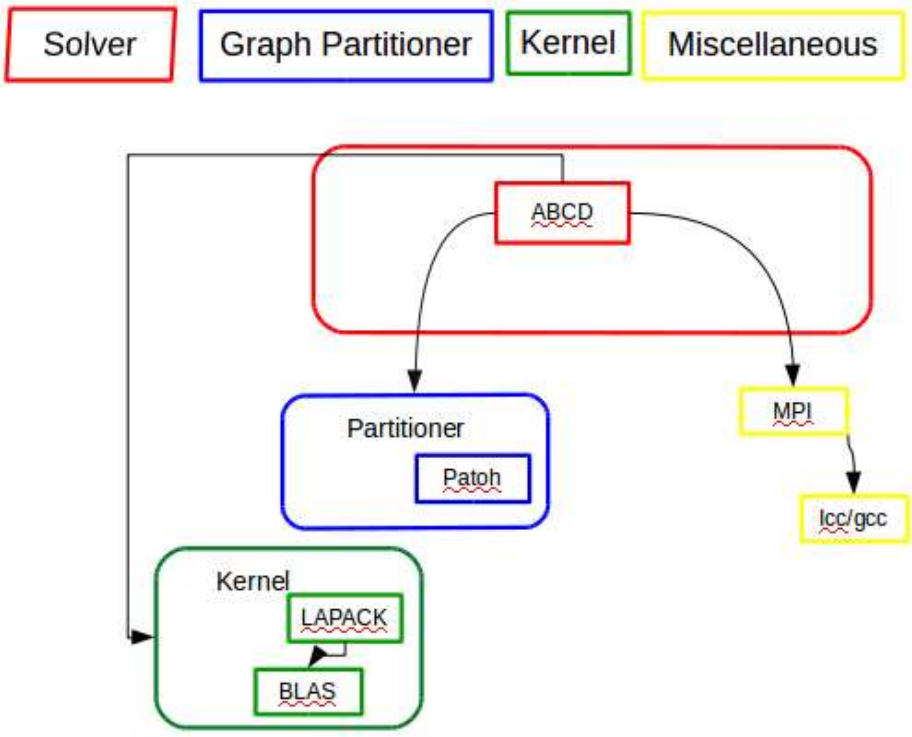


Figure A.5 : ABCD Hybrid Solver Installation Dependencies

APPENDIX C: Hybrid Solvers Input Parameters

Table C.1: Maphys Input Parameters

MATFILE	=		audikw-1.mtx
SYM = 1	=	1	#0 (General) 1 (SPD) 2 (symmetric)
ICNTL(1)	=	1	#Controls where to write error messages. def:0 stderr
ICNTL(2)	=	1	1 #Controls where to write warning messages def:0 stderr
ICNTL(3)	=	6	# where are written statistics messages def: 6:stdout
ICNTL(4)	=	5	#Controls where to write stat.msg 1-4,def:print errors,warnings &detailled statistics:3-> 5:print every thing
ICNTL(5)	=	1	#Controls when to print list of controls (Xcntl).Default:0.never print.1:begining,2:each step
ICNTL(6)	=	1	#Controls when to print list of informations (Xinfo).default:0:Never print.1:begining,2:each step
ICNTL(7)	=	0	#Partitioning strategy (1:METIS-NODEND 2:METIS-EDGEND 3:METIS-NODEWND 4:SCOTCH-CUSTOM)old value :4
ICNTL(8)	=	50	#level of filling for L and U in the ILUT method.Default:-1.imp if ICNTL(30)->2
ICNTL(9)	=	50	#level of filling for the Schur complement in the ILUT ICNTL(10) = 0
ICNTL(10)	=	0	
ICNTL(11)	=	0	
ICNTL(12)	=	0	
ICNTL(13)	=	2	#(P,MT:2)fact. & the the precd. direct solver.1:mumps.2:pastix.3:Use multiple sparse direct solvers.see ICNTL(15,32)
ICNTL(14)	=	0	#Output format.Default : 0 stdout,1:emak
ICNTL(15)	=	2	#(P,MT:2)direct solver for preconditioner1 :mumps.2:pastix.3:multiple.see ICNTL(13,32)
ICNTL(15)	=	2	
ICNTL(16)	=	0	
ICNTL(17)	=	0	
ICNTL(18)	=	0	
ICNTL(19)	=	1	
ICNTL(20)	=	1	#(P)3rd party iterative solver0:unset.1:gmres.2:CG.3:automatic.def:3
ICNTL(21)	=	2	#(imp) prcndtr strtgy.# (1:local DENSE 2:local sparse.3:From ILUT based.4: No preconditioner) values:1,2,3,4,5,10
ICNTL(22)	=	2	#(P)Controls the iterative solver. 0:modGS, 1:iter.selGS,2classicalGS, 3:iter GS. def:3
ICNTL(23)	=	0	#Controls whether the user wishes to supply an initial guess.0:no,1:yes. def:0
ICNTL(24)	=	1000	#(P) Iterative Solver - Maximum number of itrs.for difficult problems ->7k
ICNTL(25)	=	0	#strategy to compute residual. 0:recurrent,1:residual.->irrelevant when iterative solver is CG (ICNTL(20) = 2,3).
ICNTL(26)	=	500	#(P)Iterative Solver - GMRES: restart every X iterations. ignored if solver is CG (ICNTL(20) = 2,3 with SPD
ICNTL(27)	=	0	# Iterative Solver SCHUR Complement Matrix/Vector product.# (1:EXPLICIT 2:IMPLICIT)
ICNTL(28)	=	1	#(P)scaled residual is computed.def:1->similar to pdslin
ICNTL(29)	=	1	#mode of the iterative solver FABULOUS.def:1
ICNTL(30)	=	0	#how to fnd schur its approx.def:0:shur retrnd by sparse drct slvr package->2:Sparse approx.bsd on partl ILU(t;p)shld set ICNTL(8,9),RCNTL(8,9)
ICNTL(31)	=	50	
ICNTL(32)	=	2	#(P,MT:2)direct solver for local schur factorisation.def:ICNTL(13),see ICNTL(13,15)
ICNTL(33)	=	10	#Number of eigenvalues per subdomain.def:10.Ignored ifICNTL(21)=10.
ICNTL(34)	=		#cnvrgnc hstry of itrtve slvr is wrtn a file.def:0 regular output.1->file is named gmrescvgN.dat or cgcvgN.dat
ICNTL(35)	=	0	
ICNTL(36)	=	2	#How to bind thread inside MAPHYS.0 :nobind.1:Thread to core bind.2:Grouped bind.def:0.eg:smph_examplethread.
ICNTL(37)	=	16	#(P)2level parlsm,pecifies the number of nodes.only useful if ICNTL(42) > 0.def:1
ICNTL(38)	=	28	#(P)2level parlsm,spcfs # cores per node.only useful if ICNTL(42) > 0.def:1
ICNTL(39)	=	16	#(P)2level parlsm,spcfs # threads per domains.only useful if ICNTL(42) > 0.def:1
ICNTL(40)	=	16	

```
ICNTL(41)    =   0            #(P)2level parlsm,spcfs # domains.only useful if ICNTL(42) > 0.def:1
ICNTL(42)    =   0             #(P)2level,0->mpi only,1:multitheading-> level of parallelism def:0,1-
                              >multithreading.shld set
ICNTL(43)    =   1             #input system (central.on the host, distributed,...)def:1.eg:smph_examplerestart-
                              >3.paddle->2(experimental).
ICNTL(44)    =   0            #When activated, it means user permutation you want MAPHYS to use.def:0.eg:
                              xmph_exampledistkv in examples
ICNTL(45)    =   0            #local output after analysis. If set to 1, it allows to perform a dump of the local
                              matrices. def:0
ICNTL(46)    =   0
ICNTL(47)    =   20           #Crls the MUMPS instance.def:20.
ICNTL(48)    =   10           #Crls FABULOUS Deflated Restart algorithm. de:20
ICNTL(49)    =   1            #crls domain decp lbry/algrthm.Warning: Modifies the behavior of
                              ICNTL(43).def:1: maphys.2:paddle
ICNTL(50)    =   0            #When MUMPS error indicates that requires more memory workspace, def:0
RCNTL(1)     =   0.000E+00
RCNTL(2)     =   0.000E+00    #is the target for FABULOUS Deflated Restart algorithm.def:0.0
RCNTL(3)     =   0.000E+00    # sets the value of alpha for custom stopping criteria of GMRes and
                              CG(ICNTL(28) = 2).def:0
RCNTL(4)     =   0.000E+00    #sets the value of beta for custom stoppingcriteria of GMRes and CG (ICNTL(28)
                              = 2). def:0
RCNTL(5)     =   2.000E+00    #mumps:gives the amount by which the extra workspace(initially given by
                              ICNTL(47)) will be multiplied for the next try. def:2.0
RCNTL(6)     =   0.000E+00
RCNTL(7)     =   0.000E+00
RCNTL(8)     =   1.e-02       #thrsld used to sparsify the LU factors while using PILUT.def:0.0.imp if
                              ICNTL(30)->2
RCNTL(9)     =   1.e-02       #thrsld used to sparsify the schur compl. While computing it with
                              PILUT.def:0.0.imp if ICNTL(30)->2
RCNTL(10)    =   0.000E+00
RCNTL(11)    =   1.0e-4       #(imp) Preconditioner - local SPARSE – Sparsifying tolerance(imp. if
                              ICNTL(21)->2)
RCNTL(12)    =   0.000E+00
RCNTL(13)    =   0.000E+00
RCNTL(14)    =   0.000E+00
RCNTL(15)    =   2.000E-01    #Specifies the imbalance tolerance used in Scotch partitioner to create the
                              subdomains. def:0.2
RCNTL(16)    =   0.000E+00
RCNTL(17)    =   0.000E+00
RCNTL(18)    =   0.000E+00
RCNTL(19)    =   0.000E+00
RCNTL(20)    =   0.000E+00
RCNTL(21)    =   1e-12        #(P) Iterative solver - convergence criteria. see ICNTL(28).
```

Table C.2: PDSLin Input Parameters

```
4                    # nproc_schur : number of processors on schur complement
4                   # nproc_dcomp : number of processors to compute partitoning, (>=num_doms) INPUT_HB
INPUT_HB            # input_type : input matrix file format (INPUT_IJV, INPUT_HB, or INPUT_BIN)
                    SYMMETRIC
UNSYMMETRIC         # mat_type : input matrix type (SYMMETRIC or UNSYMMETRIC) SYMMETRIC
UNSYMMETRIC         # mat_pattern : input matrix type (SYMMETRIC or UNSYMMETRIC) UNSYMMETRIC
UNSYMMETRIC         #SHUR_MATRIX_pattern : input matrix type (SYMMETRIC or UNSYMMETRIC)
NO                  # remove_zeros : YES if matrix file contains zero elements
SCOTCH              # dcomp_type : how partitioning will be computed (SCOTCH or HIPS)
8                   # num_doms : number of domains to be extracted. --(0: iteraitve solver, 1: direct solver)
0e-3                # tol_sub(tau_sub) : ILU threshold for subdomains (used in serial superlu).
SLU_DIST            # dom_solver : interior domain solver (SLU/SLU_DIST/MUMPS)
1000                # dom_size : interior domain size
10                  # blk_size : block size for the interface solves
YES                 # pperm_schur : parallel parmutation for schur Complement
YES                 # psymb_schur : parallel symbolic factorization for schur complement
5                   # equil_schur : equilbration for schur complement
20                  # relax_factor : compression factor to compute nested diss. of schur complement
```

5	# equil_dom: equilbrtn fr intror doms used only by serial SuperLU(if 0: no equi or row perm;equil_schur ->0) domains(METIS,PARMETIS,NATURAL,MMD_ATA,MMD_AT_PLUS_A,COLAMD)
PARMETIS	# perm_dom : permutation of interior
YES	# hybrid_fgmres->use FGMRES as outer-loop, and hybrid solver as preconditioner
1e-12	# residual_tol : stopping criteria for outer-loop 50 # outer_max
0.0	# dtol: diag.thrcld fr serl SuperLU(!=0->rplc tiny pivots;=0->acrcy high.diag.pert.is turned off hgir sol-acrcy mb obtnd usng iterative efinements)
1e-6	# tol0: drop tolerance for F (L^{-1} E)-> threshold for G
0.0	# tol1 : drop tolerance for W (E*F) ->threshold for T
1.0e-5	# tol2 : drop tolerance for schur complement
0.0	# tol3 : drop tolerance for ILUT(used by petsc)
-1	# ilu_lvl : level threshold for ILU(if this>=0 ->psymp_schur=no:using serial symbolic for ILU)
1	# asm_ovlp:/* overlap and # of doms per processor for ASM */
1	# asm_nsub BICGSTAB #(M) itsolver : iterative solver for schur complement (GMRES, BICGSTAB, FGMRES, TFQMR)
1000	#(M) itrs : maximum number of matrix operations
500	#(M)restart : number of operations at restart
1e-12	#(M) residual_tol : stopping criteria for iterative method
CLASSICAL	#(M) orth : orthogonalization scheme for GMRES (CLASSSICAL ,MODIFIED)
PR_SLU_ILU	# prcnd_type:type of prcnditner (PR_PETSC_ILUK ,PR_PETSC_ILUT PR_PHIDAL_ILU,PR_LSCHUR_ASM)
0.0	0.0 # patoh_lbound :
0.4	0.4 # patoh_ubound :
0.1	0.1 # patoh_sparsify

Table C.3: HIPS Input Parameters

D	0	# D 0 HIPS_SYMMETRIC = [0:nonsymmetirc, 2: symmetic, 1:symmetric pattern]
D	4	# D 1 HIPS_VERBOSE = [0-5]
D	0	# D 2 HIPS_SCALE = reserved for develpoers
D	100	# D 3 HIPS_LOCALLY = [0: no-fillin 100 : allow fillin anywhere]
D	1	#D 4 HIPS_KRYLOV_RESTART = restart parameter of GMRES
D	1	#D 5 HIPS_ITMAX = maximum number of iteration allowed in the krylov method
D	0	#D 6 HIPS_FORWARD = DHIPS developers reserved.
D	0	#D 7 HIPS_SCHUR_METHOD =DD HIPS developers reserved.
D	150	#D 8 HIPS_ITMAX_SCHUR =
D	3	#D 9 HIPS_PARTITION_TYPE = HIPS developers reserved.
D	0	#D 10 HIPS_KRYLOV_METHOD = 0 Preconditioned GMRES, =1 Preconditioned CG.
D	4	#D 11 HIPS_DOMSIZE =
D	1	#D 12 HIPS_SMOOTH_ITER_RATIO = HIPS developers reserved.
D	1	#D 13 HIPS_DOMNBR = fff
D	1	#D 14 HIPS_REORDER D=# 0 No reordering inside the subdomain to minimize fill-in, =1 reordering inside the subdomain to minimize fill-in. This option is only used in the recursive ITERATIVE preconditioneur.
D	0	#D 15 HIPS_SCALENBR =[1-] this value is used to set the number of time the normalisation is applied to the matrix. One should set a value >1 only in special case.
D	0	#D 16 HIPS_MASTER = the master process, zero is the default.
D	0	#D 17 HIPS_COARSE_GRID =#HIPS developers reserved.
D	1	#D 18 HIPS_CHECK_GRAPH = [0, 1] (default 1) : set this option to 1 if you want to check (and repair) the matrix adjacency graph. This option ensures the graph is symmetric and that there is no double edge.
D	1	#D 19 HIPS_CHECK_MATRIX = [0, 1] (default 1) : set this option to 1 if you want to check (and repair) the coefficients matrix. This option check if there are coefficient with the same indices and sum them up in this case.
D	1	#D 20 HIPS_DUMP_CSR =# HIPS developers reserved.
D	0	#D 21 HIPS_IMPROVE_PARTITION =# HIPS developers reserved.
D	0	#D 22 HIPS_TAGNBR =# HIPS developers reserved.
D	0	#D 23 HIPS_SHIFT_DIAG = HIPS developers reserved.
D	0	#D 24 HIPS_GRAPH_SYM = set this option to 0, if
D	0	#D 25 HIPS_GRID_DIM = HIPS developers reserved. you are sure that the graph you give to HIPS is symmetric ;

```
                  this disable the graph symmetrization
D       0          #D 26 HIPS_GRID_3D = HIPS developers reserved.
D       0          #D 27 HIPS_DISABLE_PRECOND = [0, 1] if set to 1 then when new matrix coefficient are entered
                     the preconditioner is not recalculated in HIPS.Nevertheless, this option is taken into account only if a
                     preconditioner has already been computed.
D       1          #D 28 HIPS_FORTRAN_NUMBERING =[0, 1] (default 1) : numbering in indexes array will start at
                     0 or 1. This options modify the default numbering for the inputs and returns in all HIPS's functions
D       0         #D 29 HIPS_DOF =[1-] (default 1) : number of unknowns per node in the matrix non-zero pattern
                     graph(degree of freedom).
D       0         D 0 #HIPS_PIVOTING =[0, 1] (default 0) : disable or enable column pivoting in ILUT (only for
                     unsymmetric matrix).This
                  option is not yet fully implemented in parallel.
R       0.0        #R 0 HIPS_PREC = Wanted norm error at the end of solve.
R       0.0        #R 1 HIPS_DROPTOL0 =
R       0.005      #R 2 HIPS_DROPTOL1 =
R       0.0        #R 3 HIPS_DROPSCHUR = fff
R       0.0        #R 4 HIPS_DROPTOLE =
R       0.0        #R 5 HIPS_AMALG = fff
```

CURRICULUM VITAE

Name Surname	**: Afrah Farea**
E-Mail	**: Afrah.nacib@gmail.com**

EDUCATION **:**

- **B.Sc.** **:** 2013,Taiz University,Enineering faculgty, Software Enineering Department

PROFESSIONAL EXPERIENCE AND REWARDS:

- On the Evaluation of Sparse Hybrid Solvers (not pubished yet)