ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

DESIGNING A FAST DIRECT SPARSE MATRIX SOLVER FOR MULTI-CORE DISTRIBUTED SYSTEMS

M.Sc. THESIS

Mehmet TUNÇEL

Computational Science and Engineering

Computational Science and Engineering Program

MAY 2013

ISTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

DESIGNING A FAST DIRECT SPARSE MATRIX SOLVER FOR MULTI-CORE DISTRIBUTED SYSTEMS

M.Sc. THESIS

Mehmet TUNÇEL
(702101006)

Computational Science and Engineering

Computational Science and Engineering Program

Thesis Advisor: Prof. Dr. M. Serdar ÇELEBİ

MAY 2013

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

ÇOK ÇEKİRDEKLİ DAĞITIK SİSTEMLER İÇİN HIZLI DOĞRUDAN SEYREK MATRİS ÇÖZÜCÜ TASARLANMASI

YÜKSEK LİSANS TEZİ

Mehmet TUNÇEL
(702101006)

Hesaplamalı Bilim ve Mühendislik

Hesaplamalı Bilim ve Mühendislik Programı

Tez Danışmanı: Prof. Dr. M. Serdar ÇELEBİ

MAYIS 2013

Mehmet TUNÇEL, a M.Sc. student of ITU Informatics Institute 702101006 successfully defended the thesis entitled **"DESIGNING A FAST DIRECT SPARSE MATRIX SOLVER FOR MULTI-CORE DISTRIBUTED SYSTEMS"**, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. M. Serdar ÇELEBİ** Istanbul Technical University	.....
Jury Members :	**Assoc. Prof. Dr. Ahmet DURAN** Istanbul Technical University	.....
	Assoc. Prof. Dr. F. Aylin SUNGUR Istanbul Technical University	.....
		.....

Date of Submission : **03 May 2013**
Date of Defense : **20 May 2013**

To my family,

FOREWORD

I would like to thank my advisor Prof. Dr. M. Serdar Çelebi, and Assoc. Prof. Dr. Ahmet DURAN for his guidance and advice during this thesis. I also would like to thank my family for supporting all of my endeavours throughout the years.

May 2013 Mehmet TUNÇEL

TABLE OF CONTENTS

Page

ABBREVIATIONS

UHeM	**:** Ulusal Yüksek Başarımlı Hesaplama Merkezi (Eng: National Center for High Performance Computing)
UFMM	**:** University of Florida Matrix Market
NNZ	**:** Number of nonzero of matrices
N	**:** Square matrix dimesions
CRS	**:** Compressed row storage
CCS	**:** Compressed column storage
PD	**:** Public Domain
LPGL	**:** GNU Library or "Lesser" General Public License

LIST OF TABLES

Page

LIST OF FIGURES

Page

DESIGNING A FAST DIRECT SPARSE MATRIX SOLVER FOR MULTI-CORE DISTRIBUTED SYSTEMS

SUMMARY

Many scientific and industrial problems are described by partial differential equations (PDEs). Handling of numerical solution of PDEs has been producing sparse linear equation systems $AX = B$. Generally, two methods are most common used to solve linear equation systems in computational science. One of them is direct methods and another is iterative methods. Along with easily practicability of iterative methods which are sequence of improving approximate solutions, direct methods attempt to solve the problems with exact solution in the the absence of the rounding error. So direct methods are seen more appealing through developing capacity of high performance computing (HPC) systems.

Direct solvers for sparse matrices have more different algorithmic mechanisms than for dense matrices because of the sparse matrix data structure and handling higher dimensional scientific problems. And parallel sparse direct solvers especially have another important issues like load balancing and scalability.

In this thesis, we consider parallel scalable direct solvers. We examine the effectiveness of the Distributed SuperLU for multi-core distributed memory parallel machines among several variants of sparse direct solvers.

Giving of background with general sparse direct solver algorithms, some important mechanisms have been mentioned separately in more detail.

Advantages and limitations of the sparse direct solvers for distributed memory systems have been discussed.

In our tests, scalability, tuning factors and constructions which needs further customization for various large sparse matrices have been separately examined.

Although it is not possible to use only one direct solver for all pattern of matrices, we propose a new algorithm SuperLU_MCDT (Multi-core Distributed SuperLU) which can exceed some limitations with new hardware and software developments.

Proposed SuperLU_MCDT is expected to take the fully advantage of multi-core distributed systems. Our studies show that the inter-node communication and intra-node memory requirements are critical and this existing overhead is partly removed with our new algorithm SuperLU_MCDT.

ÇOK ÇEKİRDEKLİ DAĞITIK SİSTEMLER İÇİN HIZLI DOĞRUDAN SEYREK MATRİS ÇÖZÜCÜ TASARLANMASI

ÖZET

Bilimsel ve endüstriye yönelik birçok problemin çözümünde doğrusal denklem sistemleri ortaya çıkmaktadır. Diferansiyel denklemlerin büyük bir yer edindiği bu problemlerde, birçok kısmi diferansiyel denklemlerin bağlaşık (ing: coupled) çözülme ihtiyacından dolayı analitik çözümlerden çok sayısal yöntemler tercih edilmektedir.

Sayısal yöntemlerle diferansiyel denklemlerin çözümü sonlu farklar ve sonlu elemanlar gibi birçok ayrıştırma yöntemi ile problemin sürekli uzaydan ayrık uzaya taşınmasını baz alır. Bu eşleme belli kafes (ing: mesh) noktalarında gerçekleştirilir ve sonucunda seyrek matrislerin katsayıları içerdiği doğrusal denklem sistemleri ortaya çıkmaktadır.

Sayısal yöntemleri iki ana başlık içinden ifade edebiliriz. Bunlar belli bir adım basamağında kesin sonuca ulaşan doğrudan (ing: direct) yöntemler ve yaklaştırım ile hatayı her adımda azaltmayı hedefleyen yinelemeli (ing: iterative) yöntemlerdir.

Yinelemeli yöntemlerin daha kolay programlanabilirliği hesaplamaların bilgisayar ortamında kullanımında ilk tercih olmasına neden olsa da, günümüz problemlerinin daha karmaşık bir yapıda olması yinelemeli yöntemlerin yaklaştırımını zorlaştırmaktadır. Bununla beraber, bir takım ön koşullandırıcı (ing: preconditioner) olarak adlandırdığımız yinelemeli yöntemlerde ele alınan problemden doğan katsıyalar matrisinin koşul sayısını (ing: condition number) düşürerek yakınsaklığını sağlayan ön uygulamalar ise her duruma cevap verememektedir. Bu nedenler doğrudan yöntemlerin programlanabilme kolaylığının yinelemeli yöntemler kadar olmamasına rağmen artık tercih edilebilir bir yöntem olarak görülmesine neden olmuştur. Günümüz yüksek başarımlı hesaplama teknolojilerindeki gelişmeler de doğrudan yöntemlerin daha geniş bir problem sahasına uygulanabilirliğini arttırmıştır.

Seyrek matrislerin doğrudan yöntemlerde ki geleneksel faktorizasyon algoritmaları ile ele alınması, bellekteki direk olmayan adreslemelerden dolayı ciddi performans kayıplarına neden olmaktadır. Bu nedenle supernode yaklaşımı gibi bazı yöntemler bu problemin giderilmesi için ele alınmaktadır. Böylece bilgisayar işlemcileri daha etkin bir şekilde kullanılmış olur. Bunun diğer bir performans metriğini etkileyen faktörü ise tıkız (ing: dense) BLAS kütüphanelerinin kullanımıdır ki matris matris ve matris vektör çarpımları için optimize edilmiş rutinler içerirler.

Seyrek matrislerin doğrudan yöntemler ile birlikte ele alınmasında dikkat edilecek noktalardan bir tanesi de faktorizasyon sırasındaki matristeki sıfır olan elemanların sıfır olmamasıdır. Çünkü seyrek matrisler tıkız olanlar gibi iki boyutlu dizilerde (n^2) değil , belleğin etkili kullanımı için daha az yer kaplayan üç ayrı dizide ($\approx 3n$) saklanmaktadır. Kontrolsüz artış gösteren sıfır olmayan matris elemanlarının çoğalması ise algoritmaları olumsuz etkileyebilmekte ve hatta bellek yersizliğinden dolayı başarısız sonuçlayabilmektedir.

Kısmi diferansiyel denklemlerin ayrıştırımında kafes noktalarının çözüm hassasiyetinin artırılması ihtiyacından dolayı sık olması veya hesaplama gerektirecek problem tanım alanının büyüklüğü nedeniyle çok büyük seyrek matrisler ortaya çıkmaktadır. Böyle denklem sistemlerinin tek bir hesaplama biriminde ele alınması ise donanımsal limitlerden dolayı imkansızdır. Çünkü çok büyük hesap yükü günlerce ve belki aylarca sonuçlanamayacak veyahut da bellek sınırlamasından dolayı hiç çalışamayacaktır. Bu nedenle böyle büyük problemlerin dağıtık sistemler ile ele alınması gerektir.

Bu tezde, yukarıda bahsettiğimiz hususlar sonucu paralel çalışan dağıtık bellek sistemlerini kullanan doğrudan çözücüler dikkate alınmıştır. Bu çözücülerden Distributed SuperLU merkezde olarak testler gerçekleştirilmiş ve çıkan sonuçlar aynı zamanda paralel bir doğrudan çözücü olan SuperLU_MCDT (Multi-core Distributed SuperLU)'nin tasarımın da bazı donanımsal ve yazılımsal limitlerin açılması noktalarında katkı sağlamıştır.

Tezin ilk kısmında örneklerle diferansiyel denklemlerin ayrıklaştırılması, bunun sonucunda çıkan seyrek matrislerin yinelemeli ve doğrudan yöntemler ile ele alınması karşılaştırılmış. Yapılan çalışmalar hakkında bilgi verilmiştir.

İkinci kısımda ise seyrek matris algoritmalarının çıkışı ve gelişimi; günümüzdeki doğrudan yöntemleri kullanan çözücüler, Distributed SuperLU ve SuperLU_MCDT'nin buradaki yeri ve özellikleri anlatılmıştır.

Doğrudan yöntemler için temel teşkil eden Gauss eliminasyon yönteminin ve basamak olduğu LU faktorizasyon yönteminin tıkız ve seyrek matrislerdeki matematiksel altyapısı ise üçüncü bölümde ele alınmıştır.

Distributed SuperLU ve doğrudan yöntemleri kullanan çözücüler için kritik mekanizmalar dördüncü bölümde tek tek ele anlatılmıştır. Bu mekanizmaların işleyişi ve önemli noktaları paralel dağıtık bellek sistemleri tasarımı için gerekli yönleri açısından ele alınmıştır.

Beşinci bölümde, testlerin hangi sistemlerde nasıl parametrelerle ele alındığına ve test sonuçlarının değerlendirilmesine yer verilmiştir.

Son olarak ise bu çalışmadan elde ettiğimiz sonuçlar ve genel değerlendirilmesi yer almaktadır.

Sonuç olarak şöyle diyebiliriz ki birçok bilimsel ve endüstriye ait problemlerin sonucunda seyrek doğrusal denklem sistemleri $AX = B$ ortaya çıkmaktadır. Bu sistemlerin hızlı, gürbüz ve ölçeklenebilir algoritmalar ile çözülmesi çok önemlidir. Aynı zamanda bu algoritmalarının günümüz yüksek performanslı sistemlerin getirdiği kapasite ölçeklerine göre uyarlanması birçok algoritmik yapının daha verimli uygulanmasına olanak sağlayacaktır.

Bütün matris desenleri için iyi performansı olan tek bir çözücünün olması mümkün gözükmemekle beraber, yeni yazılımsal ve donanımsal gelişmelere bağlı olarak bazı sınırlamaları aşan yeni bir algoritma (Superlu_MCDT) sunuyoruz. Bu algoritma ile çok çekirdekli işlemciye sahip dağıtık sistemlerin avantajlarından mümkün oldukça yüksek yararlanmaya çalıştık. Nodlar arası haberleşme yükü ve nod içi bellek gereksimi önemli bir yere sahiptir ve bu yükü yeni algoritmamız olan SuperLU_MCDT ile bir miktar kaldırmış olduk.

SuperLU_MCDT'nin geliştirilmesi yanında çalışmakta olduğumuz kısımlar: satır permütasyon matrisinin paralel bir algoritma ile elde edilmesi, otomatik olarak ayar parametrelerinin belirlenmesi, MPI + OpenMP hibrit programının geliştirilmesi ve çok çekirdekli işlemciler için geliştirilen paralel doğrusal cebir kütüphanesinin SuperLU_MCDT ye eklenmesidir. Bunun yanında GPU (Grafik İşleme Ünitesi, ing: Graphichs Processing Unit) ile heterojen dağıtık sistemlerde SuperLU_MCDT'nin uygulanması da yapmayı planladığımız çalışmalardandır.

1. INTRODUCTION

Many important problems in science and engineering are described by partial differential equations (PDEs). Some of these PDE problems can be handled analytically, but problems arising form complex coupled systems force us to use numerical methods since their more complicated analytic structure. Numerical solution of PDEs are based on the transferring continuous equations into the discrete space and there are a lot of possible methods like finite difference, finite element or volume for mapping. Also these methods generates linear systems which include large sparse matrices involving more zero entries than nonzero.

For example, we consider the problem of the steady-stead temperature distribution in a long uniform road and it is given by the second order and two point boundary value problem.

$$-u''(x) + \sigma u(x) = f(x), \qquad 0 < x < 1, \ \ \sigma \geq 0 \tag{1.1}$$

$$u(0) = u(1) = 0 \tag{1.2}$$

When finite difference methods are considered, the domain of the problem is partitioned into n subintervals with mesh points where width of the subintervals is equal.

$$x \in [0,1], \ \ x_j = jh, \ h = 1/n \tag{1.3}$$

The original differential equation 1.1 is replaced with a second order central finite difference approximation at each interior mesh point. In this replacement, we introduce an approximation $v_j \approx u(x_j)$ whose values satisfy $n-1$ linear equations.

$$\frac{-v_{j-1} + 2v_j - v_j + 1}{h^2} + \sigma v_j = f(x_j), \qquad 1 \leq j \leq n-1, \ \ v_0 = v_n = 0 \tag{1.4}$$

Thus, the problem may be represented in matrix form as linear equation system $Ax = b$ where A is coefficient matrix, b is right hand side vector and x is unknown vector.

$$\begin{pmatrix} 2+\sigma h^2 & -1 & & & & \\ -1 & 2+\sigma h^2 & -1 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & -1 \\ & & & & -1 & 2+\sigma h^2 \end{pmatrix} \begin{pmatrix} v_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} f(x_1) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ f(x_{n-1}) \end{pmatrix} \quad \textbf{(1.5)}$$

The linear system 1.5 is symmetric, tridiagonal, and another important observation is that coefficient matrix is also sparse. Sparse means that coefficient matrix includes zero entries much more than nonzero entries. In other word, a matrix is sparse if there is an advantage in exploiting its zero [3]. For instance, One of the advantages of storing only non-zeros is that this strategy makes possible to solve the linear system. Otherwise, memory will restrict us after a mesh size amount. if the mesh points are increased and the coefficient matrix is stored as dense having all entries. But increasing of mesh points is necessary for more accurate results and handling problems with big domain.

Generally, two methods are most common used to solve linear equation systems in computational science. One of them is direct and another is iterative methods. Direct methods attempt to solve the problems with exact solution in the the absence of the rounding error, after n step. But iterative methods struggle to obtain enough accuracy within many process steps which can not be estimated exactly. On the other hand, iterative methods have less time complexity. For example, the complexity is $O(n^3)$ for direct methods and it is $O(n^2)$ for Jacobi and Gauss-Seidel. Even, multigrid method has $O(n)$ complexity. So iterative methods have a big advantages on this point. But, more complicated iterative algorithms having less complexity like algebraic multigrid is difficult to implement on complex problems. And less complicated iterative algorithms having more complexity need also preconditioner which is more complicated. Moreover, another disadvantage of iterative methods is that it must start over again from the beginning in order to solve $Ax = b_2$, after solving $Ax = b_1$. In sum, direct methods is seen more appealing through developing capacity of high performance computing (HPC) systems.

Direct solvers for sparse matrices need more different algorithmic mechanisms than for dense matrices. For instance, One of them is fill-in which is the arising of new nonzero

values during the process of an algorithm in L and U factors. So extra memory usage can negatively effect if it is not controlled.

In the handling of the solution problem of the linear system $AX = B$, where A is a given large square sparse matrix, X is unknown vector or matrices and B is a given vectors or matrices. Gaussian elimination has an important part as a direct method in the numerical linear algebra for the solution of $AX = B$. The conventional LU decomposition algorithms for sparse matrices is not efficient because of the indirect memory addressing for sequential computers and also load balancing, scalability issue for parallel distributed memory systems.

In this thesis, we consider parallel scalable direct solvers. We examine the effectiveness of the SuperLU_DIST (Distributed SuperLU) for distributed memory parallel machines among several sparse direct solvers (see Li et al. [4], Amestoy et al. [5], Schenk and Gartner [6], Duran and Saunders [7], Duran et al. [8] and references contained therein). Several important points explained in the following chapters have taken part in the design of SuperLU_MCDT (Many-core Distributed SuperLU) (see [9] and [10]). These points taken out of tests can be mentioned as follows.

In our tests, a lot of results have been found about scalability [11] of Distributed SuperLU as far as 512 cores. Along with these successful results, Distributed SuperLU may show performance decreases for matrices having same sparsity level. On the other hand, achievement of the Distributed SuperLU about availability of the supernodes which are consecutive structures of entries make clear that there are some synchronization issues arising from the insufficient load balancing of the algorithm. Because availability test of supernode structures for randomly populated matrices shows that supernodal approach gives answer for wide-range domain of matrices.

Another result coming from the tests is about BLAS routines which supernodal approach make its usage possible [12]. As it can be seen in the numerical results, BLAS routines are optimized for CPU by vendors give performance increment which is multiple times.

Parallelization of the column ordering algorithms are based on graph partitioning. The test results of ParMETIS [13] assigned for column ordering and symbolic factorization in Distributed SuperLU and many solvers light the way that the usage of multi-core

technology with hybrid programing is a necessary since the overhead of the inter-node communication and inefficient usage of the intra-node.

Many matrices having apart difference patterns make tuning of the algorithm parameters important. Tests about three supernode parameters in Distributed SuperLU show that approximate % 14 performance gain is possible with tuning. So auto-tuning issue is the important mechanism which have been taken part in the design of SuperLU_MCDT, as well.

The remainder of this thesis is organized as follows. After the introduction and literature review chapters of the thesis, mathematical background of LU decomposition is presented in Chapter 3. Critical mechanisms of SuperLU for distributed systems are introduced in Chapter 4. In Chapter 5, numerical results are discussed. Chapter 6 concludes the thesis.

2. LITERATURE REVIEW

Sparse direct solvers has been changing since 1970 first appeared. In this chapter, development of the sparse direct solvers, their features and limitations, current available parallel direct solvers and Distributed SuperLU which is in the center of the our proposals will be mentioned.

2.1 Development of Sparse Direct Methods

In the 1950s, iterative methods generally were used for solving large systems. And there were only references about sparse matrices in the part of the books like combinatoric and graph theory. In the 1960s, linear programming problems and solution of the implicit ODES from engineering problems increased the usage of the sparse matrices. The first organization of Sparse Matrix Symposium was made at IBM Yorktown Heights in 1968 by the Mathematical Sciences Department [14].

A conferences on "Large Sparse Sets of Linear Equations" at St. Catherine's College, Oxford followed the symposium. About this time, the first theses [see Table2.1] about solution of sparse linear systems with direct methods were written [15].

Table 2.1: The first theses about solution of sparse linear systems with direct methods.

Author	Year	University	Thesis Title
Donald Rose	1970	Harvard	Symmetric elimination on sparse positive definite systems and the potential flow network problem
Alan George	1971	Stanford	Computer implementation of the finite-element method
Iain Duff	1972	Oxford	Analysis of sparse systems
Andrew Sherman	1975	Yale	On the efficient solution of sparse systems of linear and non-linear equations

In the 1970s, the solver packages started to appear. Some of them are MA18, M28 from Harwell Subroutine Library (HSL); SPARSPAK by George and Liu at University

of Waterloo, and YSMP by Andrew Sherman. Following of these years, the topics of sparse matrices and implementation of direct method on computational mathematics increased rapidly and showed results as books [see Table 2.2], conferences and meetings.

Table 2.2: Books about sparse linear systems and direct methods.

Year	Author	Book
1973	Tewarson	Sparse Matrices
1976	Brameller, Allan and Hamam	Sparsity
1981	George and Liu	Computer Solution of Large Sparse Positive Definite Systems
1983	Osterby and Zlatev	Direct Methods for Sparse Matrices
1984	Pissanetsky	Sparse Matrix Technology
1986	Duff, Erisman and Reid	Direct Methods for Sparse Matrices
1991	Zlatev	Computational Methods for General Sparse Matrices
2006	Davis	Direct Methods for Sparse Linear Systems

Many future research challenges have followed the this rapidly development. Some important issues of sparse direct methods appeared in its developing process like elimination tree and pivot strategies, error handling, supernode, and new approach on LU decomposition and triangular solution [16].

One of the important development for sparse direct methods is to find new approaches about fill-reducing orderings because a good fill reducing algorithm is essential for reducing time and memory needings. But they are an NP-hard problem [17]. So many heuristics are used and one important algorithm is *nested dissection*. As it is seen in the Figure 2.1, algorithm can gather the fill-in values to near of the non-zeros.

This algorithm applied on symmetric matrices has two successful implementors: ParMETIS and PT-Scotch [18] [19].

Another important issue for sparse direct solvers is avoiding from indirect addressing and use dense BLAS routines. Sparse matrices are generally stored in three arrays, and finding value for a entry needs to seek for arrays. So vectorized operations which is very important for efficiency is not used directly. Two important methods were

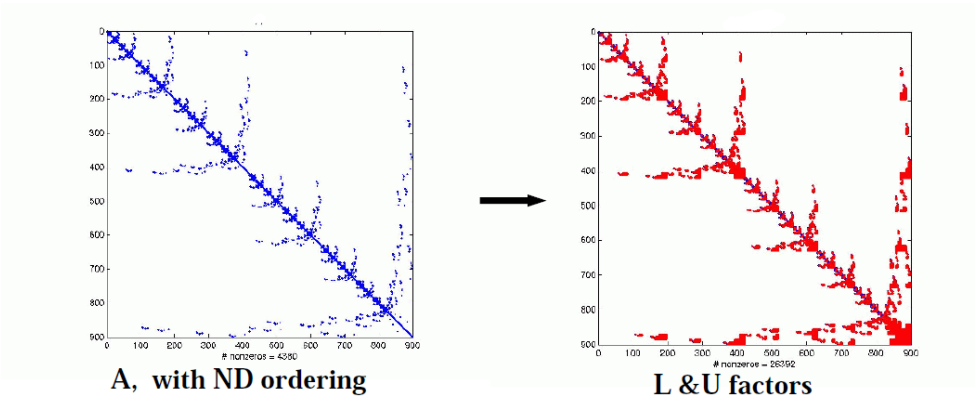

Figure 2.1: Nested dissection ordering [1].

presented to overcome this deficiency. They are *multi-frontal* [5] and *supernodal* [2] methods. The main idea of this approaches to put in order entries, such that matrix-matrix, matrix-vector multiplications are performed like in dense matrices without affecting negatively factorization.

A big deal for parallelization of the *LU* factorization is partial pivoting. In execution time, searching of the suitable pivot element, and transferring of the data is affecting negatively the organization of the data structures and memory usage. As a solution Li and Demmel has presented *static pivoting* algorithm not using of the partial pivoting and which is stable as partial pivoting algorithms [20].

In our days, scientific and industrial problems have been being more complicated. Consequently, being handled of the problem is challenge no longer. In this thesis, we considered direct solvers oriented to Distributed SuperLU and analyzed the limitations and researched the solution with algorithmic and hardware aspects.

2.1.1 Current list of direct solvers for distributed memory systems

Many solvers were arised in parallel with sparse direct methods progresses as we mentioned above. During these developments which have been keeping on, many types of HPC (High-Performance Computing) environments have become available. Some of them are massively parallel computers and PC clusters with distributed memory. As a result, new direct methods on distributed memory and based on MPI programming have come out. When it comes to HPC development, linear algebra libraries start to implement new suitable strategies for algorithms.

Here, we list the softwares for high performance computers for solving sparse and dense linear system problems with direct methods and give some informations about their description, version, license and language written. In tables below, all direct solvers are aimed for distributed systems via MPI. The Table 2.4 shows a list of dense direct solvers [21]. The Table 2.5 shows a list of sparse direct solvers [21], [9].

2.2 Introduction on SuperLU

SuperLU is a general purpose direct solver performing LU decomposition and, its first version was developed in 1997. Supernodal approach which is one of the important mechanism of SuperLU gives advantages of performing of dense vector matrix operation [2]. Using unsymmetrical matrix implementation of supernode, it also generalized this technic.

Table 2.3: Status of SuperLU software.

	Sequential SuperLU	SuperLU_MT	SuperLU_DIST
Platform	Serial	Shared memory	Distributed memory
Language	C	C + Pthreads (or OpenMP)	C + MPI
Data Type	Real/Complex Single/Double	Real/Complex Single/Double	Real/Complex Double

SuperLU covers a set of libraries including three subroutines for solving sparse linear systems. All three libraries oriented on LU decomposition of the equations $AX = B$ where A is square nonsingular matrices and X, B are dense vectors. Matrix A may be non-symmetric and it is not need to be positive definite. SuperLU were especially designed and developed for unsymmetrical matrices.

SuperLU algorithm were implemented on three libraries for different platforms. They are as follows: Sequential SuperLU, Multithreaded SuperLU (SuperLU_MT) for shared memory systems and Distributed SuperLU (SuperLU_DIST) for distributed memory systems. All three libraries use memory hierarchy organization as advantage and have some different strategy and mechanism [22]. Here, we are related to distributed version.

Table 2.4: The list of dense direct solvers.

Name	Description	License	Version	Language
Elemental	A framework for distributed-memory dense linear algebra.	BSD	0.77	C++
PLAPACK	Parallel linear algebra package, an infrastructure for coding linear algebra algorithms at a high level of abstraction.	?	3.2	F77/C
PRISM	Parallel research on invariant subspace methods to develop infrastructure and algorithms for the parallel solution of eigenvalue problems.	?	1.0	F77
ScaLAPACK	A library of high-performance linear algebra routines for parallel distributed memory machines. ScaLAPACK solves dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems.	BSD	2.0.2	F77/C
Trilinos/Pliris	A dense solver package.	BSD	11.0	C

Table 2.5: The list of sparse direct solvers.

Name	Description	License	Version	Language
DSCPACK	Domain separator codes for solving sparse linear systems.	?	1.0	C
HSL	Collection of packages for large scale scientific computation.	Own	HSL 2013 released.	F77/C
PaStiX	Parallel sparse matrix package.	CeCILL-C	5.2.1	F77/C
MUMPS	Multifrontal massively parallel sparse direct solver.	PD	4.10.0	F77/C
PSPASES	For symmetric positive definite matrices.	Own	1.0.3	F77/C
SPOOLES	Sparse object oriented linear equations solver.	PD	2.2	C
SuperLU_DIST	For large non-symmetric systems of linear equations. It performs an LU factorization with partial pivoting.	BSD-like	3.2	F77/C
SuperLU_MCDT (Many Core Distributed)	A hybrid algorithm utilizing the MPI+OpenMP hybrid programming approach to obtain a scalable and improved SuperLU.	BSD-like	Beta	F77/C
Trilinos/Amesos	Interface to third-party direct solvers.	LGPL	11.0	C
Trilinos/Amesos2	Interface to third-party direct solvers.	BSD	11.0	C++

2.2.1 Distributed SuperLU

Distributed SuperLU was designed for distributed memory systems. This library uses MPI [23] for parallel programming model and can handle double precision real and complex matrices. For equation $AX = B$ LU decomposition can performing both on global A and B matrices input or with distributed row-wise partitioning. Distributed sparse matrices are stored in CRS (compress row storage) format. If there is enough memory, global input operations are faster than which in distributed input interface.

Data structures of L and U matrices in distributed SuperLU are located as blocks on rectangular process grid. After supernode detection distribution of matrices is implemented in two dimensional block-cyclic fashion.

We can say that important property apart from sequential and multi-threaded SuperLU is not using of partial pivoting during Gaussian elimination. Static pivoting are used instead and, stability is provided with permuting large elements to diagonal and iterative refinement and, the results of GESP (Gaussian elimination with static pivoting) which are as stable as partial pivoting implementation are even obtained for large range matrices [4]. In this way, there has been obtained load balancing and parallelization of algorithm [24].

Distributed SuperLU's GESP (Gaussian elimination with static pivoting) algorithm can be respectively sketched as below:

(1) Row-column equilibration and row permutation: $A \leftarrow P_r \cdot D_r \cdot A \cdot D_c$

(2) A column permutation to preserve sparsity: $A \leftarrow P_c \cdot A \cdot {P_c}^T$

(3) Symbolic analysis to determine the nonzero structures of L and U

(4) Factorization of $A = LU$.

(5) Triangular solutions using L and U.

(6) Iterative refinement.

In each step except for row permutation, algorithm is performed in parallel. In Chapter 4, we give comprehensive details.

3. MATHEMATICAL BACKGROUND OF LU FACTORIZATION

In this chapter, algebraic properties of Gaussian elimination as direct method, LU factorization and sparse matrix approach will be reviewed on

$$Ax = b$$

where A is a nonsingular matrix, x is an unknown vector, b is a given vector and matrix representation coming from linear equation system

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ...a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + ...a_{1n}x_n = b_2$$

.....

.....

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + ...a_{nn}x_n = b_n$$

3.1 Gaussian Elimination

Gaussian elimination is an transformation of linear system to triangular form [25]. It will be illustrated with the system

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.1}$$

Multiplying the first equation by a_{21}/a_{11} and subtracting from the second equation (assuming that $a_{11} \neq 0$), new equivalent system is obtained.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & {a_{22}}^{(2)} & {a_{23}}^{(2)} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ {b_2}^{(2)} \\ b_3 \end{pmatrix} \tag{3.2}$$

where

$${a_{22}}^{(2)} = a_{22} - (a_{21}/a_{11})a_{12} \tag{3.3}$$

$$a_{23}{}^{(2)} = a_{23} - (a_{21}/a_{11})a_{13} \tag{3.4}$$

$$b_2{}^{(2)} = b_2 - (a_{21}/a_{11})b_1. \tag{3.5}$$

Correspondingly, Multiplying the first equation by a_{31}/a_{11} and subtracting from the third equation, new equivalent system is obtained.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}{}^{(2)} & a_{23}{}^{(2)} \\ 0 & a_{32}{}^{(2)} & a_{33}{}^{(2)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2{}^{(2)} \\ b_3{}^{(2)} \end{pmatrix} \tag{3.6}$$

where

$$a_{32}{}^{(2)} = a_{32} - (a_{31}/a_{11})a_{12} \tag{3.7}$$

$$a_{33}{}^{(2)} = a_{33} - (a_{31}/a_{11})a_{13} \tag{3.8}$$

$$b_3{}^{(2)} = b_3 - (a_{31}/a_{11})b_1. \tag{3.9}$$

Similarly multiplying the new second row by $a_{32}{}^{(2)}/a_{22}{}^{(2)}$ and subtracting from the new third equation (assuming that $a_{22}{}^{(2)} \neq 0$), new system is produced.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}{}^{(2)} & a_{23}{}^{(2)} \\ 0 & 0 & a_{33}{}^{(3)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2{}^{(2)} \\ b_3{}^{(3)} \end{pmatrix} \tag{3.10}$$

where the new terms are given by

$$a_{33}{}^{(3)} = a_{33}{}^{(2)} - (a_{32}{}^{(2)}/a_{22}{}^{(2)})a_{23}{}^{(2)} \tag{3.11}$$

and

$$b_3{}^{(3)} = b_3{}^{(2)} - (a_{32}{}^{(2)}/a_{22}{}^{(2)})b_2{}^{(2)}. \tag{3.12}$$

Now, the linear system 3.1 has been transformed the upper triangular form 3.10 and the components of the solution can easily be gotten by the following steps

$$x_3 = b_3{}^{(3)}/a_{33}{}^{(3)} \tag{3.13}$$

$$x_2 = (b_2 - a_{23}{}^{(2)}x_3)/a_{22}{}^{(2)} \tag{3.14}$$

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \tag{3.15}$$

In general, a upper triangular system $Ux = c$ which is like 3.10 can be solved by the steps

$$x_n = c_n/b_nn \tag{3.16}$$

$$x_k = (c_k - \sum_{j=k+1}^{n} u_{kj}x_j)/u_{kk}, \qquad k = n-1, n-2, ..., 1 \tag{3.17}$$

on the condition that $u_{kk} \neq 0,\ k = 1,2,...,n$. This process is called back substitution.

Similarly the lower triangular system $Lc = b$ can be solved by the steps

$$c_1 = b_1/l_{11} \tag{3.18}$$

$$c_k = (b_k - \sum_{j=1}^{k-1} l_{kj}c_j)/l_{kk}, \qquad k = 2,3,..,n \tag{3.19}$$

on the condition that $l_{kk} \neq 0,\ k = 1,2,...,n$. This process is also called forward substitution.

Gauss elimination is a process generating zeros in the first column, second column and so on. It can be generalized on $Ax = b$ with formula

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)})a_{ij}^{(k)},\ \ i,j > k \tag{3.20}$$

$$b_i^{(k+1)} = b_i^{(k)} - (a_{ik}^{(k)}/a_{kk}^{(k)})b_k^{(k)},\ \ i > k \tag{3.21}$$

where $a_{ij}^1 = a_{ij},\ \ i,j = 1,2,...,n$. The very important requirement is that

$$a_{kk}^k \neq 0,\ \ k = 1,2,...,n.$$

These entries are known as pivot in Gaussian elimination.

In the situation of that $a_{kk}^k = 0$, the rows are exchanged. For example:

$$\begin{pmatrix} 0 & 2 \\ 5 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ 10 \end{pmatrix} \Longrightarrow \begin{pmatrix} 5 & 3 \\ 0 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 10 \\ 6 \end{pmatrix} \tag{3.22}$$

This operation selecting pivot element as largest absolute value from the column is called partial pivoting and the equation always can be reordered through interchanging rows if A is nonsingular [25].

3.1.1 LU factorization

Gaussian elimination produce upper triangular matrix U, and also lower triangular part L of the linear system $A = LU$ can be obtained. Handling the processes with another point of view is just enough. Let $A \in \mathbb{C}^{nxn}$ be a square matrix. Gaussian elimination is done by subtracting multiples from subsequent rows and maintaining this process

for all rows. This process is equivalent to multiplication of A by a sequence of lower triangular matrices L_k.

$$L_{n-1}...L_2L_1A = U \tag{3.23}$$

If we multiply 3.23 with inverses of $L_{n-1}...L_2L_1$ on the left

$$A = L_1^{-1}L_2^{-1}...L_{n-1}^{-1}U, \qquad L_1^{-1}L_2^{-1}...L_{n-1} = L \tag{3.24}$$

we obtain LU factorization of A. In practical Gaussian elimination, the matrices L and U are stored on the original matrix A and the entries of L are computed with formula

$$l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)} \qquad i > k. \tag{3.25}$$

As an example of LU factorization [25], let A be a square matrix $4x4$

$$A = \begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix} \tag{3.26}$$

Firstly, we subtract first row from second, third and fourth rows relatively two,four and three times.

$$L_1A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{pmatrix} \tag{3.27}$$

Similarly, we subtract second row from third and fourth rows relatively three and four times.

$$L_2L_1A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & -4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \end{pmatrix} \tag{3.28}$$

Thirdly, we subtract third row from fourth row.

$$L_3L_2L_1A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix} = U \tag{3.29}$$

Now, to complete of the factorization, we need to compute $L = L_1^{-1}L_2^{-1}L_3^{-1}$. Finally, the multiplication of $L_1^{-1}L_2^{-1}L_3^{-1}$ is the unit lower triangular matrix with a minus times the non-zeros subdiagonal entries L_1, L_2 and L_3.

$$A = LU \rightarrow \begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 3 & 1 & 0 \\ 3 & 4 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.30}$$

3.1.2 LU factorization for sparse matrices

Sparse matrices have mostly nonzero and generally LU factorization based on Gaussian elimination for sparse matrices has same process phases. But it needs extra processes because of its data structures. For example, when we consider the matrix S

$$S = \begin{pmatrix} x & x & x & x & x & x & x & x \\ x & x & & & & & & \\ x & & x & & & & & \\ x & & & x & & & & \\ x & & & & x & & & \\ x & & & & & x & & \\ x & & & & & & x & \\ x & & & & & & & x \end{pmatrix} \tag{3.31}$$

there will be appeared many nonzero after LU factorization. If they are not controlled, we can be faced with a big limitation for algorithmic aspect. But if we reorder matrix S with row P_r and column P_c permutation matrices as below

$$P_r S P_c = \begin{pmatrix} x & & & & & & & x \\ & x & & & & & & x \\ & & x & & & & & x \\ & & & x & & & & x \\ & & & & x & & & x \\ & & & & & x & & x \\ & & & & & & x & x \\ x & x & x & x & x & x & x & x \end{pmatrix} \tag{3.32}$$

stable a matrix appear to handle. After that, LU factorization may start on permuted matrix $P_r S P_c$. After factorization, solution can be reached by following steps.

$$x = S^{-1} b$$

$$P_r S P_c = LU \Longrightarrow$$

$$A = {P_r}^{-1} L U {P_c}^{-1}$$

$$x = \left({P_r}^{-1} L U {P_c}^{-1}\right)^{-1} b$$

$$x = P_c U^{-1} L^{-1} P_r b$$

Gaussian elimination algorithm for sparse matrices has several algorithmic mechanism. In the next chapter, ordering and other related issues will be considered.

4. CRITICAL MECHANISMS OF SUPERLU FOR DISTRIBUTED SYSTEMS

In this chapter, we will handle Distributed version 3.*X*. Giving of background with general and improved algorithms, we will separately mention some important mechanisms in more details.

4.1 Background of Distributed SuperLU

4.1.1 LU factorization for sparse systems

Direct methods for solving linear systems of the form $Ax = b$ are based on computing $A = LU$, where L and U are lower and upper triangular, respectively. Computing the triangular factors of the coefficient matrix is also known as $LUdecomposition$. Following the factorization, the original system is trivially solved by solving the triangular systems $Ly = b$ and $Ux = y$. L is a unit lower triangular matrix $(Lii = 1)$ and U is an upper triangular matrix. The factorization can also be applied to non-square matrices.

A common formulations of LU decomposition for dense matrices are shown as below [26].

for J=J+1 to N
  $A[J,I] = A[J,I]/A[I,I]$*; /*division step, computes column i of L */*
end for
for K=I+1 to N
  for J=I+1 to N
    $A[J,J] = A[J,J] - A[J,I] \cdot A[I,K]$*; /* update step */*
  end for
end for
end for

Here, simple column-based algorithm for LU decomposition of an *NxN* dense matrix.

The algorithm overwrites by L and U. The diagonal entries after factorization belong to U; the unit diagonal of L is not stored. If we take the coefficient matrix as sparse, we should differently handle problem from storage of data, data structures, load balancing to graph partioning and many mechanisms. Let us consider of these phases.

4.2 Four Phases to Solve AX=B

Direct solvers for sparse matrices involve much more complicated algorithms than for dense matrices. The main complication is due to the need for efficient handling *fill-in* in the factors L and U. A typical sparse solver consists of four distinct steps as opposed to two in the dense case:

1. An ordering step that reorders the rows and columns such that the factors suffer little fill, or that the matrix has special structure such as block triangular form.

2. An analysis step or symbolic factorization that determines the nonzero structures of the factors and create suitable data structures for the factors.

3. Numerical factorization that computes the L and U factors.

4. A solve step that performs forward and back substitution using the factors.

4.2.1 The preprocessing of matrix

First processing step for LU decomposition is tuning of the coefficient matrix A. This preprocessing including three parts, equilibration, row and column ordering, are practiced for numerical stability and fill reducing.

4.2.1.1 Numerical stability

Distributed SuperLU performs static pivoting instead of dynamic pivoting which are used for maintaining numerical stability with interchanging of rows. Hence, there has been avoided from inefficient access pattern of data and, gained ground for synchronization of the algorithm. And static pivoting approach has been as stable as partial pivoting for extensive test matrices with equilibration and row ordering [24].

Equilibration

Equilibration is implemented for rows and columns. Here D_r and D_c, diagonal row and column equilibration matrices respectively, are computed so that $A \leftarrow D_r \cdot A \cdot D_c$ has

better condition than A. Each entries of rows and columns are divided by maximum absolute value of related column or row. After scaling, A has unit norm.

Overwriting to original matrix, equilibration can be serially computed by MC64 developed by Duff and Koster [27] or in parallel. This step can be changed or stopped by input options if wanted.

Row ordering

The second important step for numerical stability is to compute a row permutation matrix P_r. Distributed SuperLU use the serial code of MC64 [27] for both equilibration and row ordering. Firstly coefficient matrix A has maximum absolute entry 1 with equilibration, then row ordering moves the absolute values 1 on diagonal. So pivoting numbers are maximized: $A \leftarrow P_r \cdot D_r \cdot A \cdot D_c$

P_r is computed by maximizing the product of the diagonal entires. Bipartite graph taken out from matrix A is used. Each vertices are weighted by entry values and maximum matching algorithm on that graph finds the related values, using search tree algorithms [28].

4.2.1.2 Fill reducing

When A is sparse, the triangular factors L and U typically have nonzero entries in many more locations than A does. This phenomenon is known as *fill-in*, and results in a superlinear growth in the memory. Column ordering algorithms are used for fill reducing in this important part of the direct solution of a sparse linear system. As a result of this permutation, load balance, communication reducing is also provided [29].

Column ordering

A sparse matrix can be represented with the adjacency matrix of a graph. Column ordering algorithms typically use this graph. Because of the NP-hard problem, heuristics are used. The column ordering are implemented on the symmetric structures which has been obtained by $A^T + A$ or $A^T \cdot A$. Computing of $A^T \cdot A$ may be expensive both for time and space, so $A^T + A$ is generally used.

AMD (Approximate minimum degree), COLAMD (Column approximate minimum degree), MMD (Multiple minimum degree) and nested dissection used in METIS and

ParMETIS are some of the column permutation algorithms. Distributed SuperLU give several option for column permutation:

- No ordering.
- Minimum degree ordering (on $A^T + A$).
- Minimum degree ordering (on $A^T \cdot A$).
- METIS ordering (on $A^T + A$).
- ParMETIS ordering (on $A^T + A$).
- User input permutation matrix.

ParMETIS is the parallel choice [13] and PT-Scotch can also used instead [19]. Unlike ParMETIS, PT-Scotch does not support the partitioning multiconstraint. So ParMETIS seems better choice. In this step, column permutation matrix P_c is implemented symmetrically like that $P_c \cdot A \cdot {P_c}^T$ not to lost the entries of the main diagonal which are same in the matrix $P_r \cdot D_r \cdot A \cdot D_c$.

4.2.2 Symbolic factorization

The golden rule of sparse matrix theory is to predict the structure of the numerical results and allocate memory before the numerical computation. So performing symbolic analysis is very important since it is used to determine the nonzero structures of L and U when there is no need to pivot. Thus there has been avoided from indirect addressing for very large data. It also allows us to organize numerical factorization before we it is done [20].

The building of elimination tree for Cholesky factorization of symmetric positive definite matrices is the base of symbolic factorization. This process is executed on reduction graph of Cholesky L factor.

In the case of unsymmetric matrices, symbolic factorization replaced directed acyclic graph of L and U apart from of symmetric matrices. Since tree structure can not be used, the parallelization of symbolic factorization is more difficult than which are for symmetric matrices.

Distributed SuperLU can handle symbolic factorization as sequentially or in parallel. Parallel symbolic factorization use ParMETIS and works only on power of two processors. If the number of processors is not equal to 2^q, possible smaller number processor is chosen and the data are redistributed.

4.2.3 Numerical factorization

In Gaussian elimination, there are several approaches for numerical factorization. Often used algorithms are left-looking (fan-in) and right-looking (fan-out). Both of them have same floating point operation but their memory access patterns are different as it can be seen in Figure 4.1, 4.2.

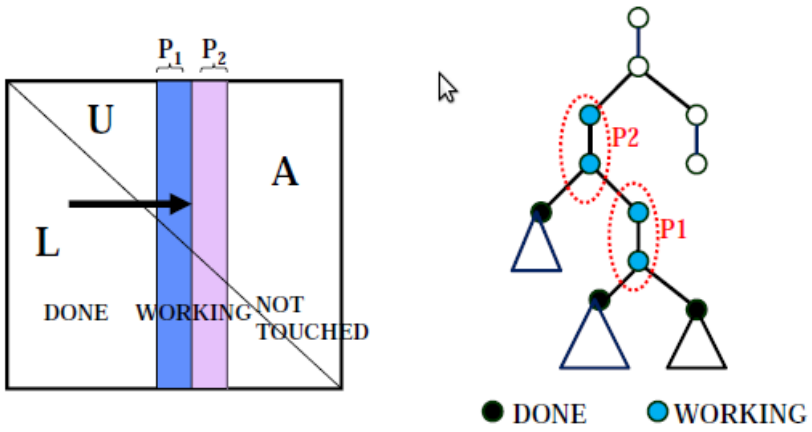

Figure 4.1: LU factorization as left looking algorithm [1].

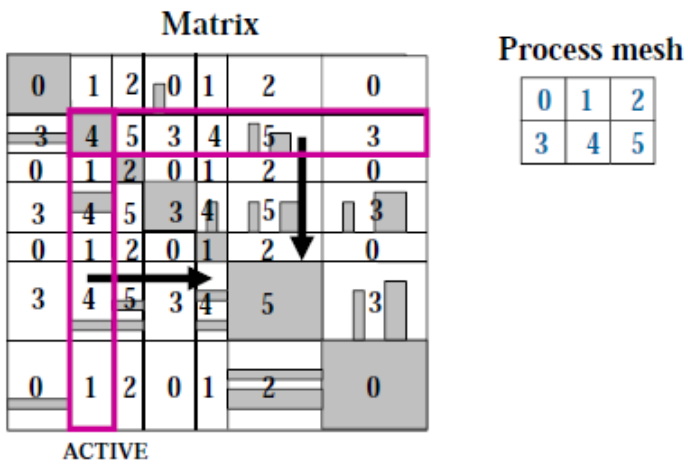

Figure 4.2: LU factorization as right looking algorithm [1].

The pseudo-code of block left and right looking Gaussian elimination algorithm are respectively given below.

for block K = 1 to N

$\quad$ *Compute* $U(1:K-1,K)$

$\quad$ *Update* $A(K:N,K) \leftarrow A(K:N,K) - L(1:N,1:K-1) \cdot U(1:K-1,K)$

$\quad$ *Factorize* $A(K:N,K) \rightarrow L(K:N,K)$

end for

for block K = 1 to N

$\quad$ *Factorize* $A(K:N,K) \rightarrow L(K:N,K)$

$\quad$ *Compute* $U(K,K+1:N)$

$\quad$ *Update* $A(K+1:N,K+1:N) \leftarrow$

$\qquad\qquad A(K+1:N,K+1:N) - L(K+1:N,K) \cdot U(K,K+1:N)$

end for

Distributed SuperLU choose right-looking algorithm for following reasons.

- The sparsity pattern can be determined before numerical factorization.
- Parallelization are easier since having independent update submatrices.
- there is only need a small amount of buffer space for transferring a block column of L and a block row of U.

Distributed SuperLU use pipelined right-looking algorithm with *mpi_isend* and *mpi_irecv*. Thus, loss of time were prevented arising from blocking operation [4].

4.2.4 Triangular solution

Distributed SuperLU use the data structures to perform the sparse triangular solution using L and U. In parallel, routines solves the sparse linear system by forward and back substitutions. Here right-hand side matrix B can handle as distributed such in coefficient matrix A.

After factorization of $P_c \cdot P_r \cdot D_r \cdot A \cdot D_c \cdot {P_c}^T$, we can solve $AX = B$ by evaluating

$$X = A^{-1} \cdot B$$

$$P_c \cdot P_r \cdot D_r \cdot A \cdot D_c \cdot {P_c}^T = L \cdot U \Longrightarrow$$

$$A = {D_r}^{-1} \cdot {P_r}^{-1} \cdot {P_c}^{-1} \cdot L \cdot U \cdot {{P_c}^T}^{-1} \cdot {D_c}^{-1}$$

As a result:

$$X = ({D_r}^{-1} \cdot {P_r}^{-1} \cdot {P_c}^{-1} \cdot L \cdot U \cdot {{P_c}^T}^{-1} \cdot {D_c}^{-1})^{-1} \cdot B$$

$$X = D_c \cdot ({P_c}^T \cdot (U^{-1} \cdot (L^{-1} \cdot (P_c \cdot P_r \cdot (D_r \cdot B)))))$$

Multiplication from the right to left solves the system. Here, Diagonal matrices D scales and permutation matrices P permutes the rows. Multiplication by L^{-1} and U^{-1} is to solve triangular system with L and U respectively.

4.3 Iterative Refinement

Iterative refinement is a phase used sometimes after the numerical factorization to improve the accuracy of computed solution [30]. Given a computed solution x, iterative refinement algorithm works for $A \cdot x = b$ like

Compute residual $r = A \cdot x - b$
While residual too large
  Solve $A \cdot d = r$ *for correction* d
  Update solution $x = x - d$
  Update residual $r = A \cdot x - b$
End while

The computed triangular factors are again used for iterative refinement. The criterion of not being “residual too large“ in the iterative refinement algorithm above is that *BERR* (componentwise relative backward error) [31] should not exceed the machine roundoff level. And *BERR* is calculated

$$BERR \equiv max_i |r_i| / s_i$$

where the scale factor s_i is

$$s_i = (|A| \cdot |x| + |b|)_i = \sum_j |A_{ij}| \cdot |x_j| + |b_i|$$

4.4 Supernode

Supernodes, considered to enhance the performance of solver, are the consecutive columns with identical structures. These consecutive structures are stored as dense blocks and used in block partitioning. The size of each supernode is matrix dependent. For unsymmetric matrices, there are several supernode definition. Some possible of them are in Figure 4.3. Here, stripes show patterns having same structure and black box is dense storage of structure [2].

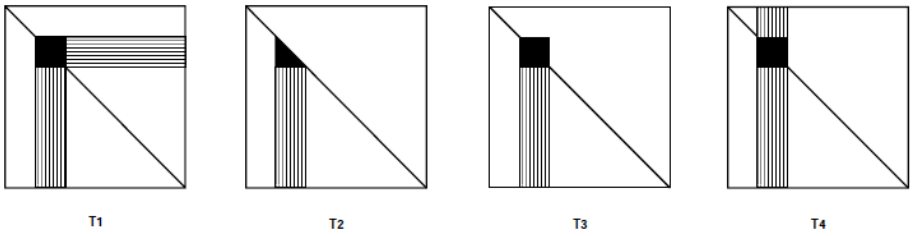

Figure 4.3: Supernodes for unsymmetric matrices [2].

When we consider the supernodes, T_1 seems more suitable for Gaussian elimination. But T_2 and T_3 have cache advantage in update process. Distributed SuperLU use T_2 because of larger structure and upper triangular part of supernode can be empty. As a result od supernode, nonzero entries in matrix A is addressed by two dimensional array. So algorithm has been able to use BLAS routines [3]. The advantage of supernodal approach for BLAS routines will mention in numerical results chapter.

5. NUMERICAL RESULTS

In this chapter, we discuss numerical results, advantages and limitations of the SuperLU solvers. Although the existing versions of SuperLU are scalable and tuned for many matrices, they are sensitive to tuning and need further customization for various large sparse matrices. Therefore, we designed and generated a collection of large patterned and random sparse matrices which are larger than most of those real matrices from the University of Florida sparse matrix collection [14]. For example, we did sensitivity analysis to several parameters including total number of non-zeros and degree of sparsity for randomly populated sparse matrices.

We modify the SuperLU solvers in order to improve their scalability via several ways. We propose a new hybrid algorithm utilizing the MPI+OpenMP hybrid programming approach.

5.1 Experimental Testbeds

Research test has done on two HPC system at National Center for High Performance Computing (UHeM) [32] and Rechenzentrum Garching (RZG) of the Max Planck Society whose sources were provided by DECI9 call that PRACE (Partnership for Advanced Computing in Europe), the European research infrastructure for High Performance Computing (HPC), makes it possible for researchers from across Europe and the world. Brief information about systems is in Table 5.1

5.2 Experimental Setups

For all experiments in this thesis, we used the Distributed SuperLU SuperLU with MC64 for static pivoting and equilibration, ParMETIS for column ordering, and parallel symbolic factorization for estimation fill-in. All programs were compiled by Intel MPI and TAU 4.2.222 was used for analyses.

Table 5.1: Description of used hardware metrics of CPU.

System name	Karadeniz	Hydra
Processor	Intel Xeon 5500 [Quad Core] (Nehalem)	Intel Sandy Bridge-EP [8 core]
Frequency	2.67 GHz	2.6 GHz
L1 cache	4x32 KB	8x32 KB
L2 cache	4x256 KB	8x256 KB
L3 cache	8 MB	20 MB
Number of compute nodes	64	610
Number of compute cores	512	9760
Memory architecture	Distributed	Distributed
Per core memory amount	3 GB	4 GB
Disk Space per node	292 GB	40 TB
High performance network	InfiniBand 20 Gbps	InfiniBand FDR14
Operating system	Linux	Linux

5.3 Test Matrices

Many multi-scale modeling applications in science and engineering would like to capture more details of the system without ignoring any important conservation laws as much as possible, resulting in more general matrices. Therefore we consider a portfolio of test matrices containing randomly populated sparse matrices in addition to patterned matrices. We generate 30 different randomly populated matrices RAND_30K_3, ..., RAND_30K_100 for each. Each experiment is done at least four times. We describe the matrices in Table 5.2 and Table 5.3, respectively.

5.3.1 Description of matrices

Table 5.2: Description of randomly populated matrices.

Name	Order	NNZ	NNZ N	Condition Number	Origin
RAND_30K_9	30000	270000	9	2.51 x 106	UHeM
RAND_30K_11	30000	330000	11	8.82 x 105	UHeM
RAND_30K_30	30000	900000	30	1.13 x 106	UHeM
RAND_30K_50	30000	1500000	50	7.03 x 105	UHeM
RAND_30K_75	30000	2250000	75	1.16 x 106	UHeM
RAND_30K_100	30000	3000000	100	3.39 x 106	UHeM
RAND_10K_3	10000	30000	3	7.10 x 105	UHeM
RAND_20K_3	20000	60000	3	3.19 x 105	UHeM
RAND_30K_3	30000	90000	3	1.20 x 106	UHeM
RAND_40K_3	40000	120000	3	3.90 x 106	UHeM
RAND_50K_3	50000	150000	3	1.20 x 106	UHeM
RAND_60K_3	60000	180000	3	2.14 x 106	UHeM

Table 5.3: Description of patterned matrices.

Name	Order	NNZ	NNZ/N	Nonzero Pattern Symmetry	Numeric Value Symmetry	Condition Number	Origin	Kind of Problem
SB1_45	541779	4581692	8.45	45.19	29.31		UHeM	
7DIAG_1M_545	1000000	5450000	5.45	0.00%	0.00%	3.47 x 105	UHeM	
BBMAT	38744	1771722	45.73	53.00%	0.00%	2.09 x 109	UFMM	Computational fluid dynamics (CFD)
ECL32	51993	380415	7.32	92.00%	60.00%	9.41 x 1015	UFMM	Semiconductor device
EMILIA_923	923136	40373538	43.74	100.00%	100.00%		UFMM	Geomechanical structural
G7JAC200SC	59310	717620	12.1	3.00%	0.00%	1.43 x 1014	UFMM	Economic
HELM2D03LOWER_20K	392257	1939353	4.94	0.00%	0.00%		UHeM	
INVEXTR1_NEW	30412	1793881	58.99	97.00%	72.00%	2.77 x 1018	UFMM	CFD
LHR71C	70304	1528092	21.74	0.00%	0.00%	1.56 x 1017	UFMM	Light hydrocarbon recovery
MARK3JAC140SC	64089	376395	5.87	7.00%	1.00%	5.83 x 1013	UFMM	Economic
MIXTANK_NEW	29957	1990919	66.46	100.00%	99.00%	4.40 x 1011	UFMM	CFD
PRE2	659033	5834044	8.85	33.00%	7.00%	3.11 x 1023	UFMM	Frequency-domain circuit simulation
STOMACH	213360	3021648	14.16	85.00%	0.00%	8.01 x 101	UFMM	3D electro-physical mode

5.4 Scalability

The code of Distributed SuperLU has been tested in order to measure the performance scalability of various randomly populated sparse matrices and patterned sparse matrices up to 512 cores (depending on number of non-zeros and sparsity level) on the Linux Nehalem Cluster [32] available at the National Center for High Performance Computing (UHeM).

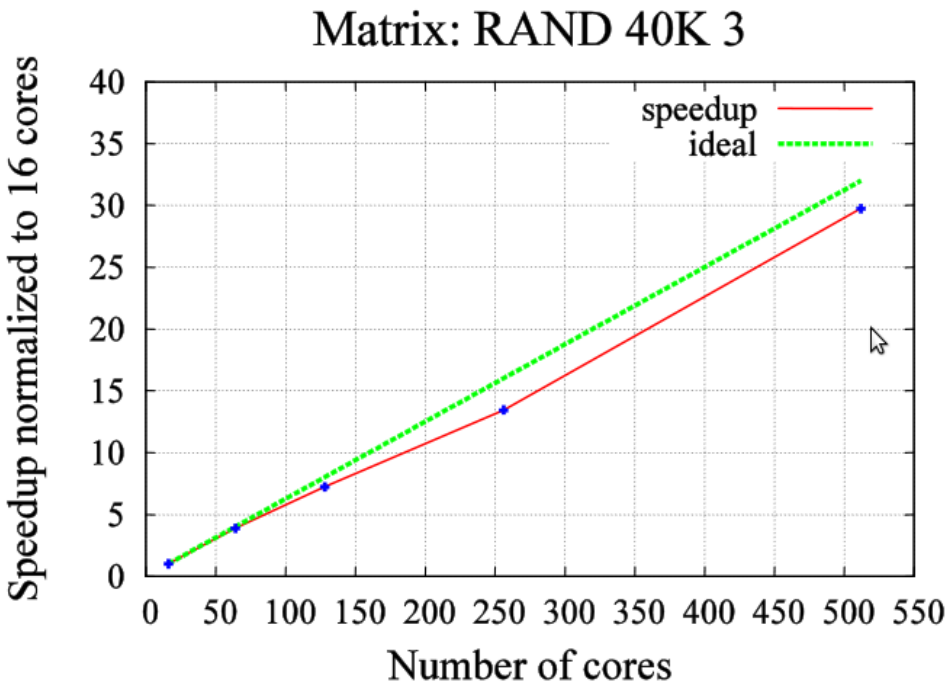

Figure 5.1: Speed up for matrix RAND_40K_3.

Table 5.4: Wall clock time and normalized speed-up for RAND_40K_3.

Number of Cores	Meshes	Wall Clock Time (s)	Speed-up
16	(4x4)	849.69	1.00
64	(8x8)	218.49	3.89
128	(8x16)	117.55	7.23
256	(16x16)	63.21	13.44
512	(16x32)	28.58	29.73

The rich pattern spectrum of matrices and the NP-complete problem of best reordering for minimum fill-in are important challenges. For example, the code has shown scalable speed-up up to 512 cores for RAND_40K_3 in our tests as illustrated in Figure 5.1 and Table 5.4. While the speed-up for the symmetric matrix EMILIA_923 is close

to ideal up to 256 cores, we observe divergence at 512 cores in Figure 5.2 and Table 5.5.

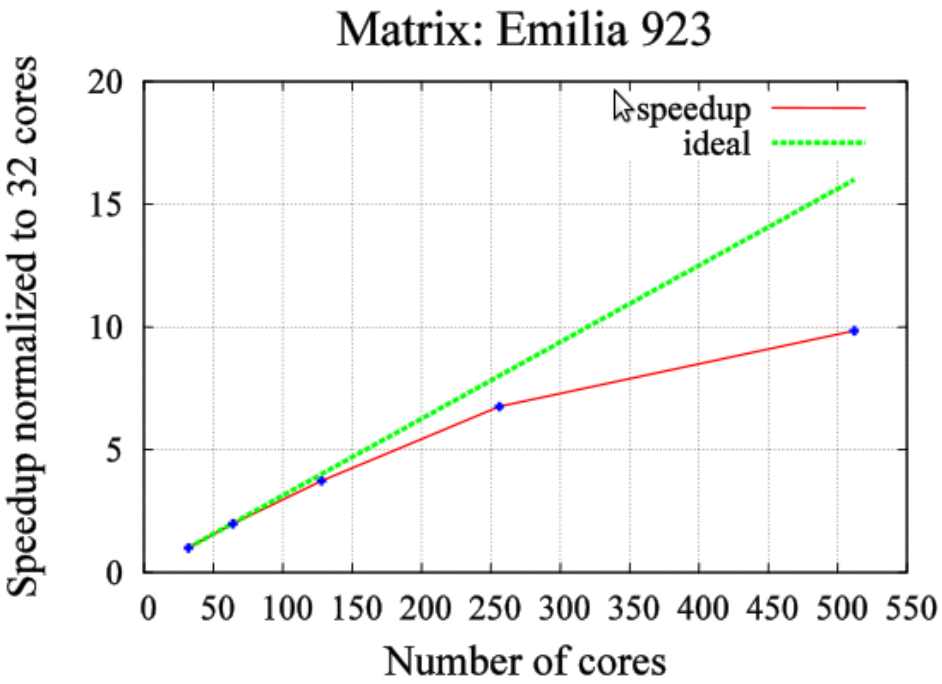

Figure 5.2: Speed up for matrix EMILIA_923.

Table 5.5: Wall clock time and normalized speed-up for EMILIA_923.

Number of Cores	Meshes	Wall Clock Time (s)	Speed-up
16	(4x4)	1472.02	1.00
64	(8x8)	743.29	1.98
128	(8x16)	394.78	3.73
256	(16x16)	217.85	6.76
512	(16x32)	149.63	9.84

For randomly populated large sparse matrices, we find a peak of numerical factorization, symbolic factorization, and consequently wall clock time for a value of seven non-zeros per row in Figure 5.3 and Table 5.6. This may be related to availability of supernodes. After 7, they decrease gradually as sparsity decreases to 75 with a slow rise at 100 non-zeros per row.

In Table 5.7, the numerical factorization time dominates in the distribution of total wall clock time as expected for the randomly populated sparse matrices with 3 non-zeros per row. We observe that the wall clock time and consequently total time increases as matrix order and number of non-zeros increase, given fixed sparsity.

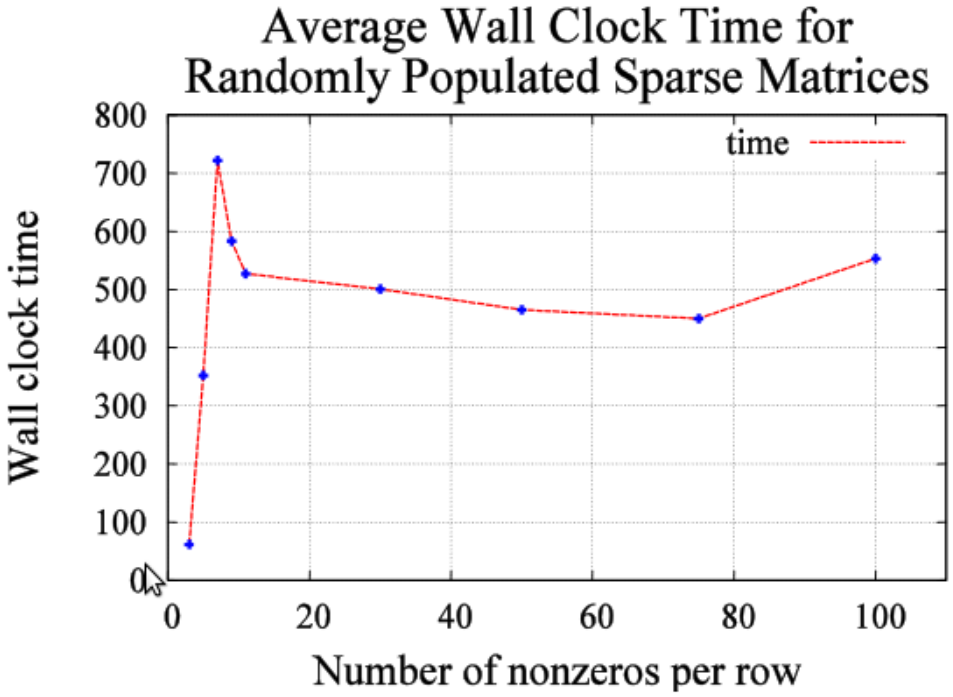

Figure 5.3: Average wall clock time as a function of various sparsity levels for randomly populated sparse matrices.

Table 5.6: Wall clock time for randomly populated sparse matrices RAND_30K_3, ..., RAND_30K_100 as the sparsity level decreases using 64 core (8x8).

NNZ per row	3	5	7	9	11
Wall clock time	61.87	352.10	721.95	583.15	527.20
NNZ per row	30	50	75	100	
Wall clock time	500.66	465.00	450.08	553.23	

Table 5.7: Distribution of wall clock time for randomly populated sparse matrices RAND_10K_3, ..., RAND_60K_3 as the number of non-zeros increases using 64 core (8x8).

Order	10000	20000	30000	40000	50000	60000
NNZ	30000	60000	90000	120000	150000	180000
Equil time	0.00	0.01	0.01	0.01	0.02	0.02
RowPerm time	0.01	0.02	0.04	0.06	0.12	0.11
ColPerm time	0.82	1.20	1.48	2.05	1.65	2.04
SymFact time	0.06	0.38	1.08	2.11	3.54	5.42
Distribute time	0.06	0.07	0.20	0.20	0.30	0.45
Factor time	0.98	14.65	74.95	212.43	334.01	857.66
Solve time	0.02	0.05	0.11	0.18	0.22	0.33
Refinement time	0.08	0.15	0.26	0.47	0.48	0.70
Total	2.03	16.53	78.13	217.51	340.34	866.73

We find that the memory overhead coming from ParMETIS [13] becomes one of the dominating factors in the distribution of wall clock time on n-diagonal sparse matrices for certain large numbers of cores. For example, we generated 7DIAG_1M_545 as

a seven diagonal unsymmetric matrix with distances +50000, +100000, +400000, - 200000, -300000 and -500000 from main diagonal having random 5450000 real numbers between 0.5 and 1. The column permutation time takes 41% of the wall clock time for 7DIAG_1M_545 when 64 cores are used. We find similar results for this kind of n-diagonal unsymmetric/symmetric sparse matrices while using a number of cores such as 64. This affects the scalability of SuperLU_DIST negatively. In Table 5.8, the total time increased from 9.96 s. (16 cores) to 17.38 s. (64 cores).

Table 5.8: Distribution of wall clock time for randomly populated sparse matrices RAND_10K_3, ..., RAND_60K_3 as the number of non-zeros increases using 64 core (8x8).

	ParMETIS			MeTiS		
Number of cores	4	16	64	4	16	64
Mesh	(2x2)	(4x4)	(8x8)	(2x2)	(4x4)	(8x8)
Equil time	0.09	0.17	0.21	0.09	0.17	0.21
RowPerm time	0.83	0.85	0.88	0.80	0.85	0.88
ColPerm time	3.41	2.30	7.11	10.06	10.29	10.55
SymFact time	0.34	0.17	0.20	0.24	0.25	0.25
Distribute time	1.17	0.64	0.54	0.59	0.41	0.13
Factor time	2.00	2.62	6.07	0.53	0.43	0.55
Solve time	0.92	0.75	0.56	0.25	0.15	0.08
Refinement time	3.09	2.46	1.81	1.04	0.66	0.37
Total	11.85	9.96	17.38	13.60	13.21	13.02

5.5 Column Ordering

One of the important phase for sparse LU factorization is column ordering. This operation that is not necessary for the decomposition on dense matrices is needed to reduce fill-in and preserve sparsity when we carry out the decomposition. We compared three important options from several column permutations for Distributed SuperLU.

- Minimum degree ordering on structure $A^T + A$.
- METIS (nested dissection ordering on structure $A^T + A$).
- ParMETIS (nested dissection ordering on structure $A^T + A$).

Ordering on structure $A^T * A$ was not chosen because of the cost of the matrix-matrix multiplication.

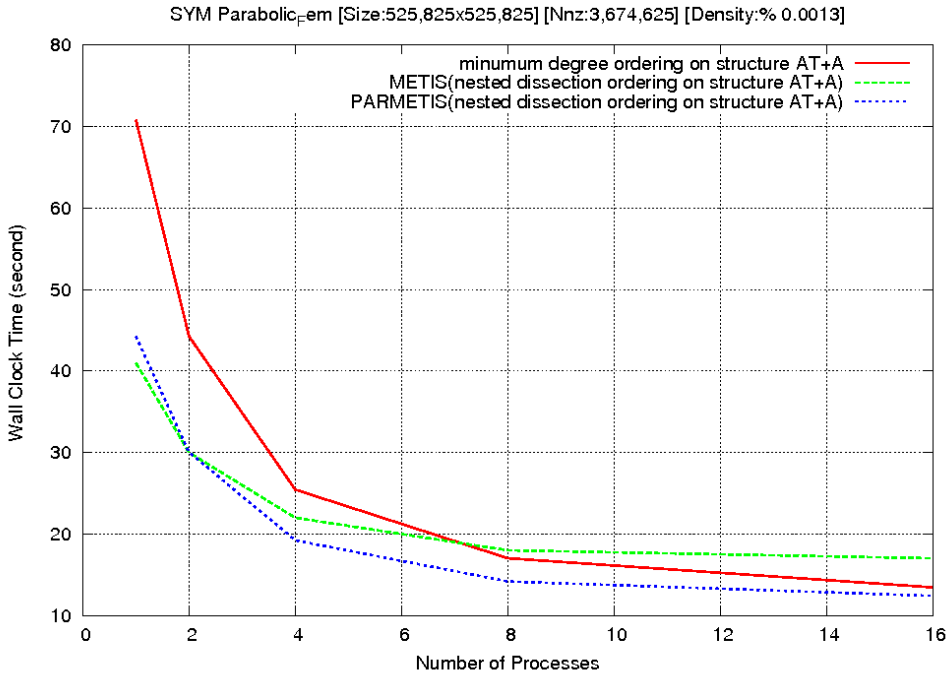

Figure 5.4: Major column ordering algorithm comparison for distributed systems.

As result seen in Figure 5.4, ParMETIS is the appealing selection even for sequential cases. Certainly, ParMETIS which is parallel version of METIS is better for distributed systems and manipulation of large matrices. Hence we will be solved the memory requirement.However ParMETIS fails for some matrices with dimension more than five millions. Indeed it is not expected that heuristic column ordering algorithms as a NP-hard problem [18] is works for all cases, but it is the open work to optimize and tune the algorithms for more cases. On the other hand, PARMETIS is the essential choice as a parallel version of column ordering necessary for sparse direct because of the intra-node memory limitations.

5.6 Linking with Different BLAS Libraries

Computation based on block submatrix updating is important part of the numerical factorization for SuperLU solvers, as well. Factorization algorithms in sequential SuperLU and Distributed SuperLU are based on supernodes [12]. and most time-consuming function in factorization is the following block update:

$$A(I,J) \leftarrow A(I,J) - L(I,K)xU(K,J).$$

Since L has a regular dense structure and block U(K, J) contains dense vectors, Level 3 BLAS is used effectively. So optimizing of the calling of the dense matrix-matrix

multiplication routine (Level 3 BLAS) on used system brings advantages about wall clock time and accuracy.

Table 5.9: TAU time analysis of factorization routine (pdgstrf) of Distributed SuperLU for matrix767440.

Number of Processes	1	4	16	64
Mesh	1x1	2x2	4x4	8x8
MKL	9139.529	928.314	286.586	167.216
C BLAS3	15131.786	2829.78	687.423	284.662

In our test, we generally observe that Distributed SuperLU solves the sparse linear about three times faster as seen in Table 5.9 when using GEMM routine of Intel MKL tuned for the Nehalem cluster instead of standard C BLAS3 routines; and tests with Intel MKL BLAS often appear more accurate because of its specific CPU vendor optimization. So there are a lot of BLAS libraries like ATLAS [33], GenBLAS [34] and GOTO BLAS [35] but, BLAS routines which are written for specific their own CPU by vendor should be chosen.

5.7 Tuning Factor

Supernodal mechanism of Distributed SuperLU has important role in algorithm. So tuning of the supernode parameters effect significantly the performance of the solver. There are three important machine-dependent parameters.

- relax: the relaxation parameter; if the number of nodes (columns) in a subtree of the elimination tree is less than relax, this subtree is considered as one supernode, regardless of the their row structures.
- maxsuper: the maximum size for a supernode.
- fill: the estimated fills factor for the adjacency structures of L and U, compared with A.

Firstly, we tested the *maxsuper* parameter without changing other two parameters. After average of the test result that are in Tables 5.10, 5.11, 5.12; it has been clear that maximum supernode size should be 110 as a different from default value 60.

Secondly, we continued to test $fill$ values with constant parameters $relax = 110$ and without changing the default value of $rest = 12$. It can be seen in Table 5.13 that Distributed SuperLU lost a little performance by reason of $fill$ parameter which cause the memory expansion when it is given not enough.

After the test of the *maxsuper* and $fill$ parameters, we examined the *relax* parameter. *Relax* parameter is very important for performance because it can cause the cache missing. We take constant for the *maxsuper* and $fill$ parameters with their optimal performance values 110 and 100 respectively. As it can be seen in Tables 5.14 and 5.15, optimal value is obtained as 80.

The solving wall clock time of matrix SB1_45 is 176.42 seconds with default parameters ($maxsuper = 60, relax = 12, fill = 5$) and it also is 162.22 seconds with optimal tuning parameters ($maxsuper = 110, relax = 80, fill = 100$). we get approximately % 10 extra performance when we compare the test result of matrix SB1_45. Performance income may arise to %17 with different test matrices like diagonal dominant matrices.

As a result, it can be concluded that machine-dependent tuning parameters are important factor for Distributed solvers and auto tuning mechanism is an open problem to get more performance and to avoid failed results.

5.8 Parallel Matrix Input

Distributed SuperLU has a subroutine which reads compress column storage (CCS) format matrix file. But it is not an efficient method while we are handling huge matrices since memory limitation. Reading a huge matrix from a single data file is limited by memory in nodes and also effects performance negatively. So we added a function in SuperLU_MCDT for parallel matrix input. Hence we have possibility to process big data.

For parallel input, we use separate matrix file parts which are written as compress row storage (CRS) file format and have local indexes. As it seen in Figure 5.5, root processes read the related parts of matrix file, divide and send matrix portion to leaves

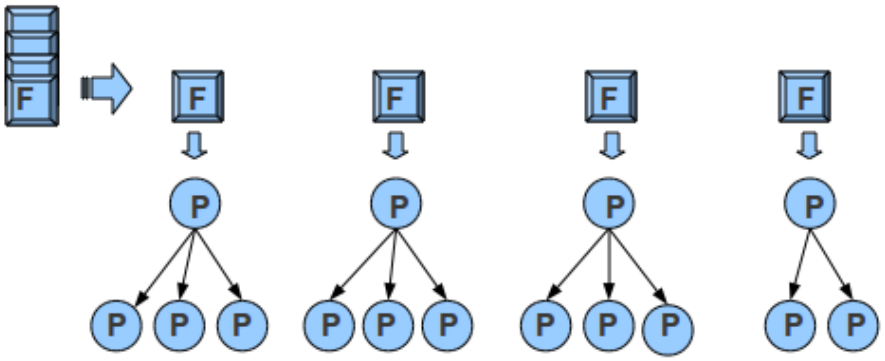

Figure 5.5: Parallel input (CRS format) for SuperLU_MCDT (Multi-core Distributed SuperLU).

of process tree. It is not necessary that number of processes is product of number of matrix file parts.

5.9 Memory Limitations

Although the existing versions of SuperLU work well for many matrices, they need to be improved for certain types of sparse matrices, even for simple pattern matrices produced by basic differential equations.

Memory requirement of direct method solvers grows in a superlinear with respect to the size of the sparse linear system because of the fill-in phenomenon. Although Distributed SuperLU uses optimized routines to take advantage of computer architecture, in particular memory hierarchy (caches) and parallelism while performing Gaussian elimination (LU factorization), there has been the situation that it uses the swap memory even for very simple matrix patterns.Here we test the sparse symmetric tridiagonal matrices with different diagonal distances on Nehalem Cluster by four processors having about three or six GB memory for each. And we got average of results after eight times repeating on four cores. In Table 5.16, we see that Distributed SuperLU can get the result three or five times slower while memory limit decreases by half. Extra memory usage coming from parallelism force to use swap memory. If we can avoid from intra-node communication and use the inter-node communications via infiniband (IB) network, we obtain several advantages of parallelism without some limitations. So Hybrid programming with MPI+OpenMP becomes indispensable for bigger matrices and thousands cores.

Table 5.10: Analysis of tuning factor of *maxsuper* for matrix SB1_45 with 64 processes.

maxsuper	20	30	40	60	80
relax	12	12	12	12	12
fill	5	5	5	5	5
Non-zeros in L	926074064	926074064	926074064	926074064	926074064
Non-zeros in U	931232142	931232448	931232640	931233002	931233166
non-zeros in L+U	1856764427	1856764733	1856764925	1856765287	1856765451
non-zeros in LSUB	319993047	304393411	297365937	291075556	288557481
no of supers	179636	176748	175375	174052	173414
EQUIL time (s)	0.14	0.14	0.14	0.14	0.14
ROWPERM time (s)	14.65	14.83	14.83	14.62	14.8
COLPERM time (s)	9.78	8.81	8.76	8.71	8.8
SYMBFACT time (s)	19.48	19.1	17.89	17.23	17.48
DISTRIBUTE time (s)	3.2	2.42	2.64	2.87	2.48
FACTOR time (s)	199.39	153.07	138.57	130.09	125.29
SOLVE time (s)	0.46	0.44	0.41	0.4	0.4
REFINEMENT time (s)	2.69	2.57	1.62	2.35	2.38
TOTAL time (s)	249.79	201.38	184.86	176.41	171.77

Table 5.11: Analysis of tuning factor of *maxsuper* for matrix SB1_45 with 64 processes.

maxsuper	90	98	110	120	160
relax	12	12	12	12	12
fill	5	5	5	5	5
Non-zeros in L	926074064	926074064	926074064	926074064	926074064
Non-zeros in U	931233507	931233588	931233724	931233757	931234271
non-zeros in L+U	1856765792	1856765873	1856766009	1856766042	1856766556
non-zeros in LSUB	287248024	287738708	286576319	287564290	287940056
no of supers	173201	173065	172897	172789	172494
EQUIL time (s)	0.14	0.14	0.14	0.14	0.14
ROWPERM time (s)	14.84	14.79	14.64	14.82	14.69
COLPERM time (s)	8.76	8.75	8.69	8.82	8.8
SYMBFACT time (s)	17.49	17.37	17.29	17.56	17.25
DISTRIBUTE time (s)	2.44	2.5	2.45	2.24	2.55
FACTOR time (s)	125.74	126.16	123.33	124.73	126.43
SOLVE time (s)	0.4	0.4	0.4	0.39	0.4
REFINEMENT time (s)	2.33	1.55	2.35	1.53	2.35
TOTAL time (s)	172.14	171.66	169.29	170.23	172.61

Table 5.12: Analysis of tuning factor of *maxsuper* for matrix SB1_45 with 64 processes.

maxsuper	180	320	640	1280
relax	12	12	12	12
fill	5	5	5	5
Non-zeros in L	926074064	926074064	926074064	926074064
Non-zeros in U	931234731	931236847	931240009	931247245
non-zeros in L+U	1856767016	1856769132	1856772294	1856779530
nonzero in LSUB	287041482	295298215	317058437	349240023
no of supers	172383	172051	171843	171751
EQUIL time (s)	0.14	0.14	0.14	0.14
ROWPERM time (s)	14.84	14.59	14.57	14.79
COLPERM time (s)	8.75	8.75	9.47	13.64
SYMBFACT time (s)	16.98	17	16.94	16.76
DISTRIBUTE time (s)	2.67	2.9	3	77.36
FACTOR time (s)	126.01	142.74	211.93	395.74
SOLVE time (s)	0.4	0.41	0.41	0.45
REFINEMENT time (s)	2.36	2.44	1.64	11.68
TOTAL time (s)	172.15	188.97	258.1	530.56

Table 5.13: Analysis of tuning factor of *fill* for matrix SB1_45 with 64 processes.

fill	5	50	60	100
memory expansion	14	3	2	0
relax	12	12	12	12
maxsuper	110	110	110	110
non-zeros in L	926074064	926074064	926074064	926074064
non-zeros in U	931233724	931233724	931233724	931233724
nonzero in L+U	1856766009	1856766009	1856766009	1856766009
nonzero in LSUB	286576319	286576319	286576319	286576319
no of supers	172897	172897	172897	172897
EQUIL time (s)	0.14	0.14	0.14	0.14
ROWPERM time (s)	14.64	14.7	14.64	14.83
COLPERM time (s)	8.69	8.69	8.8	8.78
SYMBFACT time (s)	17.29	15.84	15.7	14.7
DISTRIBUTE time (s)	2.45	2.47	4.46	2.23
FACTOR time (s)	123.33	123.78	135.22	122.93
SOLVE time (s)	0.4	0.41	0.4	0.4
REFINEMENT time (s)	2.35	1.6	1.55	2.32
TOTAL time (s)	169.29	167.63	180.91	166.33

Table 5.14: Analysis of tuning factor of *relax* for matrix SB1_45 with 64 processes.

relax	10	20	30	40	50
maxsuper	110	110	110	110	110
fill	100	100	100	100	100
non-zeros in L	924008520	932222881	938127643	943374188	948438960
non-zeros in U	929206967	937244994	943086884	948334642	953331254
nonzero in L+U	1852673708	1868926096	1880672748	1891167051	1901228435
nonzero in LSUB	286684876	286711313	286737680	286535460	286090012
no of supers	180497	154568	143292	136733	131796
EQUIL time (s)	0.14	0.14	0.14	0.15	0.14
ROWPERM time (s)	14.69	14.61	14.66	14.69	14.61
COLPERM time (s)	8.78	8.78	8.75	8.69	8.68
SYMBFACT time (s)	14.55	14.47	14.41	14.4	14.35
DISTRIBUTE time (s)	2.25	2.25	2.3	2.29	2.18
FACTOR time (s)	124.07	122.85	121.69	122.69	122.03
SOLVE time (s)	0.42	0.37	0.36	0.37	0.36
REFINEMENT time (s)	1.61	2.19	2.14	1.46	1.43
TOTAL time (s)	166.51	165.66	164.45	164.74	163.78

Table 5.15: Analysis of tuning factor of *relax* for matrix SB1_45 with 64 processes.

relax	60	70	80	90	100
maxsuper	110	110	110	110	110
fill	100	100	100	100	100
non-zeros in L	954323903	960244021	965388903	970020367	973859224
non-zeros in U	959169323	965072546	970194853	974845458	978702328
nonzero in L+U	1912951447	1924774788	1935041977	1944324046	1952019773
nonzero in LSUB	285469546	284665973	283868946	283070473	282356591
no of supers	126859	122061	118163	114981	112335
EQUIL time (s)	0.14	0.14	0.14	0.14	0.14
ROWPERM time (s)	14.65	14.61	14.7	14.62	14.61
COLPERM time (s)	8.74	8.73	8.71	8.71	8.71
SYMBFACT time (s)	14.31	14.27	14.21	14.17	14.16
DISTRIBUTE time (s)	2.21	2.18	2.17	2.17	2.17
FACTOR time (s)	121.16	121.37	120.63	121.71	121.71
SOLVE time (s)	0.36	0.34	0.34	0.33	0.33
REFINEMENT time (s)	2.08	2.01	1.32	1.9	1.92
TOTAL time (s)	163.65	163.65	162.22	163.75	163.75

Table 5.16: Wall clock time for sparse three diagonal matrices with different memory sizes per process on four cores.

Matrix	3D200K	3D600K	3D800K	3D990K	3D1200K	3D400K	3D1800K	3D1900K
Diagonal Distance	200K	600K	800K	990K	1200K	1400K	1800K	1900K
NNZ	5600K	4800K	4400K	4020K	3600K	3200K	2400K	2200K
NNZ/N	2.8	2.4	2.2	2.01	1.8	1.6	1.2	1.1
Wall Clock Time (s) via 3 GB memory per process	48.381	80.513	34.421	172.575	14.308	15.193	28.253	8.913
Wall Clock Time (s) via 6 GB memory per process	22.188	49.182	20.82	101.687	8.41	6.975	5.673	5.281

6. CONCLUSIONS

We believe that it is not possible to use only one direct solver for all pattern of matrices because of wide range of matrix sets, the NP-hard problems of graph partitioning used in ordering and unavoidable fill-in factor.

SuperLU_DIST has shown scalable speed-up between 256 and 512 cores for many test matrices. Also the tests of randomly populated large sparse matrices seemed that Distributed SuperLU is successful about finding supernodes. Moreover, we find that the memory overhead is coming from usage of ParMETIS in symbolic factorization for some matrices. We also showed that tuning of the algorithm related to dependency of distributed system gains approximately % 14 performance advantage for overall.

After obtaining a robust version of scalable SuperLU, we proposed a new hybrid algorithm for multi-core distributed server systems. Our studies reveal that inter-node communication and intra-node memory requirements are critical and this existing overhead is partly removed with our new algorithm SuperLU_MCDT (see [9] and [10]).

Proposed multi-core algorithm SuperLU_MCDT is specially works fine for sparse matrices resulting from coupled partial differential equations. Effectiveness of the algorithm is presented for both random sparse matrices and Emilia_923 sparse matrix (taken from Florida Matrix Collection [36]). This study is an initial works for SuperLU algorithm to effectively run on multi-core distributed system and further improvements on both algorithmic and programming perspectives are required.

Beside the point of development progress for SuperLU_MCDT, we work on parallelization of row ordering, auto tuning, hybrid programming with MPI + OpenMP and integration of PLASMA (The Parallel Linear Algebra for Scalable Multi-core Architectures). Also we will implement SuperLU_MCDT for heterogeneous systems with GPUs.

7. ACKNOWLEDGMENTS

This work was partially supported by the PRACE (Partnership for Advanced Computing in Europe) project funded in part by the EUs 7th Framework Programme (FP7/2011-2013) under grant agreement no. 283493

Computing resources used in the work of this thesis were provided by the National Center for High Performance Computing of Turkey (UHeM) (http://www.uybhm.itu.edu.tr) under grant number 1001682012 for Ahmet Duran, M Serdar Çelebi, Mehmet Tunçel.

Many thanks to ITU, UHeM and PRACE for their important support.

REFERENCES

[1] **Li, X.S.**, SuperLU: Sparse Direct Solver and Preconditioner, 13th DOE ACTS Collection Workshop, August 14-17, 2012.

[2] **Demmel, J.W.**, **Eisenstat, S.C.**, **Gilbert, J.R.**, **Li, X.S. and Liu, J.W.** (1999). A supernodal approach to sparse partial pivoting, *SIAM Journal on Matrix Analysis and Applications*, ***20***(3), 720–755.

[3] **Duff, I.S.**, **Erisman, A.M. and Reid, J.K.** (1986). *Direct methods for sparse matrices*, Clarendon Press Oxford.

[4] **Li, X.S. and Demmel, J.W.** (2003). SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, *ACM Transactions on Mathematical Software (TOMS)*, ***29***(2), 110–140.

[5] **Amestoy, P.R.**, **Duff, I.S. and L'Excellent, J.Y.** (2000). Multifrontal parallel distributed symmetric and unsymmetric solvers, *Computer methods in applied mechanics and engineering*, ***184***(2), 501–520.

[6] **Schenk, O. and Gärtner, K.** (2004). Solving unsymmetric sparse systems of linear equations with PARDISO, *Future Generation Computer Systems*, ***20***(3), 475–487.

[7] **Duran, A.**, **and Saunders, B.** Gen_SuperLU package (version 1.0, August 2002), a part of LinBox package, containing a set of subroutines to solve a sparse linear system A*X=B over any field., *Internet Address: http://web.itu.edu.tr/aduran/Gen_SuperLU.pdf*.

[8] **Duran, A.**, **Saunders, B.D. and Wan, Z.** (2003). Hybrid algorithms for rank of sparse matrices, *Proceedings of the SIAM International Conference on Applied Linear Algebra (SIAM-LA), Williamsburg, VA*, pp.July 15–19, 2003.

[9] **Duran, A.**, **Celebi, M.S.**, **Tuncel, M. and Akaydın, B.** (2012). Design and implementation of new hybrid algorithm and solver on CPU for large sparse linear systems PN:283493, *PRACE-2IP white paper, Libraries, WP 43*, (3), 720–755.

[10] **Celebi, M.S.**, **Duran, A.**, **Tuncel, M. and Akaydın, B.** (2012). Scalable and improved SuperLU on GPU for heterogeneous systems PN:283493, *PRACE-2IP white paper, Libraries, WP 43*, (3), 720–755.

[11] **Duran, A.**, **Celebi, M.S. and Tuncel, M.** (2012). Scalability of SuperLU solvers for large scale complex reservoir simulations, *SPE and SIAM Conference on Mathematical Methods in Fluid Dynamics and Simulation of Giant Oil and Gas Reservoirs, Istanbul, Turkey*, pp.September 3–5, 2012.

[12] **Demmel, J.W.**, **Eisenstat, S.C.**, **Gilbert, J.R.**, **Li, X.S. and Liu, J.W.** (1999). A supernodal approach to sparse partial pivoting, *SIAM Journal on Matrix Analysis and Applications*, ***20***(3), 720–755.

[13] **Karypis, G.**, **Schloegel, K. and Kumar, V.** (2003). Parmetis, *Parallel graph partitioning and sparse matrix ordering library. Version 3.1*, ***2***.

[14] **Duff, I.S.** Development and History of Sparse Direct Methods, *The SIAM Conference on Applied Linear Algebra*, p.11/34.

[15] **Duff, I.S.** Development and History of Sparse Direct Methods, *The SIAM Conference on Applied Linear Algebra*, p.21/34.

[16] **Davis, T.A.** (2006). *Direct methods for sparse linear systems*, volume 2, Society for Industrial and Applied Mathematics.

[17] **Yannakakis, M.** (1981). Computing the minimum fill-in is NP-complete, *SIAM Journal on Algebraic Discrete Methods*, ***2***(1), 77–79.

[18] **Karypis, G. and Kumar, V.** (1998). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, *Journal of Parallel and Distributed Computing*, ***48***(1), 71–95.

[19] **Chevalier, C. and Pellegrini, F.** (2008). PT-Scotch: A tool for efficient parallel graph ordering, *Parallel Computing*, ***34***(6), 318–331.

[20] **Grigori, L.**, **Demmel, J.W. and Li, X.S.** (2007). Parallel symbolic factorization for sparse LU with static pivoting, *SIAM Journal on Scientific Computing*, ***29***(3), 1289–1314.

[21] **Dongarra, J.**, Freely Available Software for Linear Algebra, `http://www.netlib.org/utk/people/JackDongarra/la-sw.html`, may 2013.

[22] **Li, X.S.**, **Demmel, J.W.**, **Gilbert, J.R.**, **Grigori, L.**, **Shao, M. and Yamazaki, I.** (2011). SuperLU Users' Guide, *Internet Address: http://crd. lbl. gov/˜ xiaoye/SuperLU/superlu_ug. pdf*.

[23] Message Passing Interface (MPI) forum, `http://www.mpi-forum.org`.

[24] **Li, X.S. and Demmel, J.W.** (1999). A scalable sparse direct solver using static pivoting, *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, pp.22–24.

[25] **Trefethen, L.N. and Bau III, D.** (1997). *Numerical linear algebra*, Society for Industrial and Applied Mathematics.

[26] **Gupta, A.** (2000). Parallel Sparse Direct Methods: A short tutorial, **Technical Report**, Technical Report RC 25076, IBM TJ Watson Research Center, Yorktown Heights, NY.

[27] **Duff, I.S. and Koster, J.** (1999). The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, *SIAM Journal on Matrix Analysis and Applications*, ***20***(4), 889–901.

[28] **Duff, I.S. and Koster, J.** (2001). On algorithms for permuting large entries to the diagonal of a sparse matrix, *SIAM Journal on Matrix Analysis and Applications*, ***22***(4), 973–996.

[29] **Amestoy, P.R.**, **Duff, I.S.**, **L'excellent, J.Y. and Li, X.S.** (2001). Analysis and comparison of two general sparse solvers for distributed memory computers, *ACM Transactions on Mathematical Software (TOMS)*, ***27***(4), 388–421.

[30] **Skeel, R.D.** (1980). Iterative refinement implies numerical stability for Gaussian elimination, *Mathematics of Computation*, ***35***(151), 817–832.

[31] **Arioli, M.**, **Demmel, J.W. and Duff, I.S.** (1989). Solving sparse linear systems with sparse backward error, *SIAM Journal on Matrix Analysis and Applications*, ***10***(2), 165–190.

[32] **National Center for High Performance Computing of Turkey (UHeM)**, `http://www.uhem.itu.edu.tr/page.php?id=106`.

[33] **Clint Whaley, R.**, **Petitet, A. and Dongarra, J.J.** (2001). Automated empirical optimizations of software and the ATLAS project, *Parallel Computing*, ***27***(1), 3–35.

[34] **Duran, A.**, **and Saunders, B.** Gen_BLAS Generic Basic Linear Algebra Subroutines in C++,2002, *http://www.math.pitt.edu/˜ahd1/GenBLAS.pdf*.

[35] **GOTO-BLAS-High-Performance, B.** by Kazushige Goto, *See http://www. cs. utexas. edu/users/flame/goto*.

[36] **Davis, T.A. and Hu, Y.** (2009). University of Florida sparse matrix collection.

CURRICULUM VITAE

Name Surname: Mehmet Tunçel

Place and Date of Birth: Istanbul, 01/17/1985

Adress: Informatics Institute, ITU Ayazaga Campus,Maslak-34469, İstanbul, Turkey

E-Mail: mehmet.tuncel@be.itu.edu.tr

B.Sc.: Mimar Sinan Fine Arts University

M.Sc.: Istanbul Technical University

List of Publications and Patents:

- Duran, A., Çelebi, M.S., **Tunçel, M.** and Akaydın, B. (2012). Design and implementation of new hybrid algorithm and solver on CPU for large sparse linear systems PN:283493, *PRACE-2IP white paper, Libraries, WP 43,* (3), 720–755.

- Çelebi, M.S., Duran, A., **Tunçel, M.** and Akaydın, B. (2012). Scalable and improved SuperLU on GPU for heterogeneous systems PN:283493, *PRACE-2IP white paper, Libraries, WP 43,* (3), 720–755.

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- Çelebi M.S., Duran A., **Tunçel M.**, 2012: Scalability of SuperLU solvers for large scale complex reservoir simulations. *SPE and SIAM Conference on Mathematical Methods in Fluid Dynamics and Simulation of Giant Oil and Gas Reservoirs*, September 3-5, 2012 Istanbul, Turkey.