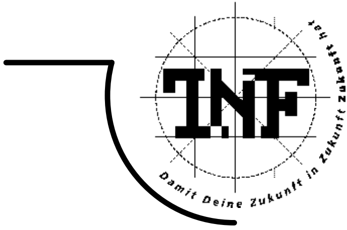
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Parallel Algorithms for Sparse Matrices in an Industrial Optimization Software

Master's Thesis

for obtaining the academic title

Master of Science
in

Internationaler Univerisätslehrgang
Informatics: Engineering & Management

composed at ISI-Hagenberg

Handed in by:
Kenji Miyamoto, 0756088
Fnished on: 3rd July, 2008

Scientific Management:
A.Univ.-Prof. DI Dr.Wolfgang Schreiner

Grade:
Name of Grader: *A.Univ.-Prof. DI Dr.Wolfgang Schreiner*

Hagenberg, July, 2008

Abstract

Optimization problems have an important role in industry, and the finite element method is a popular solution to solve optimization problems numerically. The finite element method rely on linear algebra, high performance linear equation solvera are there important topic in applied science.

In this thesis, we study the high performance forward/backward substitution method by means of parallel computing. We implement various solutions and benchmark each of them in detail on two computers with different hardware architectures; one is a shared memory multicore machine, and the other one is a multicore machine with virtual shared memory machine. We tried two programming models, MPI and POSIX threading, and find the difference of these programming models by run benchmarks on two computers. We achieve good speedup by integrating some solutions.

Acknowledgments. The author is grateful to Prof. Dr. Wolfgang Schreiner for his advice.

The topic of this thesis is motivated by Dr. Peter Stadelmeyer and Dr. Gabor Bodnar of the RISC Software GmbH. A lot of advice were given by them.

I owe my master's study at the International School for Informatics to Prof. Dr. Bruno Buchberger.

Contents

1 Introduction **3**

2 State of the Art **5**
2.1 Background 5
2.2 Linear Algebra 6
2.2.1 Basic Problems 6
2.2.2 Sparse Linear Algebra 8
2.2.3 Software for Linear Algebra 10
2.2.4 Software for Sparse Linear Algebra 11
2.3 Parallel Linear Algebra 11
2.3.1 Parallel Algorithms 11
2.3.2 Parallel Algorithms for Sparse Linear Algebra 12
2.3.3 Parallel Software for Linear Algebra 12
2.3.4 Parallel Software for Sparse Linear Algebra 12

3 Parallel Computation of Multiple Forward/Backward Substitutions **14**
3.1 The Basic Algorithm 14
3.2 Data Representation 15
3.3 Parallel Computation of Multiple Substitutions 17
3.4 Implementation in MPI 17
3.4.1 Implementation without Considering the L/B 18
3.4.2 Implementation with Considering the L/B 21
3.5 Implementation in POSIX Threading 21
3.5.1 Implementation without Considering the L/B 23
3.5.2 Implementation with Considering the L/B 24
3.5.3 Implementation with the L/B and Blocking 24
3.6 Benchmarking of the Implementations 27
3.6.1 The Parallel Computation Solver by using MPI 27
3.6.2 The Parallel Computation Solver by Using POSIX Threading . 28
3.6.3 Conclusions 32

4 Parallelization of the Forward/Backward Substitution Method **38**
4.1 Basic Idea 38
4.2 Data Distribution for the Parallel F/B Substitution Method 39
4.3 Implementation of the Parallel F/B Substitution in MPI 41
4.4 Implementation of the Parallel F/B Substitution in POSIX Threading . 42
4.5 Implementation of the Parallel F/B Substitution in MPI with Blocking 44
4.6 Benchmarking of the Implementation 45

4.6.1 The Solver by using MPI 45
4.6.2 The Solver by POSIX Threading 47
4.6.3 Conclusions 47

5 Parallel F/B Substitution for Sparse Matrices 49
5.1 Generating Task Lists 49
5.2 Joining Task Lists 51

6 Conclusion 53

A Source Code 54
A.1 Organization of Source Code 54
A.2 Source Code 55
A.2.1 Header File 55
A.2.2 Common Routines 55
A.2.3 Sequential Solver 63
A.2.4 Sequential Multiple Substitutions 65
A.2.5 Parallel Multiple Substitutions in MPI 66
A.2.6 Parallel Multiple Substitutions in POSIX Threading 68
A.2.7 Parallel Multiple Substitutions with L/B in MPI 71
A.2.8 Parallel Multiple Substitutions with L/B in POSIX Threading . 73
A.2.9 Parallel Multiple Substitutions with L/B and Blocking in POSIX Threading 76
A.2.10 Parallelized Forward Substitution in MPI 79
A.2.11 Paralellized Forward Substitution in POSIX Threading 82

Bibliography 87

Chapter 1

Introduction

Background Since the role of sparse matrices is a key in manufacturing industry, a lot of high performance algorithms for large sparse matrices are studied by mathematicians and computer scientists. Demands for developing high performance software for sparse matrices are great not only for science but also for industry. Progress of computer software enables to handle large scale algorithmic problems, which makes a contribution towards the further development of industry.

This thesis is motivated by an industrial application for airframe design which requires the time critical implementation for an optimization problem. The optimization problem is given by partial differential equations. A certain software company has their eye upon parallel computing to improve their optimization software, so that the software satisfies higher demands.

Our Goal and Approach Our goal is to clarify the efficiency of applying parallel computing to the optimization problem by implementing and benchmarking. We focus on the forward/backward substitution method [2], because the software solves partial differential equations numerically by means of the finite element method. The problem is reduced to solving sparse linear equations specified by one symmetric positive definite sparse matrix and a lot of right hand sides; the equation can be described by triangular matrices as a result of the Cholesky factorization method. We investigate the efficiency of parallelizing the forward/backward substitution method by comparing several kinds of implementation with detailed benchmarking results. We ran computing benchmarks on two computers with different hardware architectures. One is a usual multicore machine with shared memory, and the other one is a multicore machine with virtual shared memory and high memory bandwidth. [14, 13]

Our Contributions Our original work is to try to achieve speedup of the forward/backward substitution method by two ideas. One is to parallelize the computation of multiple forward/backward substitutions. The other one is to parallelize the forward/backward substitution method itself. We implemented variants of each idea by using MPI as well as POSIX threading, and benchmarked them in detail. We consider the efficiency of all implementations based on benchmarking results, and give discussions on our ideas. We also investigate a further idea for the parallel implementation of the sparse linear equation solver by means of the forward/backward substitution method, although it has not been implemented yet.

Structure of the thesis The structure of the rest of the thesis is as follows. In Chapter 2, we investigate the state of the art. In Chapter 3, we describe the parallel computation of multiple forward/backward substitutions with its implementation and benchmarking results. In Chapter 4, we describe a parallelization of the forward/backward substitution method in the same way to the previous chapter. In Chapter 5, we investigate the idea for the parallel forward/backward substitution method optimized for sparse matrices. In Chapter 6, we give concluding remarks of the thesis.

Chapter 2

State of the Art

2.1 Background

In this section, we tell of the background and context of this thesis.Our research topic is to develop the high performance software implementation for the optimization problem in industry. The RISC Software GmbH develops mathematical and numerical software solutions for industrial engineering by means of advanced scientific results as well as software technologies. They provide solutions for simulation, analysis and optimization systems.

A customer of the RISC Software GmbH (we are not allowed to give its name) has used the optimization software system developed by a software company for air frame design. Now, the RISC Software GmbH supports the customer for their further development and maintenance of the optimization software.

Therefore, the RISC Software GmbH has developed an updated version of the software system. The purpose of this thesis is to develop parallelized algorithms for the software and to implement them to be a prototype of the future version of the software.

The software is an optimization software system based on finite element methods to improve structures with regard to design options, and it is based on structural mechanics and aerodynamics. To obtain more detailed results, the software is continuously extended by means of new analysis and optimization methods. Now, it is necessary to replace some central computational routines of the software for new algorithms so that the improvements of the software satisfy growing demands for more comprehensive modelling.

The software allows design options which include cross sections of certain components or layer thickness and fiber orientation in new laminate materials. The software minimizes the weight of the structure under restrictions such as specific manufacturing restriction for fibrous composite materials, auto-oscillation, and flutter speed. Furthermore, the software allows to model based on a few thousand design parameters and countless boundary conditions.

In optimization procedures, we need to solve partial differential equations numerically. Since the finite element method is used in the software, the problem is discretized and specified by the sparse linear system with a lot of right hand sides. This sparse linear system is always given by the symmetric and positive definite sparse matrix, so that the Cholesky factorization method [3] is the central step of the procedure. As a result, the problem is reduced to solving the simultaneous linear equation given by

sparse triangular matrices, whose ranks are 100.000 and more, with positive entries on the diagonals. Throughout the optimization procedure, it is necessary to iterate computations on sparse matrices, which is the time critical section of the system. If this portion is parallelized, the productivity of the optimization software improves drastically. Thus, it is efficient to exploit the computational power of modern computer clusters by parallelized algorithms for sparse matrices.

Currently, a sequential implementation in C and FORTRAN is available. The Cholesky factorization is implemented in pure C code.

2.2 Linear Algebra

We describe algorithmic aspects of linear algebra relevant to the software. Especially, we focus on the way to solve simultaneous linear equations.

2.2.1 Basic Problems

In this section, we describe basic problems in linear algebra and algorithms for them. As basic problems, we pick up multiplication of matrices, solving simultaneous linear equations, and solving inverse matrices [2].

To deal with matrices, multiplication of matrices is a fundamental operation. The Coppersmith-Winograd algorithm is the fastest known algorithm for square matrix multiplication [1]. By using this algorithm, we can multiply two $n \times n$ matrices in $O(n^{2.376})$ time. This algorithm is not used in practice because there are hidden huge constants in Big O notation. The Strassen algorithm is the other well known algorithm for square matrix multiplication [16]. We can multiply two $n \times n$ matrices in $O(n^{\log_2 7})$ time by the Strassen algorithm. There are also hidden big constants in Big O notation, but it works efficient if n is approximately greater than 45 and matrices are dense. Otherwise, this algorithm is not efficient.

Solving simultaneous linear equations is necessary in various application fields. The problem is to find some $\boldsymbol{x}$ for given A and $\boldsymbol{b}$ such that

$$A\boldsymbol{x} = \boldsymbol{b}.$$

where A is a matrix of type $\mathbf{R}^{n \times n}$ and $\boldsymbol{b}$ is a vector of type $\mathbf{R}^n$. For solving this problem, we may use the LUP decomposition method and the forward/backward substitution method.

In the *LUP decomposition method*, we calculate three matrices L, U, and P which satisfy

$$PA = LU$$

and

1. L is a lower triangle unit matrix,
2. U is an upper triangle matrix, and
3. P is a permutation matrix.

We call these three matrices L, U, and P an LUP decomposition of A. It is known that there exists an LUP decomposition for all regular matrices A. To find $\boldsymbol{x}$, we multiply both sides of $A\boldsymbol{x} = \boldsymbol{b}$ by P, then we have $PA\mathbf{x} = P\boldsymbol{b}$. Next, we have $LU\boldsymbol{x} = P\boldsymbol{b}$ by $PA = LU$. Now we solve two equations:

1. $L\boldsymbol{y} = P\boldsymbol{b}$ by the forward substitution method, and
2. $U\boldsymbol{x} = \boldsymbol{y}$ by the backward substitution method.

We can solve

$$L\boldsymbol{y} = P\boldsymbol{b}$$

by the *forward substitution method* in $\Theta(n^2)$ time. We represent a permutation matrix P by an array $\pi[1..n]$. $\pi[i]$ means $P_{ij} = 0$ if $P_{i,\pi[i]} = 1$ and $j \neq \pi[i]$. So, we have $(PA)_{ij} = a_{\pi[i],j}$ and $(P\boldsymbol{b})_i = b_{\pi[i]}$. $L\boldsymbol{y} = P\boldsymbol{b}$ is written as follows:

$$\begin{array}{rcl} y_1 &=& b_{\pi[1]}, \\ l_{21}y_1 + y_2 &=& b_{\pi[2]}, \\ l_{31}y_1 + l_{32}y_2 + y_3 &=& b_{\pi[3]}, \\ &\dots& \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \dots + y_n &=& b_{\pi[n]}. \end{array}$$

We substitute $y_1 = b_{\pi[1]}$ for $l_{21}y_1 + y_2 = b_{\pi[2]}$, and have $y_2 = b_{\pi[2]} - l_{21}b_{\pi[1]}$. After that, we substitute y_1 and y_2 for $l_{31}y_1 + l_{32}y_2 + y_3 = b_{\pi[3]}$, and have $y_3 = b_{\pi[3]} - (l_{31}b_{\pi[1]} + l_{32}(b_{\pi[2]} - l_{21}b_{\pi[1]}))$. As the above procedure, we can solve each y_i by forwarding substitutions, and eventually we have

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$$

The *backward substitution method* is similar to the forward substitution method, but we solve x_n at first. We can solve $U\boldsymbol{x} = \boldsymbol{y}$ in $\Theta(n^2)$ time. $U\boldsymbol{x} = \boldsymbol{y}$ is written as follows:

$$\begin{array}{rcl} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &=& y_1, \\ u_{22}x_2 + \cdots + u_{2n}x_n &=& y_2, \\ &\dots& \\ u_{nn}x_n &=& y_n \end{array}$$

We have each x_i as

$$\begin{array}{rcl} x_n &=& y_n/u_{nn}, \\ x_{n-1} &=& (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}, \\ x_{n-2} &=& (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2}, \\ \dots && \end{array}$$

Therefore, we have

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii}.$$

We can solve simultaneous linear equations by the above method in $\Theta(n^2)$ time. The forward/backward substitution method is also main issue of our thesis. We are going to parallelize this method and to implement it.

We sometimes need to solve inverse matrices. The LUP decomposition can be applied to solve inverse matrices. Now, we find A^{-1}. We assume that we have already solved P, L, and U which satisfy $PA = LU$. We can solve $A\boldsymbol{x} = \boldsymbol{b}$ in $\Theta(n^2)$ time. Because we have had P, L, and U, we can solve k equations of the form $A\boldsymbol{x} = \boldsymbol{b}$ for fixed A in $\Theta(kn^2)$ time. Now, we solve an equation

$$AX = I_n$$

It is a set of equations of the form $A\boldsymbol{x} = \boldsymbol{b}$. We can find X by solving

$$AX_i = e_i$$

for each i. X_i can be solved in $\Theta(n^2)$ time, so X can be solved in $\Theta(n^3)$ time. It takes $\Theta(n^3)$ time to solve L, U, and P of A, we can solve A^{-1} in $\Theta(n^3)$ time.

If A is a symmetric positive definite matrix, we may factorize A by the *Cholesky factorization* [3]. It is the product $LL^T = A$ where L is a lower triangular matrix with positive entries on its diagonal. The Cholesky factorization is twice more efficient than LUP decomposition. We give a simple way to compute the Cholesky factorization $LL^T = A$. Assume L and A are n-by-n matrices. Let us consider a 2-by-2 block decomposition as following:

$$\begin{bmatrix} L_{11} & \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ & l_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix},$$

where L_{11} and A_{11} are $(n-1)$-by-$(n-1)$ matrices, and l_{12} and a_{12} are $(n-1)$ vectors. The above equation leads three equations:

$$\begin{aligned} L_{11}L_{11}^T &= A_{11} \\ L_{11}l_{12} &= a_{12} \\ l_{12}^T l_{12} + l_{22}^2 &= a_{22}. \end{aligned}$$

The first equation can be solved recursively. The second equation can be solved by the forward substitution. The third equation can be solved as $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$. Because the matrix is positive definite, we hold $a_{22} > l_{12}^T l_{12}$, and the Cholesky factorization exists.

2.2.2 Sparse Linear Algebra

We describe an algorithm to solve simultaneous linear equations $A\boldsymbol{x} = \boldsymbol{b}$ where A and $\boldsymbol{b}$ are sparse matrix and vector respectively. This section is based on the textbook [3]. We have methods to represent A by triangular matrices, so we can focus on solving triangular systems.

We can solve $L\boldsymbol{x} = \boldsymbol{b}$ by following pseudo code:

$\boldsymbol{x} = \boldsymbol{b}$
for $j = 0$ to $n-1$ do
  $x_j = x_j / l_{jj}$
  for each $i > j$ for which $l_{ij} \neq 0$ do
    $x_i = x_i - l_{ij} x_j$

If L has a unit diagonal, the line $x_j = x_j/l_{jj}$ can be omitted. We can improve the above algorithm by modifying the first for loop by assuming a list $\mathcal{X} = \{j \mid x_j \neq 0\}$ which is sorted in ascending order. Then, the algorithm would be improved as following:

$\boldsymbol{x} = \boldsymbol{b}$
for each $j \in \mathcal{X}$ do
  for each $i > j$ for which $l_{ij} \neq 0$ do
    $x_i = x_i - l_{ij} x_j$

Now, the problem is how to determine $\mathcal{X}$ and how to sort it. Entries in $\boldsymbol{x}$ would be nonzero at the first and the last lines. There are two conditions for entries in $\boldsymbol{x}$:

1. $b_i \neq 0 \Rightarrow x_i \neq 0$.
2. $x_j \neq 0 \wedge \exists i . l_{ij} \neq 0 \Rightarrow x_i \neq 0$.

They are drawn as Figure 2.1. We express these two statements as a graph traversal

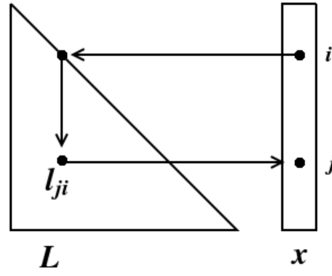


Figure 2.1: Sparse triangular solve

problem. Let $G_L(V, E)$ be a directed graph where $V = \{1, \ldots, n\}$ and $E = \{(j, i) \mid l_{ij} \neq 0\}$ [17]. Note that we neglect numerical cancellations, and the graph is acyclic. If we mark nodes in G_L corresponding to nonzero entries in $\boldsymbol{x}$, the first clause makes all nodes $i \in \mathcal{B}$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$. The second clause says that if node j is marked, and there is an edge from node j to node i, then node i is marked. Finally, we set $\mathcal{X} = \mathrm{Reach}_{\mathrm{G_L}}(\mathcal{B})$. We have following theorem, see [5].

Theorem 2.1 (Gilbert and Peierls) *Define the directed graph $G_L = (V, E)$ with nodes $V = \{1, \ldots, n\}$ and edges $E = \{(j, i) \mid l_{ij} \neq 0\}$. Let* $\mathrm{Reach}_L(i)$ *denote the set of nodes reachable from node i via paths in G_L, and let* $\mathrm{Reach}(\mathcal{B})$*, for a set $\mathcal{B}$, be the set of all nodes reachable from any node in $\mathcal{B}$. The nonzero pattern $\mathcal{X} = \{j \mid x_j \neq 0\}$ of the solution $\boldsymbol{x}$ to the sparse linear system $L\boldsymbol{x} = \boldsymbol{b}$ is given by $\mathcal{X} = \mathrm{Reach}_L(\mathcal{B})$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$, assuming no numerical cancellation.*

The set $\mathcal{X}$ can be computed by a depth-first search of the graph G_L, starting at nodes in $\mathcal{B}$. The set is always in topological order. In this case, the set $\mathcal{X}$ is sorted in ascending order. So far, we achieved $\mathcal{X}$ for which the improved algorithm can be applied.

We give an example for solving $L\boldsymbol{x} = \boldsymbol{b}$. The graph G_L is in Figure 2.2. Let

$$L = \begin{bmatrix} 1 & & & & \\ 7 & 2 & & & \\ 7 & & 3 & & \\ & 7 & 7 & 4 & \\ & & & 7 & 5 \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} \\ 8 \\ 8 \\ \\ \\ \end{bmatrix}$$

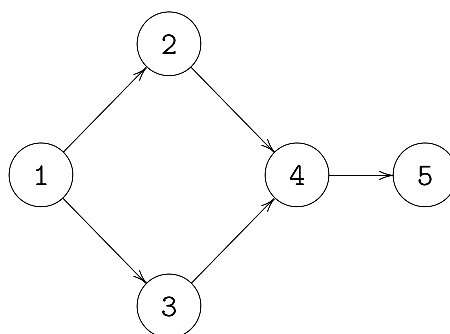


Figure 2.2: Graph G_L

Now, we calculate Reach(2) $= \{2, 4, 5\}$ by G_L, and mark each nodes. Next, we calculate Reach(3) $= \{3, 4, 5\}$ same. The node 4 and 5 is already marked, so we add $\{2, 4, 5\}$ to the empty list $\mathcal{X}$ and after that, we add only $\{3\}$ to the list $\mathcal{X}$. Obviously, the list $\mathcal{X} = \{3, 2, 4, 5\}$ is in topological order, and the forward substitution works well. The work done at column 3 is not affected by the work done at column 2.

2.2.3 Software for Linear Algebra

We describe software for linear algebra.

LINPACK[1] is a collection of subroutines which analyze and solve linear equations and least-square problems, and written in Fortran. LINPACK is not efficient on shared memory vector and parallel processors because of its memory access pattern. LINPACK is largely replaced by LAPACK.

LAPACK[2] is software written in Fortran77 which provides routines for solving simultaneous linear equations, eigenvalue problems, singular value problems, and etc. LAPACK also provides LU, Cholesky, and other factorization methods. LAPACK is optimized for shared memory vector and parallel processors.

LAPACK and LINPACK are implemented as efficient as possible by calls to the *BLAS*[3] (Basic Linear Algebra Subprograms). BLAS consists of three packages. BLAS level 1 performs scaler, vector, and vector-vector operations, BLAS level 2 performs matrix-vector operations, and BLAS level 3 performs matrix-matrix operations. BLAS provides basic operations implemented in Fortran77 with C interface. For example, it has functionalities to calculate following formulae:

- $\alpha\boldsymbol{x} + \boldsymbol{y}$, where α is a scalar, and $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors.

[1] http://www.netlib.org/linpack/
[2] http://www.netlib.org/lapack/
[3] http://www.netlib.org/blas/

- $\alpha \boldsymbol{x}\boldsymbol{x}^T + A$, where α, $\boldsymbol{x}$, and A are a scalar, an n-vector, and an n-by-n matrix, respectively.
- $\alpha A^T B + \beta C$, where α is a scalar, and A, B and C are l-by-n, l-by-m, and n-by-m matrices, respectively.

Some efficient machine specific implementations of BLAS are also available.

2.2.4 Software for Sparse Linear Algebra

CSparse[4] is a small package for sparse linear algebra, especially for a textbook [3]. CSparse is about 2200 lines of program in C which contains the LU factorization, the Cholesky factorization, linear equations solvers, and etc. An interface to MATLAB is also provided.

CHOLMOD[5] is C routines for sparse Cholesky factorization which is more efficient than CSparse. An interface to MATLAB is also provided.

2.3 Parallel Linear Algebra

In this section, we describe parallel linear algebra. First, we give a brief introduction to parallel algorithm. Next, we describe parallel algorithms for sparse linear algebra, and software and tools for them.

2.3.1 Parallel Algorithms

Parallel algorithms are developed by using independences among data to be processed. For example, we can easily develop a parallel algorithm to compute a matrix-vector multiplication:

For given A and $\boldsymbol{x}$, find $\boldsymbol{y}$ such that $\boldsymbol{y} = A\boldsymbol{x}$.

Obviously, each entry of $\boldsymbol{y}$ is independent from other entries of $\boldsymbol{y}$, so we can compute all entries in parallel.

On the other hand, it is difficult to parallelize algorithms which process data with dependences. For example, we consider an algorithm which computes x_{10000}, defined by recurring formula:

$$x_{n+1} = \begin{cases} \alpha x_n & x_n < \frac{1}{2} \\ \alpha(1 - x_n) & \frac{1}{2} \leq x_n \end{cases} \qquad \text{for } 0 \leq x_0 \leq 1 \text{ and } 0 < \alpha.$$

To compute x_{i+1}, it is necessary to know x_i. So, this algorithm is sequential, or we have to find other property of x_n to parallelize.

[4] http://www.cise.ufl.edu/research/sparse/CSparse/
[5] http://www.cise.ufl.edu/research/sparse/cholmod/

2.3.2 Parallel Algorithms for Sparse Linear Algebra

We briefly describe a parallel forward substitution for symmetric positive definite sparse linear systems. We give details of the parallel forward/backward substitution in Chapter 4. At the beginning, we compute the elimination tree of the matrix.

Theorem 2.2 *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellations, $a_{ij} \neq 0 \Rightarrow l_{ij} \neq 0$.*

Theorem 2.3 (Parter [9]) *For a Choesky factorization $LL^T = A$, and neglecting numerical cancellations, $i < j < k \wedge l_{ji} \neq 0 \wedge l_{ki} \neq 0 \Rightarrow l_{kj} \neq 0$.*

To compute Reach(i), we do not need the edge (i, k) if there is a path from i to k via j. After removing all edges as above, we have the elimination tree as a result. To compute the elimination tree, one more theorem is necessary.

Theorem 2.4 (Schreiber [11]) *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $l_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.*

We can do the forward substitution method in parallel from leaves to the root of the elimination tree [7].

2.3.3 Parallel Software for Linear Algebra

In this section, we describe parallel software for linear algebra.

ScaLAPACK[6] is a project to provide parallel implementations for linear algebra. This project provides *ScaLAPACK* [7] as dense and band matrix software. Especially, ScaLAPACK is designed for heterogeneous computing and implemented by using MPI and PVM for portability. *PBLAS* [8] provides the fundamental operations of ScaLAPACK. PBLAS is a distributed memory version of BLAS that has similar functionalities to BLAS. *ARPACK*[9] is software for solving large scale eigenvalue problems. This software is suitable for large sparse matrices or for a matrix A which require $O(n)$ floating point operations but $O(n^2)$ operations when we calculate a matrix-vector product $A\boldsymbol{x}$.

PLAPACK[10] is a software package of parallel linear algebra. PRAPACK provides parallel solvers by using Cholesky, LU, and QR factorization. This package is implemented in a sophisticated way so it works highly efficiently. PLAPACK shows higher performance than ScaLAPACK.

2.3.4 Parallel Software for Sparse Linear Algebra

We describe specialized software and tools for parallel sparse linear algebra.

The ScaLAPACK project also provides software and tools for sparse linear algebra.

MFACT [11] is a specialized solver for symmetric and positive definite linear systems. The current version is just a first release and it has a simple interface. This project is

[6] http://www.netlib.org/scalapack/index.html
[7] http://www.netlib.org/scalapack/scalapack_home.html
[8] http://www.netlib.org/scalapack/html/pblas_qref.html
[9] http://www.caam.rice.edu/software/ARPACK/
[10] http://www.cs.utexas.edu/users/plapack/
[11] http://www.cs.utk.edu/~padma/mfact.html

ongoing and MFACT will be expanded to solve also indefinite systems. The project is developing a version for shared memory multi processors too.

PARPACK [12] is software for solving large sparse eigenvalues. PARPACK is a parallel version of ARPACK.

PETSc [13] is a collection of lots of parallel scientific software which includes not only software for sparse matrices but also solvers for nonlinear equations, ODEs, PDEs, and etc. Interfaces for C, C++, Fortran, and Python are provided.

[12] http://www.caam.rice.edu/~kristyn/parpack_home.html
[13] http://www-unix.mcs.anl.gov/petsc/petsc-as/index.html

Chapter 3

Parallel Computation of Multiple Forward/Backward Substitutions

We describe a parallel computation of the forward/backward substitution method (called "f/b substitution method") [15]. First, we review the sequential f/b substitution method. Next, we describe the parallel computation of the f/b substitution method.

In the optimization procedure, it is required to solve equations given by one sparse matrix and a lot of right hand sides. Therefore, distributing right hand sides among processors is expected to achieve high performance on multi processor machines. However, if each task is too small, we cannot get a good performance because overheads of parallel computing can seriously affect the total computing time. We therefore make the granularity of tasks big enough such that the overhead is outweighed by the benefits of parallel computing.

3.1 The Basic Algorithm

The f/b substitution method solves simultaneous linear equations described as:

$$A\boldsymbol{x} = \boldsymbol{b}$$

where A and $\boldsymbol{b}$ are of type $\mathbf{R}^{n\times n}$ and $\mathbf{R}^n$ respectively, and A is a lower or an upper triangular matrix. The forward substitution method solves equations specified by a lower triangular matrix, and the backward substitution method solves one of specified by an upper triangular matrix.

Now we assume A is a lower triangular matrix. Then $A\boldsymbol{x} = \boldsymbol{b}$ is written as follows:

$$\begin{aligned}
a_{11}x_1 &= b_1, \\
a_{21}x_1 + a_{22}x_2 &= b_2, \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3, \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n.
\end{aligned}$$

The solving procedure is as follows:

1. To find x_1 of the first equation.

2. To substitute the value of x_1 for x_1 in the second equation and find x_2.

3. To substitute the values of x_1 and x_2 for x_1 and x_2 in the third equation respectively, and find x_3.

4. To iterate the above procedure until finding x_n.

On the other hand, the backward substitution method solves simultaneous linear equations if A is an upper triangular matrix. We assume A is an upper triangular matrix. Then $A\boldsymbol{x} = \boldsymbol{b}$ is written as follows:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1,n-2}x_{n-2} + a_{1,n-1}x_{n-1} + a_{1n}x_n &= b_1, \\ &\vdots \\ a_{n-2,n-2}x_{n-2} + a_{n-2,n-1}x_{n-1} + a_{n-2,n}x_n &= b_{n-2}, \\ a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n &= b_{n-1}, \\ a_{nn}x_n &= b_n. \end{aligned}$$

The solving procedure is essentially same to the forward substitution method, but done in the reverse order.

1. To find x_n of the nth equation.

2. To substitute the value of x_n for x_n in the *(n-1)*th equation and find x_{n-1}.

3. To substitute the values of x_n and x_{n-1} for x_n and x_{n-1} in the *(n-2)*th equation respectively, and find x_{n-2}.

4. To iterate the above procedure until finding x_1.

The sequential solver is described by the pseudo-code shown in Figure 3.1.

3.2 Data Representation

In our implementation, we represent matrices by using a data structure [3] shown in Figure 3.2.

`colptr` is an array whose size is n+1. It has numbers such that `colptr`$[i]-$`colptr`$[i-1]$ denotes the number of entries in the ith column for $1 \leq i \leq$ `n`. So, `colptr`[0] is always 0 and `colptr`$[\mathtt{n}+1]$ is the number of entries of the matrix. `rowind` is an array whose size is `nzmax`. It has numbers to describe row indices of each column. For example, we can represent matrix A by two arrays:

$$A = \begin{bmatrix} a_{11} & & & a_{14} \\ & a_{22} & & \\ a_{31} & a_{32} & & \\ & a_{42} & & a_{44} \end{bmatrix}$$

```
int[5] colptr = {0,    2,       5, 5,   7}
int[7] rowind = {0, 2, 1, 2, 3,    0, 3}
```

The values of the array `val` are $\{\ a_{11}, a_{31}, a_{22}, a_{32}, a_{42}, a_{14}, a_{44}\ \}$.

```
SequentialForwardSubstitution(A, b, An, myid)

 input    A: partial matrix
          b: right hand side
          An: the size of the original matrix
          myid: the index of this processor
 output   b solution

for rowind = 0 to An do:
  colind_mat = get_matrix_index(myid, A->i[A->p[rowind]])
    // computing the index of the original matrix by one of split matrix

  if colind_mat = rowind do: // on the diagonal
    b[colind_mat] := b[colind_mat] / A->x[A->p[rowind]]
  done

  for ind_part = A->p[rowind] to A->p[rowind+1] do:
    colind_mat := get_matrix_index(myid, A->i[ind_part])

    if colind_mat > rowind do:
      // b[colind_mat] is necessary to be updated
      b[colind_mat] -= A->x[ind_part]*b[rowind]
    done
  done
done
```

Figure 3.1: The Sequential Equation Solver

```
typedef struct cs_sparse
{
  int nzmax; /* maximum number of entries */
  int m; /* number of rows */
  int n; /* number of columns */
  int* colptr; /* column pointers */
  int* rowind; /* row indices */
  double* val; /* numerical values */
  int nz; /* number of entries */
} cs;
```

Figure 3.2: Data structure of matrix

3.3 Parallel Computation of Multiple Substitutions

In this section, we describe why and how to execute substitutions in parallel.

During the optimization procedure, it is required to solve a lot of simultaneous linear equations $L\boldsymbol{x} = \boldsymbol{b}$ given by one large sparse matrix L and many right hand sides $\boldsymbol{b}_i$. Therefore, we can execute the equation solver in parallel by distributing the given matrix and right hand sides among processors. As described at the beginning of this chapter, we cannot get speedup if distributed tasks are too small. We expect to get a good performance because solving a simultaneous linear equation is a large task, and parallelizing it is more effective than overheads.

First, we distribute the given large sparse matrix A among processors as shown in Figure 3.3. Then, all processors share the matrix. After that, we distribute the right

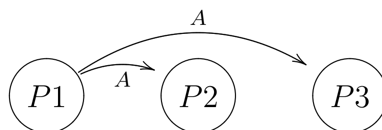


Figure 3.3: Broadcasting the given matrix A

hand sides among the processors as shown in Figure 3.4. (in the figure, we assume *P1* is

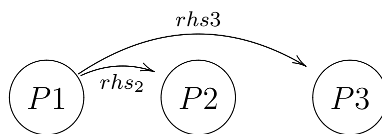


Figure 3.4: Distributing right hand sides

the root processor which reads files containing the matrices and the right hand sides.) The preparation procedure has been done so far.

Now, we solve the linear equations in parallel. Each processor iteratively solves equations with the assigned right hand sides; at last, the solutions are collected as shown in Figure 3.5.

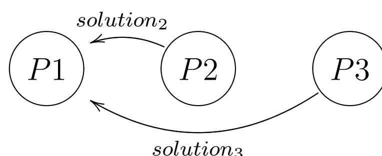


Figure 3.5: Collecting Solutions

3.4 Implementation in MPI

In this section, we describe two implementations of the parallel computation of multiple substitutions in MPI [4].

First, we describe the implementation without considering the load balance. In the given set of right hand sides, some of them take more time than other ones in the

solution of the equation. As a result, time spent by processors becomes unbalanced, and computational resources are not used in an efficient way.

Next, we try to manage the load balance (called "the L/B") among processors [12]. In our solution, one processor has a role as a task manager; it assigns right hand sides to each worker processors such that every worker receives the same share of load.

At last, we try to reduce overhead costs for the synchronization of POSIX threading solution. We assign more than one right hand sides to each worker thread at one time [15]. We call this solution the blocking.

3.4.1 Implementation without Considering the L/B

We implemented the parallel computation of multiple substitutions without considering the l/b. The procedure is listed below:

1. Broadcasting the given matrix A among processors.
2. Splitting right hand sides.
3. Distributing split right hand sides among processors.
4. Each process solves assigned simultaneous equations in parallel.
5. Gathering solutions.

Broadcasting the matrix is done by invoking `MPI_Bcast`. It is done by the pseudo-code shown in Figure 3.6. In this code, each member of the structure `cs` is broadcasted. `cs` is shown in Figure 3.2.

```
DistributeMatrixBcast(A)

 input   A: matrix

MPI_Bcast(A->nzmax)
MPI_Bcast(A->m)
MPI_Bcast(A->n)
MPI_Bcast(A->nz)
MPI_Bcast(A->p)
MPI_Bcast(A->i)
MPI_Bcast(A->x)
```

Figure 3.6: Distribute matrix by MPI_Bcast

In the next procedure, for available n processors, right hand sides are split into n sets of vectors. In this implementation, the ith right hand side is assigned to the $i\%n$th processor, where n is the number of worker processors. After that, Distributing split right hand sides are done by invoking `MPI_Send` and `MPI_Recv`. This procedure is done by two pseudo-code shown in Figure 3.7 and Figure 3.8, as depicted in Figure 3.9. In this figure, we assumed there are three available processors, and the number of right hand sides is seven.

Solving procedure is just done by sequential equation solver by each processor. The solver is shown in Figure 3.10.

```
DistributeRhsSend(rhs, length, num)

 input    rhs: an array of right hand sides
          length: the length of a right hand side
          num: the number of right hand sides
          mysize: the number of worker processors
 output   ret: an array of rhs for the root processor

index = 0 // index for the array to return
num_assign = number_of_rhs_assigned(num, 0) // for the root processor
for i = 0 to num-1 do:
  proc = i % mysize // now ith rhs is assigned to proc
  if proc ≠ 0 do: // proc is not the root processor
    MPI_Send(rhs[i], proc) // assign rhs[i] to proc
  else:
    ret[index] = rhs[i] // assign rhs[i] to the root processor
    index = index+1
  done
done
```

Figure 3.7: Distributing right hand sides (send)

```
DistributeRhsRecv(length, num)

 input    length: the length of a right hand side
          num: the number of right hand sides
          myid: the index for this process
 output   ret: the array of rhs received

num_assign = number_of_rhs_assigned(num, myid)
  // the number of right hand sides assigned to this processor

for i = 0 to num_assign do:
  MPI_Recv(array[i]) // receiving num_assign times
done
```

Figure 3.8: Distributing right hand sides (receive)

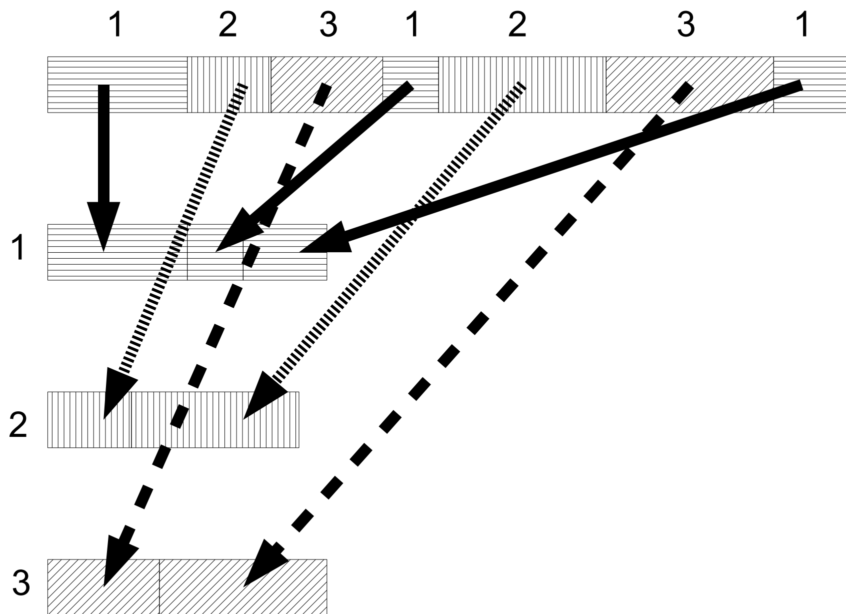


Figure 3.9: Distributing right hand sides

ForwardSubstitutionSequential(`A, b, length`)

input	`A`: matrix
	`b`: right hand side
	`length`: the length of a right hand side
output	`b`: the solution

```
for rowind = 0 to length do:
  colind = A->i[A->p[rowind]]
    // column index of the first nonzero element of the row rowind
  if (colind == rowind) do: // on the diagonal
    b[colind] /= A->x[A->p[rowind]] // computing a solution
  done

  for i = A->p[rowind] to A->p[rowind+1] do:
    colind = A->i[rowind[i]]
    if (colind > rowind) do:
      // index indicates the lower side of the diagonal
      b[colind] -= A->x[i]*b[rowind] // updating
    done
  done
done
```

Figure 3.10: Sequential solver

At the last, The root processor gather all solutions from processors by means of `MPI_Send` and `MPI_Recv`.

3.4.2 Implementation with Considering the L/B

As we will show in the benchmarking results in Section 3.6, the previous implementation is problematic. The effect of the parallelization is far from the ideal case, and speedup is not stable. We considered one reason is the load balance among processors. Various right hand sides are given, and sometimes a particular processor receives heavier tasks than others. As a result, this processor seriously affects the total computation time.

For example, we have detailed data for solving M2 by four processors on `speedy`. According to the benchmarking, each processor spend different time for the substitution procedure as shown in Table 3.1.

Processor	0	1	2	3
Subst. Time (sec.)	38.808	34.197	37.641	17.511

Table 3.1: Unbalanced computation time for M2 on `speedy`

In this example, the processor 3 finished its computation around 17 sec., and waited others to collect solutions. To avoid such a situation, we implemented the parallel computation with the l/b. To make the load balanced, one processor operates as a manager of tasks, and the other processors operate as workers. At the beginning, the manager processor assigns tasks to each worker processor. Each worker processor solves the given equation, and gives back a solution to the manager processor. The manager processor assigns the next task to the worker processor which has finished its given task. Therefore, the manager task distributes right hand sides in a balanced way. This implementation requires at least two processors because one processor is used as a manager.

The pseudo-code for the core part is shown in Figure 3.11 and Figure 3.12.

It can be easily understood that the procedure for distributing right hand sides are invoked as needed, and tasks are assigned only to those worker processors which have finished their previous tasks.

3.5 Implementation in POSIX Threading

In this section, we describe three implementations of the parallel computation of multiple substitutions by means of POSIX threading [6].

First, we describe the implementation without the l/b. This is just a POSIX threading variant of the algorithm described in Section 3.4.1.

Next, we modified the above implementation by the l/b. This is also a POSIX threading variant of the implementation in MPI given in Section 3.4.2. The difference from the previous MPI version is that we do not prepare the manager by using one thread. We just use the shared variable as the "manager".

Third, we implemented a blocking solution. The above l/b solution required the synchronization mechanism because we used the shared variable as the manager. To guarantee the mutual exclusive access for the shared variable, the variable is locked

```
Manager(rhs, num_rhs, num_worker)

 input   rhs: array of right hand sides
         num_rhs: the number of right hand sides
         num_worker: the number of worker processors

for i = 1 to num_worker do: // the first assignment
  assign_list[i] = i-1 // store to which processor was assigned which rhs
  MPI_Send(rhs[i-1], i, i-1) // rhs[i-1] is assigned to proc i
done

for i = num_worker-1 to num_rhs do:
  MPI_Recv(rhs_buf, ANY, &stat) // receiving the solution
  swap(rhs[assign_list[stat.MPI_SOURCE]], rhs_buf) // swapping two pointers
  assign_list[stat.MPI_SOURCE] = i

  MPI_Send(rhs[i], status.SOURCE, i) // sending the next rhs
done

for i = 0 to num_worker-1 do: // receiving residual solutions
  MPI_Recv(rhs_buf, ANY, &stat) // receiving the solution
  swap(rhs[assign_list[stat.MPI_SOURCE]], rhs_buf)
done

for i = 1 to num_worker do: // send termination signals
  MPI_Send(rhs_buf, i, num_rhs) // tag = num_rhs means the termination
done
```

Figure 3.11: Core part for the manager process

```
Worker(rhs, num_rhs, num_worker)

 input   myid: index of this worker process
         A: the give matrix
         num_worker: the number of worker processors

MPI_Recv(work_rhs, manager, &stat) // receiving the rhs by manager

while stat.MPI_TAG < num_vec do: // tag = num_vec means the terminal signal
  worker_rhs = solve(A, worker_rhs)

  MPI_Send(worker_rhs, manager, stat.MPI_TAG) // giving back a solution
  MPI_Recv(worker_rhs, manager, &stat) // receiving the next rhs, the tag may be num_vec
done
```

Figure 3.12: Core part for the worker process

while one thread is accessing it. While the variable is locked, other threads which need to access the variable must wait until the variable is unlocked. If threads frequently wait for the shared variable unlocked, the performance decreases. In the blocking solution, more than one right hand sides are assigned to each processor at one time. As a result, the frequency of synchronization should be reduced.

3.5.1 Implementation without Considering the L/B

We implemented the parallel computation of multiple substitutions by means of POSIX threading without the l/b. We do not need to implement the functionality for broadcasting the given matrix because it can be shared by using a global variable. We do not need the functionality to split and distribute right hand sides either. The procedure for this implementation is listed below:

1. Creating threads with assigning their indices.
2. Each thread calculates the number of assigned RHSs.
3. Each thread calculates indices of assigned RHSs.
4. Each thread solves the assigned simultaneous linear equations in parallel.

Threads are created by invoking the function `pthread_create`. We specify a function and its argument as a task for threads. In this implementation, we only pass thread indices as an argument. Each thread calculate their assigned RHSs by using these thread indices.

The function assigned to threads is shown in Figure 3.13.

```
ThreadTask(arg)

 input             arg: parameter
 shared variables  mysize: the number of processors
                   A: matrix
                   rhs: array of right hand sides
                   num_rhs: the number of right hand sides

thread_index = arg[0]
num_assign = number_of_rhs_assigned(num_rhs, thread_index)
  // calculate the number of assignment for this thread

for i = 0 to num_assign do:
  rhs[mysize*i+thread_index] = solve(A, rhs[mysize*i+thread_index], A->m)
done
```

Figure 3.13: Task assigned for each thread

In Figure3.13, `mysize*i+thread_index` is the index of RHSs assigned for this thread. It means the number of assigned right hand sides among threads is balanced in this solution. It does not mean that it is a load balanced task assignment; therefore we implement the l/b solution in Section 3.5.2.

3.5.2 Implementation with Considering the L/B

We implemented the load balanced right hand sides assignment also for POSIX threading solution. The main idea is same as the one presented in Section 3.4.2. When we implemented it in MPI, one processor is devoted to manage the right hand sides assignment. In this POSIX threading solution, the manager thread is not needed. We just use a shared variable to manage the l/b.

The difference of the implementation with the l/b from one without the l/b is listed below:

1. Preparing the function which returns the next right hand side to be solved.
2. In the assigned task function for threads, each thread ask the next RHS to be solved by invoking the above function.
3. Threads finish their assigned task function when all right hand sides are solved.

The function which returns the index of the next right hand side is defined by the pseudo-code shown in Figure 3.14.

```
GetNextRHS()

 shared variables   mutex: object for the mutual exclusion
                    next_rhs: index of the right hand side to be solved the next
 output             ret: the index of right hand side to be solved

mutex_lock(mutex) // critical section
ret = next_rhs
next_rhs = next_rhs+1
mutex_unlock(mutex) // the end of critical section

return ret
```

Figure 3.14: Function to calculate the index of the right hand side to be solved

The function GetNextRHS is invoked in the task function for threads. The task function is described by the pseudo-code shown in Figure 3.15.

3.5.3 Implementation with the L/B and Blocking

As we described at the beginning of this section, there is a critical section in the function `GetNextRHS()`. To decrease the frequency of the synchronization, we implemented the blocking solution. In this implementation, more than one right hand sides are assigned to each threads at one time. We call the set of right hand sides a block. The size of a block is specified by users.

We modified `GetNextRHS` and `thread_task` as shown in Figure 3.16 and Figure 3.17 respectively.

TaskFunction(arg)

input	arg: parameter
shared variables	A: matrix
	rhs: right hand sides
	num_rhs: the number of right hand sides

```
thread_index = arg[0]
i = GetNextRHS()

while i<num_rhs do: // if i ≥ num_rhs, threads finish this function
  rhs[i] = solve(A, rhs[i], A->m)
  i = GetNextRHS() // get the next rhs after solving the assigned equation
done
```

Figure 3.15: Task function with the L/B

GetNextRHS()

shared variables	next_rhs: index for the next right hand side
	block: the size of a block
	mutex: object for the mutual exclusion
output	ret: the first index to be solved

```
mutex_lock(mutex)
ret = next_rhs
next_rhs = next_rhs+block
mutex_unlock(mutex)

return ret
```

Figure 3.16: GetNextRHS with blocking

thread_task(arg)

input	arg: parameter
shared variables	A: matrix
	rhs: right hand sides
	num_rhs: the number of right hand sides
	block: the size of a block

```
thread_index = param[0]
block_begin = GetNextRHS()

while block_begin<num_rhs do:
  if block_begin+block <= num_rhs do:
    block_until = block_begin+block
  else
    block_until = num_rhs
  done

  for j = block_begin to block_until do:
    rhs[j] = solve(A, rhs[j], A->m)
  done

  block_begin = GetNextRHS()
done
```

Figure 3.17: Task function with blocking

3.6 Benchmarking of the Implementations

In this section, we describe the performance of the implementations described in the previous section. We benchmarked the sequential solver and the parallel solver.

The specification of the computer we used for benchmarking the implementation is listed below:

1. Machine Name: `speedy`
   (a) Multi core machine with shared memory
   (b) CPU: Four 2.33 GHz dual core Intel Xeon processors
   (c) Memory: 16.433.856 kB
   (d) GNU C Compiler: gcc version 4.2.1
2. Machine Name: `lilli`
   (a) Multi core machine with virtual shared memory and high memory bandwidth
   (b) CPU: 1.59 GHz Intel processors
   (c) Memory: 1.017.161.392 kB
   (d) Intel C Compiler: icc version 9.1

We used two different matrices and sets of right hand sides as shown in Table 3.2 and Table 3.3.

Matrix Name	Size	Nonzero Entries
M1	19.138-by-19.138	314.243
M2	104.442-by-104.442	2.589.787

Table 3.2: Matrices

Rhs Name	The number of rhs	Nonzero Entries
M1	2.904	161.610
M2	4.617	248.169

Table 3.3: Right hand sides

3.6.1 The Parallel Computation Solver by using MPI

The Solver without the L/B

The results are shown in Table 3.4 and Table 3.5.

We can find that the total time reduces by using many processors. However, the result is not so efficient with regard to the number of processors used. We show the reason why the result is not so efficient.

Table 3.6 shows how much time each processor spent for the substitution procedure on `speedy`.

Matrix	Sequential	Processors	Total	Dist. Mat.	Dist. rhs	Subst.
M1	4.946 (sec.)	1	5.959	9.060×10^{-6}	3.099×10^{-5}	4.763
		2	4.115	0.336	0.761	2.374
		3	4.042	0.336	0.630	2.299
		4	3.595	0.367	0.615	1.857
		5	3.881	0.353	0.656	2.115
		6	3.629	0.356	0.657	1.829
M2	61.263 (sec.)	1	130.371	9.060×10^{-6}	7.391×10^{-5}	61.638
		2	59.225	9.949	4.106	40.186
		3	88.690	2.831	3.192	31.586
		4	135.129	2.888	2.796	38.808
		5	45.219	3.001	3.409	32.416
		6	43.703	3.023	3.357	30.660

Table 3.4: Benchmarking for parallel computation solver by MPI on `speedy`

There is much difference between the processor No. 0 and the processor No. 3. We considered this difference is because of the unbalanced load among processors. The worst case seriously affects the total performance of the algorithm, so this difference is expected to be decreased.

The Solver with the L/B

The results are shown in Table 3.7 and Table 3.8.

As described, splitting and distributing right hand sides are done while worker processors solve equations. So, we did not benchmark time to split and distribute right hand sides. They are included in time for substitution in Table 3.7 and Table 3.8.

As a result of the L/B, the performance is improved very well. Even though one processor is devoted to be the manager, this implementation is efficient when we can use at least four processors. By the above benchmarking result, we can conclude that the L/B is a good solution to improve the performance of the solver.

3.6.2 The Parallel Computation Solver by Using POSIX Threading

We benchmarked three implementations by means of POSIX Threading.

The Solver without the L/B

The benchmarking results are shown in Table 3.9 and Table 3.10. We can find a similar problem as explained in Section 3.6.1. We show more detailed benchmarking result when it has solved the problem of M2 by using eleven threads on `lilli` in Table 3.11.

In this case, the processor No. 1 spent much time, 44.926 seconds, which critically affects the total result. The processor finished the assigned tasks by the least time, 27.121 seconds, was the processor No. 2, and the difference between the processor No. 1 is 17.805 seconds. We implemented the L/B solution to reduce this difference as we did in Section 3.5.2.

Matrix	Sequential	Processors	Total	Dist. Mat.	Dist. rhs	Subst.
M1	7.476 (sec.)	1	16.193	4.755×10^{-5}	1.047×10^{-4}	14.037
		2	9.668	0.729	1.343	7.112
		3	7.174	0.724	1.277	4.718
		4	6.625	0.762	1.455	3.618
		5	5.702	0.737	1.433	2.938
		6	5.079	0.747	1.343	2.384
		7	5.004	0.837	1.370	2.087
		8	4.488	0.752	1.307	1.891
		9	4.272	0.756	1.340	1.340
		10	4.482	0.836	1.513	1.525
		11	4.100	0.774	1.387	1.350
		12	3.977	0.759	1.424	1.259
		13	3.783	0.764	1.353	1.155
		14	3.797	0.768	1.444	1.025
		15	3.966	0.786	1.517	1.068
		16	3.746	0.863	1.366	0.959
M2	213.333 (sec.)	1	488.134	5.290×10^{-5}	1.595×10^{-4}	474.888
		2	169.951	6.093	4.361	155.221
		3	118.973	6.104	4.289	103.504
		4	159.066	6.158	4.717	77.224
		5	106.157	6.230	5.464	86.657
		6	101.517	6.268	5.039	85.150
		7	84.254	6.316	5.344	65.608
		8	58.145	6.291	5.866	38.980
		9	51.781	6.377	6.423	34.393
		10	47.418	6.317	5.167	31.408
		11	61.538	6.398	5.827	44.926
		12	57.353	6.336	5.422	36.654
		13	59.758	6.386	6.339	40.706
		14	69.715	6.482	5.551	51.871
		15	51.152	6.429	6.031	33.341
		16	56.812	6.586	6.964	38.684

Table 3.5: Benchmarking for the parallel computation solver by MPI on `lilli`

Matrix	Processors	Index of Proc.	Substitution
M2	4	0	38.808 (sec.)
		1	34.197
		2	37.641
		3	17.511

Table 3.6: Difference among processors in the case of the MPI solver

Matrix	Sequential	Processors	Total	Dist. Mat.	Subst.
M1	4.946 (sec.)	2	5.580	6.792×10^{-3}	5.573
		3	3.764	1.165×10^{-2}	3.752
		4	2.549	1.645×10^{-2}	2.532
		5	2.341	2.113×10^{-2}	2.319
		6	2.243	2.450×10^{-2}	2.218
M2	61.263 (sec.)	2	70.815	7.708×10^{-2}	70.738
		3	40.076	1.402×10^{-1}	39.936
		4	44.151	1.600×10^{-1}	43.991
		5	34.445	2.548×10^{-1}	34.190
		6	34.366	2.704×10^{-1}	34.095

Table 3.7: Benchmarking for the L/B solver by MPI on `speedy`

The Solver with the L/B

The benchmarking results are shown in Table 3.12 and Table 3.13.

Our observation on the benchmarking results is as follows:

- If we solve equations by not so many threads, like less than six, we can find the L/B solution makes an efficiency.
- If we can use more threads, more than ten for example, the result is not so efficient with regard to the number of threads and the benchmarking result of the sequential one.

Especially, we cannot see big difference by comparing two results on `lilli` as shown in Table 3.10 and Table 3.13.

As we described at the beginning of Section 3.5, This result can be because of the synchronization of threads.

On the other hand, we can find an interesting result by comparing a result of the L/B solution in MPI on `speedy` with one of `lilli`, and a result of the L/B solution in POSIX threading on `speedy` with one of `lilli`. We show an excerpt of some data in Table 3.14.

The table shows that POSIX threading is more efficient than MPI on `speedy`, but, in contrast, POSIX threading is less efficient than MPI on `lilli`. It can be understood as the difference of hardware architectures. In generally, overhead costs of POSIX threading is less than MPI, so that it made better results on `speedy`. As described at the beginning of this section, the memory bandwidth of `lilli` is higher than `speedy`. The purpose is parallel computing by multi processors, and the hardware architecture of `lilli` is very suitable for MPI. It results as the efficiency of MPI on `lilli`.

The Solver with the L/B and Blocking

To achieve better results, we implemented the blocking solution and benchmarked it. The effect of this solution is somewhat unclear as shown in Table 3.15.

The reduction of time spent for synchronizing can be expected, but the effect of the blocking solution is not remarkable. We also benchmarked the previous non-blocking

Matrix	Sequential	Processors	Total	Dist. Mat.	Subst.
M1	7.476 (sec.)	2	15.170	0.012	15.1
		3	8.176	0.023	8.153
		4	5.174	0.024	5.150
		5	3.925	0.037	3.887
		6	3.168	0.035	3.133
		7	2.655	0.034	2.621
		8	2.276	0.037	2.240
		9	2.042	0.042	2.000
		10	1.820	0.047	1.773
		11	1.623	0.044	1.578
		12	1.500	0.046	1.454
		13	1.350	0.045	1.304
		14	1.282	0.039	1.243
		15	1.209	0.056	1.153
		16	1.114	0.053	1.061
M2	213.333 (sec.)	2	308.252	0.083	308.168
		3	208.022	0.174	207.848
		4	113.865	0.190	113.675
		5	82.461	0.288	82.172
		6	68.863	0.307	68.557
		7	55.924	0.250	55.673
		8	53.131	0.281	52.850
		9	39.921	0.377	39.543
		10	35.667	0.384	35.283
		11	32.736	0.431	32.305
		12	31.908	0.427	31.480
		13	28.065	0.463	27.602
		14	25.787	0.392	25.395
		15	23.290	0.394	22.896
		16	22.339	0.444	21.894

Table 3.8: Benchmarking for the L/B solver by MPI on `lilli`

Matrix	Sequential	Threads	Total	Substitution
M1	4.946 (sec.)	1	5.002	5.001
		2	2.701	2.700
		3	2.285	2.274
		4	1.874	1.863
		5	1.256	1.245
M2	61.263 (sec.)	1	68.344	68.344
		2	43.498	43.486
		3	40.010	39.999
		4	33.248	33.234
		5	25.517	25.516

Table 3.9: Parallel computation solver by threading on `speedy`

version ten times, and got a result shown in Table 3.16. For cases that the blocking size is 6, 7, and 8 look as if more efficient than others, but we can consider these differences are not important.

We also benchmarked the case for fewer threads. The result is shown in Table 3.17.

We also benchmarked the previous non-blocking version ten times, and got a result shown in Table 3.18.

For cases that the blocking size is 6, 7, 8, and 9 look as if more efficient than others. However, we can consider again these differences are not important.

3.6.3 Conclusions

We implemented various solutions as shown in Table 3.19.

We benchmarked the implementations on two computers, `speedy` and `lilli`. We found the load balancing solution is better than implementations without it. The blocking solution did not make speedup. By comparing benchmarking results of MPI with ones of POSIX threading, We found MPI is more efficient than POSIX threading for `lilli`. It is because the hardware architecture of `lilli` is more suitable for MPI than `speedy`.

Matrix	Sequential	Threads	Total	Substitution
M1	7.476 (sec.)	1	16.486	16.485
		2	7.491	7.489
		3	5.008	5.005
		4	3.812	3.808
		5	2.993	2.991
		6	2.507	2.462
		7	2.227	2.225
		8	1.864	1.863
		9	1.688	1.677
		10	1.598	1.596
		11	1.547	1.542
		12	1.684	1.670
		13	1.148	1.146
		14	1.088	1.086
		15	1.044	1.035
		16	1.002	0.981
M2	213.333 (sec.)	1	611.471	611.470
		2	312.144	312.142
		3	214.994	214.990
		4	167.441	167.440
		5	131.630	131.628
		6	112.727	112.726
		7	97.602	97.598
		8	87.053	87.051
		9	77.195	77.195
		10	70.434	70.420
		11	63.426	63.416
		12	58.843	58.827
		13	53.723	53.715
		14	53.365	53.358
		15	53.318	53.305
		16	47.439	47.428

Table 3.10: Parallel computation solver by threading on `lilli`

Matrix	Threads	Thread index	Substitution
M2	11	0	27.446 (sec.)
		1	44.926
		2	27.121
		3	27.573
		4	29.605
		5	27.198
		6	28.228
		7	35.793
		8	28.414
		9	27.689
		10	27.356

Table 3.11: Difference among processors in the case of the POSIX threading solver

Matrix	Sequential	Threads	Total	Substitution
M1	4.946 (sec.)	1	4.811	4.811
		2	2.547	2.543
		3	1.830	1.829
		4	1.485	1.477
		5	1.248	1.244
M2	61.263	1	68.907	68.906
		2	41.124	41.124
		3	35.526	35.525
		4	25.603	25.593
		5	21.927	21.914

Table 3.12: Parallel computation solver with L/B by threading on `speedy`

Matrix	Sequential	Threads	Total	Substitution
M1	7.476 (sec.)	1	14.417	14.416
		2	7.549	7.549
		3	4.977	4.976
		4	3.642	3.642
		5	2.965	2.965
		6	2.504	2.503
		7	2.288	2.287
		8	1.841	1.839
		9	1.672	1.671
		10	1.504	1.496
		11	1.477	1.475
		12	1.452	1.451
		13	1.209	1.206
		14	1.151	1.147
		15	1.003	1.001
		16	0.972	0.969
M2	213.333 (sec.)	1	556.860	556.851
		2	198.379	198.371
		3	193.664	193.662
		4	124.031	124.030
		5	115.891	115.889
		6	99.543	99.541
		7	85.264	85.260
		8	72.334	72.327
		9	71.665	71.656
		10	61.452	61.451
		11	60.085	60.031
		12	55.890	55.886
		13	47.800	47.798
		14	43.191	43.181
		15	43.337	43.307
		16	38.447	38.430

Table 3.13: Parallel computation solver with L/B by threading on `lilli`

	The number of processors/threads	The L/B in MPI	The L/B in threading
`speedy`	5	34.445 (sec.)	21.927
`lilli`	16	22.339	38.447

Table 3.14: Comparing `speedy` with `lilli`

Matrix	Threads	Total without blocking	Blocking size	Total with blocking
M2	16	38.447 (sec.)	1	37.907
			2	35.297
			3	38.489
			4	41.681
			5	39.013
			6	35.819
			7	35.740
			8	35.156
			9	38.231
			10	45.917
			20	39.841
			50	39.559
			100	46.599
			150	51.928
			200	56.005
			250	70.101

Table 3.15: Parallel solver with the L/B and blocking on `lilli`

Matrix	Threads	Total with L/B non-blocking
M2	16	35.873 (sec.)
		34.526
		37.804
		37.193
		36.804
		35.798
		34.893
		35.438
		36.031
		34.184

Table 3.16: Parallel solver with L/B by using 16 threads on `lilli`

Matrix	Threads	Total without blocking	Blocking size	Total with blocking
M2	4	25.603 (sec.)	1	26.209
			2	23.007
			3	24.657
			4	23.107
			5	25.822
			6	22.628
			7	23.099
			8	23.032
			9	23.770
			10	25.747
			20	22.665
			50	26.524
			100	26.829
			150	23.831
			200	23.924
			250	25.103

Table 3.17: Parallel solver with the L/B and blocking on `speedy`

Matrix	Threads	Total with L/B non-blocking
M2	4	24.027 (sec.)
		23.556
		22.381
		27.779
		22.663
		25.816
		22.598
		23.552
		24.257
		23.366

Table 3.18: Parallel solver with L/B by using 4 threads on `speedy`

	None	Load Balancing	Blocking
MPI	✓	✓	
Threading	✓	✓	✓

Table 3.19: Implemented solutions

Chapter 4

Parallelization of the Forward/Backward Substitution Method

4.1 Basic Idea

We describe the parallel f/b substitution method based on message passing, which is a concept for parallel computing. Message passing system enables each process to send and receive data as messages, and is widely used for parallel computing. *MPI*[1] is a specification of message passing for parallel programming, and we implement parallel algorithms by means of MPI in Section 4.3.

We consider the case for the forward substitution method. The case for backward substitution is a straightforward variant. To parallelize the forward substitution method, we consider dependencies of data in the equation. Our problem is to solve following equations:

$$\begin{array}{rcl} \ell_{11}x_1 & = & b_1, \\ \ell_{21}x_1 + \ell_{22}x_2 & = & b_2, \\ \ell_{31}x_1 + \ell_{32}x_2 + \ell_{33}x_3 & = & b_3, \\ & \vdots & \\ \ell_{n1}x_1 + \ell_{n2}x_2 + \ell_{n3}x_3 + \ldots + \ell_{nn}x_n & = & b_n. \end{array}$$

At first, we find x_1 by the sequential forward substitution, and find also all other equations depend on the value of x_1. In the next step, we can substitute the value of x_1 for x_1 which occurs in all other equations in parallel. We do not need to wait these substitutions until finding the value of x_2 because this procedure is independent from the value of x_2. Therefore, we proceed to solve the second equation, and find the value of x_2. We can substitute the value of x_2 in parallel, same to the above case.

If we ignore practical issues for implementation, the parallel forward substitution method is described as follows:

1. To solve the first equation and find x_1.

2. To substitute the value of x_1 for all other occurrence of x_1 in parallel.

[1] http://www.mpi-forum.org/

3. To solve the second equation and find x_2.

4. To substitute the value of x_2 for all other occurrence of x_2 in parallel.

5. To solve the third equation and find x_3.

6. To substitute the value of x_3 for all other occurrence of x_3 in parallel.

7. To iterate the above procedure until finding x_n.

The parallel backward substitution method is derived by similar way to the parallel forward substitution method. We find x_n at first, and proceed to find x_{n-1}, x_{n-2}, ..., and x_1. We give the simultaneous equation to be solved below:

$$\begin{array}{rcl} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n & = & b_1, \\ \vdots & & \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n & = & b_{n-2}, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n & = & b_{n-1}, \\ u_{nn}x_n & = & b_n. \end{array}$$

The parallelized backward substitution methods is described as follows:

1. To solve the nth equation and find x_n.

2. To substitute the value of x_n for all other occurrence of x_n in parallel.

3. To solve the *(n-1)*th equation and find x_{n-1}.

4. To substitute the value of x_{n-1} for all other occurrence of x_{n-1} in parallel.

5. To iterate the above procedure until solving x_1.

4.2 Data Distribution for the Parallel F/B Substitution Method

Several technical problems arise when we implement the parallel f/b substitution method. In this section, we describe technical issues to implement the algorithm in C and MPI, and how to solve them.

We should distribute each row to processors, but the above data structure is not so suitable to take an arbitrary row. So, we transform these data by taking the transposed matrix and reading each column.

$$A^T = \begin{bmatrix} a_{11} & & a_{31} & \\ & a_{22} & a_{32} & a_{42} \\ & & & \\ a_{14} & & a_{44} & \end{bmatrix}$$

We split A^T into columns to assign them to worker processors. An example is depicted in Figure 3.9. This figure explains how a matrix with seven rows is distributed among three processors.

```
SplitMatrix(A, wp)
input A: matrix, wp: number of worker processors
output B: array of matrix

for p=1 to wp do: // wp is the number of worker processors
  partial_matrix[p-1] = prepare_matrix()
  numrow[p-1] = 0 // initialize
done

At = transpose(A) // getting a transposition of A

for rowind=0 to At->m-1 do: // At->m is the number of rows
  // pnum is the processor number which solves rowind
  pnum = proc_number_solves_row(rowind)

  for colind=At->p[rowind-1] to At->p[rowind]-1 do:
    // enter At->x[ind_col] at (num_row[pnum], At->i[ind_col])
    entry(partial_matrix[pnum], num_row[pnum], At->i[ind_col], At->x[ind_col])
  done

  num_row[pnum]++ // to process the next row
done
```

Figure 4.1: SplitMatrix

This procedure can be described by the pseudo-code shown in Figure 4.1. At first, we prepare the array of partial matrices and index for row numbers. `numrow[]` is incremented in the next loop. Next, we get the transposition of the matrix `A`. By this procedure, we can access each row of `A` sequentially. `At->p` means row pointers and `At->i` means column indices. Finally, we construct partial matrices. In the next loop, we process all rows of the matrix `At`. At the beginning, we get the number of processor which solves the row of `rowind`. In the next loop, we enter all entries of the row to `partial_matrix[pnum]`. After finishing to process one row, we increment `num_row[pnum]` and iterate the above procedure until entering the last row of `At` to some partial matrix. As a result, we have the array of partial matrices.

Then, we assign them to different processors. If we have an n-by-n matrix and m worker processors, n mod m processors get $\lfloor n \div m \rfloor + 1$ rows and the other processors get $\lfloor n \div m \rfloor$ rows.

If the size of matrix is five and there are three worker processors, the assignment is as follows:

$$\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ a_{41} & a_{42} & a_{43} & a_{44} & \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

where ░, ╬, ▒ are assigned to different processes.

Finally, we solve the simultaneous linear equation in parallel [10]. We assume there are three processors P1, P2, and P3. First, P1, which received the first row, finds x_1 by using given b_1. After that, P1 broadcasts the value of x_1 to all other processors as Figure 4.2. Next, P2 solves the second row by means of x_1, and finds x_2. After that, P2

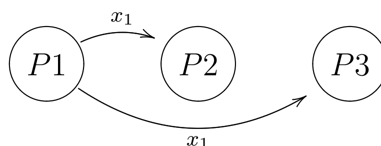


Figure 4.2: Broadcasting the value of x_1

broadcasts the value of x_2 to all other processors as Figure 4.3. As same way, P3 solves

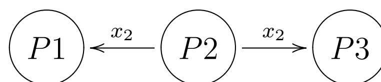


Figure 4.3: Broadcasting the value of x_2

the third row by means of x_2, and finds x_3. P3 broadcasts the value of x_3 to all other processors as shown in Figure 4.4. The above way is easily generalized for an arbitrary size of matrices and an arbitrary number of worker processors.

4.3 Implementation of the Parallel F/B Substitution in MPI

We describe how to implement the parallel f/b substitution method by using MPI.

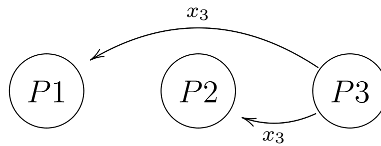


Figure 4.4: Broadcasting the value of x_3

At the beginning, we implement functionalities to solve simultaneous linear equations in a simple way. The way to implement the core of the equation solver by using `MPI_Bcast` is shown by the pseudo-code in Figure 4.5. The procedure in Figure 4.5 is iterated `max` times, which is the number of rows of the original matrix. `rowind` is the index of the row of A^T and `b[rowind]` is solved.

If the `colind_mat` equals to `rowind`, it computes the solution `b[rowind]`. The entry `Ap->x[Ap->p[rowind]]` is on the diagonal, and it is used for dividing. After that, the solution `b[rowind]` is shared among all processors by using `MPI_Bcast`.

The next loop updates `b[colind_mat]`. This updating operation is executed if `colind_mat` is greater than `rowind` so that entries of the array `b` which have already solved are not updated any more.

The last loop is required to synchronize `b` among all processors, because `colmax` depends on partial matrices.

4.4 Implementation of the Parallel F/B Substitution in POSIX Threading

In this section, we implement the parallel f/b substitution by using POSIX threading. Features of threading is as follows:

1. Each thread can share variables simply by means of global variables or pointers of C.
2. It is not straight forward to port programs with threading to programs with MPI.

Instead of distributing partial matrices among processors by invoking MPI functions, We pass pointers to each partial matrices to worker threads. Furthermore, we prepare the array `b` for the solution, and `b` is shared among worker threads by using a pointer to `b`. It reduces the time for communication of MPI. We introduce flags to avoid race condition in critical sections.

We invoke `pthread_create()` to create new threads. This function requires the function assigned to threads and its argument. In our implementation, following data is given to each thread.

1. An index for the thread.
2. A size of the original matrix.
3. A pointer to the assigned partial matrix.
4. A pointer to the array `b` for the solution.

ParallelForwardSubstitution(Ap, rhs, max, wp, myid)

input	Ap: the partial matrix
	rhs: vector,
	max: the number of row of the original matrix,
	wp: the number of worker processors,
	myid: the index of this processor,
output	b: solution.

```
b = rhs

for rowind = 0 to Ap->n do:
  proc_solve = proc_number_solves_row(rowind)
  colind_mat = get_matrix_index(myid, Ap->i[Ap->p[rowind]]);
  // colind_mat is a column index in the original matrix

  if(colind_mat = rowind) do: // on the diagonal
    b[rowind] /= Ap->x[Ap->p[rowind]]
  done

  MPI_Bcast(&b[colind_mat], 1, MPI_DOUBLE, proc_solve, MPI_COMM_WORLD);
  // all processors shared b[colind_mat]

  for rowind_pm = Ap->p[rowind] to Ap->p[rowind+1]-1 do
    colind_mat = get_matrix_index(myrank, Ap->i[rowind_pm])

    if colind_mat > rowind do: // the matrix is always lower triangular
      b[colind_mat] -= Ap->x[rowind_pm]*b[rowind] // updating
    done
  done

  for rowind = Ap->n to max do:
    proc_solve = proc_number_solves_row(rowind)
    MPI_Bcast(&b[rowind], 1, MPI_DOUBLE, proc_solve, MPI_COMM_WORLD)
  done
done
```

Figure 4.5: Parallel forward substitution

```
void* thread_task(void* param) {
  int index;
  int An;
  cs* Ap;
  double* b;
  char** str;

  pthread_setspecific(key, ((char**)param));
    // key is a global variable
  str = (char**)pthread_getspecific(key);

  index = atoi(str[0]); // index for each thread
  An = atoi(str[1]); // the size of the original matrix
  Ap = (cs*)str[2]; // the pointer to the assigned partial matrix
  b = (double*)str[3]; // the pointer to the array for the solution

  forward_subst_threading(Ap, b, An, index);

  return;
}
```

Figure 4.6: The function for threads

The function for threads is described in Figure 4.6. The function `forward_subst_threading` is basically same to the function for MPI, but it requires a couple of modifications to be suitable for threading. We do not need to invoke functions for communication, but need to avoid race conditions.

4.5 Implementation of the Parallel F/B Substitution in MPI with Blocking

In Section 4.3, we split the given matrix every one row. By splitting every $n$rows, the time to invoke MPI function might be reduced. We call this way to split the matrix *blocking*. We did not implement this solution because speedup could not be expected. In this section, we just give an idea for the implementation.

As an example, we consider the following matrix.

$$\begin{bmatrix} 1 & & & & \\ 2 & 6 & & & \\ 3 & 7 & 10 & & \\ 4 & 8 & 11 & 13 & \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

We assume two worker processes and we split every two rows. Then, we have fol-

lowing two partial matrices.

$$\begin{bmatrix} 1 & & & & \\ 2 & 6 & & & \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}, \begin{bmatrix} 3 & 7 & 10 & \\ 4 & 8 & 11 & 13 \end{bmatrix}$$

The modification of the function SplitMatrix is done by adding the number n as an argument, and extending the function `proc_number_solves_row()` to take the number n as an additional argument. Others are same to Figure 4.1.

4.6 Benchmarking of the Implementation

In this section, we describe the performance of our implementation. We benchmarked the sequential solver, the parallel solver, and the concurrent solver.

The specification of the computer we used for validating the implementation is listed below:

1. Machine Name: `speedy`
   (a) CPU: Four 2.33 GHz dual core Intel Xeon processors
   (b) memory: 16.433.856 kB
2. Machine Name: `lilli`
   (a) CPU: 1.59 GHz Intel processors
   (b) memory: 1.017.161.392 kB

We used three different matrices shown in Figure 4.1 for benchmarking.

Matrix Name	Size	Nonzero Entries
M1	19.138-by-19.138	314.243
M2	104.442-by-104.442	2.589.787
M3	1.585.478-by-1.585.478	4.623.152
M4	986.703-by-986.703	36.326.514

Table 4.1: Matrices

4.6.1 The Solver by using MPI

We timed the implementation of the parallel solver with following parameters:

1. The number of processors.
2. The number of rows as a unit to split the matrix.

The result of benchmarking is shown in Table 4.2 and Table 4.3. In this benchmarking, we timed the core of solving equations. Time to read matrices and output solutions are excluded.

Matrix	Sequential (sec.)	processors	Total (sec.)	Splitting	Distributing	
M2	0.0146	1	0.247	0.182	9.5×10^{-7}	0.0643
		2	0.460	0.184	0.0293	0.246
		3	1.464	0.210	0.0490	1.204
		4	1.443	0.220	0.0521	1.171
		5	2.131	0.310	0.0850	1.734
M3	0.0595	1	1.289	0.434	9.5×10^{-7}	0.840
		2	6.734	0.466	0.0824	6.170
		3	17.249	0.513	0.116	1.660
		4	19.107	0.565	0.134	18.388
		5	21.946	0.558	0.146	21.222
M4	0.189	1	3.797	3.131	9.5×10^{-7}	0.657
		2	7.629	3.344	0.525	3.729
		3	11.674	3.258	0.6744	7.706
		4	14.949	3.268	0.687	10.955
		5	18.909	3.155	0.933	14.778

Table 4.2: Benchmark for solving equations by using MPI on `speedy`

Matrix	Sequential (sec.)	processors	Total (sec.)	Splitting	Distributing	Substitution
M1	0.0021	1	0.0541	0.0306	7.497×10^{-7}	0.0233
		2	0.0994	0.0254	0.0037	0.0704
		3	0.1511	0.0231	0.0050	0.1229
		4	0.1273	0.0248	0.0049	0.0976
		5	0.2354	0.0271	0.0091	0.1992
		6	0.1913	0.0261	0.0099	0.1554
		7	0.1846	0.0281	0.0083	0.1490
		8	0.1829	0.0265	0.0138	0.1477
M2	0.0561	1	0.390	0.222	5.99×10^{-7}	0.168
		2	0.642	0.274	0.0234	0.343
		3	0.747	0.220	0.0310	0.496
		4	0.877	0.249	0.0346	0.597
		5	0.921	0.210	0.0361	0.674
		6	1.045	0.214	0.0398	0.791
		7	0.973	0.241	0.0389	0.692
		8	1.003	0.252	0.0425	0.708
M3	0.252	1	2.084	0.602	7.5×10^{-7}	1.475
		2	4.588	0.593	0.0353	3.952
		3	9.073	0.624	0.0791	8.358
		4	9.049	0.628	0.0703	8.343
		5	11.752	0.685	0.0856	10.967
		6	11.797	0.652	0.104	11.028
		7	11.567	0.700	0.162	10.700
		8	11.837	0.739	0.187	10.902
M4	0.766	1	6.358	4.403	6.50×10^{-7}	1.951
		2	6.686	3.505	0.310	3.043
		3	8.500	3.449	0.420	4.628
		4	10.495	3.508	0.549	6.436
		5	11.285	3.046	0.531	7.705
		6	11.188	3.036	0.584	7.564
		7	12.276	3.591	0.743	7.938
		8	11.616	3.387	0.652	7.572

Table 4.3: Benchmark for solving equations by MPI on `lilli`

4.6.2 The Solver by POSIX Threading

We timed the implementation of the concurrent solver with following parameters:

- The number of threads.

The result of benchmarking is shown in Table 4.4. Total time means time to split the given matrix and to solve it. Time to read matrices and output results are excluded.

Matrix	Sequential (sec.)	Threads	Total (sec.)	Split	Substitution
M1	0.0021	1	0.028	0.020	0.007
		2	0.113	0.021	0.092
		3	0.168	0.020	0.148
		4	0.175	0.021	0.153
M2	0.01460	1	0.233	0.185	0.047
		2	2.353	0.188	2.164
		3	2.839	0.192	2.646
		4	2.832	0.188	2.643
M3	0.252	1	0.987	0.412	0.560
		2	31.336	0.445	20.876
		3	23.985	0.471	23.499
		4	42.026	0.482	41.529
M4	0.766	1	3.778	3.268	0.501
		2	22.243	3.163	19.071
		3	27.476	2.986	24.481
		4	28.791	2.992	25.786

Table 4.4: Benchmark for solving equations with threading on `speedy`

4.6.3 Conclusions

We have implemented the parallelized forward substitution by means of MPI and POSIX threading as shown in Table 4.6. As a result, we did not achieve speedup at all. It is because of the granularity of the problem. Split problems are too small to apply parallel computing.

Matrix	Sequential (sec.)	Threads	Total (sec.)	Split	Substitution
M1	0.0021	1	0.061	0.024	0.032
		2	0.967	0.022	0.943
		3	1.617	0.022	1.595
		4	1.965	0.023	1.938
M2	0.056	1	0.360	0.222	0.137
		2	5.271	0.198	5.070
		3	5.355	0.216	5.138
		4	7.931	0.261	7.656
M3	0.0594	1	1.534	0.513	1.007
		2	71.445	0.500	70.931
		3	94.065	0.547	93.502
		4	169.824	0.546	169.266
M4	0.1888	1	5.756	3.034	2.713
		2	57.754	3.103	54.643
		3	64.158	2.979	61.170
		4	92.125	2.977	89.140

Table 4.5: Benchmark for solving equations with threading on `lilli`

	None	Blocking
MPI	✓	
Threading	✓	

Table 4.6: Implemented solutions

Chapter 5

Parallel F/B Substitution for Sparse Matrices

In this chapter, we give a parallel algorithm of the f/b substitution for sparse matrices. Since this algorithm has not been implemented yet, implementing of the algorithm is future work.

The procedure to solve sparse linear equations is as follows:

1. Generating the dependency graph G of the given matrix.
2. Generating task lists for processors.
3. Solving equations in parallel.

Dependency graphs are generated as described in Section 2.2.2. We describe how to generate task lists without considering the number of processors. After that, we consider how to join task list with regard to the number of available processors.

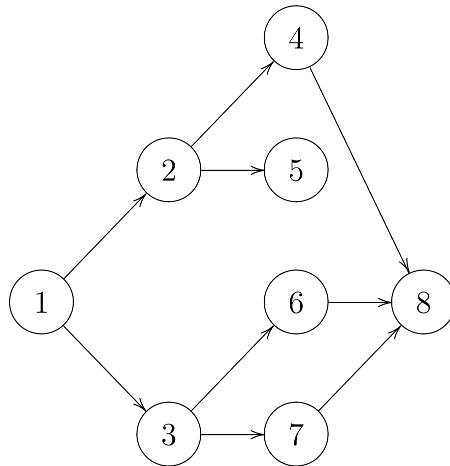
5.1 Generating Task Lists

We represent graphs by adjacency-list representation because the matrix is sparse. Short cut paths should be eliminated. For example, We have a graph of three vertex, 1, 2, and 3, and there are paths $1 \to 2 \to 3$, and $1 \to 3$, the latter one is a short cut. In adjacency-list representation, it comes as multiple elements in one array.

$$Adj[1] = \{1, 2, 3, 3\}$$

Next, we generate task lists for processors. We assume the following dependency

graph.



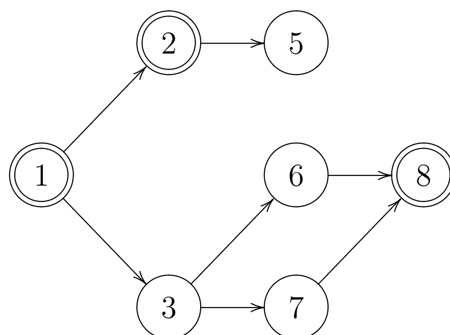
These dependency relations show which part can be computed in parallel, and which part cannot be parallelized. Next, we assign tasks to processors. The procedure is as follows:

1. Finding the longest path.
2. Assigning them to some processor.
3. Marking assigned nodes.
4. Eliminating unnecessary paths and nodes.
5. Iterating the above procedure until all nodes are eliminated.

We assume the number of processor is four. We denote processors by P_i. We took the longest path, $1 \to 2 \to 4 \to 8$, and assign the path to P_1. Then, task list of P_1 is:

$$task[1] = \{solve(1), send(1), solve(2), send(2), solve(4), receive(6), receive(7), solve(8)\}$$

Now, 4 can be eliminated because the degree of 4 is zero if the path $2 \to 4 \to 8$ is eliminated. As a result, we have the following graph:

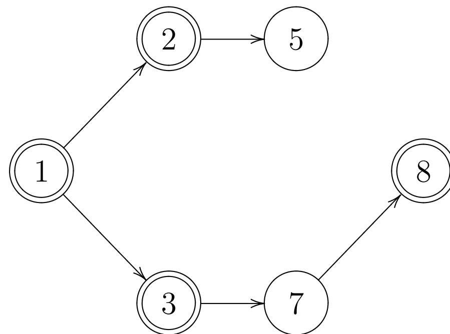


Double circles mean the node have already been assigned.

We iterate this procedure. We take the path $3 \to 6 \to 8$ and assign it to P_2. Then, the task list of P_2 is:

$$task[2] = \{receive(1), solve(3), send(3), solve(6), send(6)\}$$

Now, 6 can be eliminated and we have the following graph:



The longest path is $1 \rightarrow 3 \rightarrow 7 \rightarrow 8$ and 7 is not solved, so we assign this path to P_3. Because 7 depends on 3, the task list is:

$$task[3] = \{receive(3), solve(7), send(7)\}$$

At the last, 5 is solved by P_4, and the task list is:

$$task[4] = \{receive(2), solve(5)\}$$

5.2 Joining Task Lists

The basic idea for parallelization is shown as above. Next, we need to consider the number of processors and join task lists. To get an optimal joined task list, we need to consider how to increase the number of communication.

If two processor is available, 5 should be assigned to P_1 and 7 should be assigned to P_2 because it reduces *send/receive* operations. In this case, we take a concatenation of $task[1]$ and $task[4]$, and delete unnecessary *send/receive*. Then, we have the task list:

$$task[1] = \{solve(1), send(1), solve(2), solve(4), receive(6), receive(7), solve(8), solve(5)\}$$

We take a concatenation of $task[2]$ and $task[3]$ in the same way:

$$task[2] = \{receive(1), solve(3), solve(6), send(6), solve(7), send(7)\}$$

More generally, we count the number of *send/receive* operation in each task list. We recursively join task list by joining the task list with the least number of *send/receive* operation.

In the above example, each list has following *send/receive* operations list.

List No.	Operations
1	send(1), send(2), recv(6), recv(7)
2	recv(1), send(3), send(6)
3	recv(3), send(7)
4	recv(2)

We process the list 4 at first. It receives x_2 from the list 1, so joining two of them decreases *send/receive* operations. As a result of the first joining, we have the following table:

List No.	Operations
1	send(1), recv(6), recv(7)
2	recv(1), send(3), send(6)
3	recv(3), send(7)

Now, we have two options: joining the list 3 with 1, or joining the list 3 with 2. If one of them had a priority such that two operation of the list 3 can be removed, we don't need to think anymore. However, there is no such a choice in this example. Therefore, we join the list 3 with the list which has less computation tasks. The complete task list of list 1 and 2 is as follows:

List No.	Tasks
1	solve(1), send(1), solve(2), solve(4), recv(6), recv(7), solve(8), solve(5)
2	recv(1), solve(3), send(3), solve(6), send(6)
3	recv(3), solve(7), send(7)

In this case, the list 2 has fewer tasks and the list 3 should be joined with the list 2. Then, we have a following task list:

List No.	Tasks
1	solve(1), send(1), solve(2), solve(4), recv(6), recv(7), solve(8), solve(5)
2	recv(1), solve(3), solve(6), send(6), solve(7), send(7)

This procedure is invoked recursively, and it joins lists as requires. As we see the above example, we can expect balanced task lists as a results. However, we need further investigation. For example, we may get more balanced task lists by comparing result of two or three joining steps every time. Obviously, we only considered one joining step in the procedure.

We consider this problem can be optimized by a greedy algorithm. The above discussion is not formal and to be sophisticated by future work.

Chapter 6

Conclusion

We have developed various ways of the parallel forward/backward substitution method.

1. Parallel computation of multiple forward/backward substitutions,
2. Parallelization of the forward/backward substitution method,
3. Parallel forward/backward substitution method for sparse matrices.

We have implemented 1 and 2 in various programming models, MPI and POSIX threading, and have benchmarked them on various hardware architectures, the shared memory architecture (`speedy`) and the massively parallel virtual shared memory architecture (`lilli`).

We achieved good speedup with the first solution, especially by means of the Load Balancing solution. We found MPI is better solution than POSIX threading for `lilli`. In contrast, we found POSIX threading is more suitable for `speedy` than MPI.

As we have benchmarked the second solution, there is no speedup by this solution. It is because communication costs are much higher than benefits of the parallelization.

We designed the third sokution, and implementing of this solution is future work.

Appendix A

Source Code

In this appendix, we give source code we developed. We developed the software based on CSparse which we introduced in Section 2.2.4 [3]. Source code of CSparse is required to compile our implementation. We explain the organization of our source code in Section A.1. In Section A.2, we give lists of the source code.

A.1 Organization of Source Code

Each implementation is organized as shown in Table A.1.

Name	Source Code
Sequential Solver	seq_solver.c solver_utils.c seq_subst.c
Sequential Multiple Substitution	seq_psolver.c solver_utils.c seq_subst.c
Multiple Substitutions in MPI	pt_solver.c solver_utils.c solver_matrix.c mpi_subst.c
Multiple Substitutions in Threading	pt_psolver.c solver_utils.c solver_matrix.c pt_subst.c
Multiple Substitutions with L/B in MPI	mpi_plbsolver.c seq_subst.c solver_utils.c solver_matrix.c solver_mpicomm.c
Multiple Substitutions with L/B in Threading	pt_plbsolver.c solver_utils.c solver_matrix.c pt_plbsubst.c seq_subst.c
Multiple Substitutions with L/B and Blocking in Threading	pt_plbsolver_blocking.c solver_utils.c pt_plbsubst_blocking.c seq_subst.c
Parallelized Substitution in MPI	mpi_solver.c solver_utils.c solver_matrix.c solver_mpicomm.c mpi_subst.c
Parallelized Substitution in Threading	pt_solver.c solver_utils.c solver_matrix.c pt_subst.c

Table A.1: Organization of source code

We used MPICH [8] and GNU gcc/Intel icc to compile these code. They can be compiled with CSparse as follows:

```
mpicc mpi_psolver.c solver_utils.c solver_mpicomm.c solver_matrix.c \
cs_entry.c cs_compress.c cs_util.c cs_malloc.c cs_cumsum.c cs_transpose.c
```

```
gcc -pthread -lrt pt_psolver.c solver_utils.c solver_matrix.c pt_psubst.c \
cs_compress.c cs_entry.c cs_util.c cs_malloc.c cs_transpose.c cs_cumsum.c
```

A.2 Source Code

A.2.1 Header File

solver.h

```
#define TRUE -1
#define FALSE 0

int read_matrix(char* filename, cs** matrix);
int read_rhs_array(char* filename, double*** rhs, int An);
void output_array_int(char* filename, int* int_array, int num);
void output_array_double(char* filename, double* double_array, int num);
void output_solution(char* file, double** sol, int num_vec, int length);
cs** split_matrix(cs* A, int unit);
cs** split_matrix_nonblocking(cs* A);
cs* distribute_matrix_bcast(cs* A);
void distribute_matrix_send(cs* A, cs** pmatrix, int unit);
void distribute_matrix_recv(cs** Ap, int unit);
cs* distribute_matrix_nonblocking_threading(cs** pmatrix, int index);
int number_of_line(int matrix_size, int proc, int unit);
int number_of_row(int matrix_size, int proc, int unit);
int number_of_rhs_assigned(int num_rhs, int myid);
void gen_rhs(int n, double* v);
int proc_number_solves_row(int ind_row, int unit);
int get_matrix_index(int proc, int ind_row, int unit);
int get_split_matrix_index(int proc, int mind, int unit);
cs** distribute_rhs_send(cs** A, int num_vec);
double** distribute_rhs_array_send(double** array, int An, int num_vec);
cs** distribute_rhs_recv(int num_vec);
double** distribute_rhs_array_recv(int An, int num_vec);
cs* distribute_matrix_bcast(cs* A);
double* forward_subst_sequential(cs* A, double* b, int An);
double* forward_subst_sequential_threading(cs* A, double* b, int An);
double* forward_subst(cs* Ap, double* b, int An, int unit);
double* forward_subst_without_unit(cs* Ap, double* b, int An, int unit);
double* forward_subst_nonblocking(cs* Ap, double* b, int An);
void* thread_task(void* param);
void* thread_task_forward_pt_solver(void* param);
void* thread_task_forward_pt_psolver(void* param);
void* thread_task_forward_pt_plbsolver(void* param);
void* thread_task_forward_pt_plbsolver_blocking(void* param);
int get_next_rhs();
int get_next_rhs_blocking();
```

A.2.2 Common Routines

solver_utils.c

```
#include <stdio.h>
#include <stdlib.h>

#include "cs.h"
```

```
#include "solver.h"

extern int mysize;
extern int myrank;

int read_matrix(char* filename, cs** matrix) {
  FILE* fp;
  char line[256];
  int ny, nx, nnz;
  int i;

  fp = fopen(filename, "r");
  if(fp == NULL) {
    printf("cannot open matrix file %s\n", filename);
    exit (1);
  }

  fgets(line, 256, fp);
  fgets(line, 256, fp);

  sscanf(line, "%d %d %d", &ny, &nx, &nnz);
  *matrix = cs_spalloc(ny, nx, nnz, TRUE, TRUE);

  for(i=0; i<nnz; i++) {
    int i, j;
    double val;

    fscanf(fp, "%d %d %lg", &i, &j, &val);
    cs_entry(*matrix, i-1, j-1, val);
  }

  fclose(fp);

  return nnz;
}

int read_rhs_array(char* filename, double*** rhs, int An) {
  FILE* fp;
  char line[256];
  double** vector;
  int row;
  int i;
  int num_vectors;
  double value;

  fp = fopen(filename, "r");
  if(fp == NULL) {
    printf("cannot open right hand side file %s\n", filename);
    exit (-1);
  }

  // count the number of vectors in the file.
  num_vectors=0;
  while(fgets(line, 256, fp) != NULL) {
    if(sscanf(line, "%d %lg\n", &row, &value) == 0) {
      num_vectors++;
    }
```

```
  }
  rewind(fp);

  // read data in the file
  vector = (double**)malloc(num_vectors*sizeof(double*));

  fgets(line, 256, fp); // read "*** LC:    1"
  for(i=0; i<num_vectors; i++) {
    vector[i] = (double*)calloc(An, sizeof(double));
    fgets(line, 256, fp); // read line
    while(sscanf(line, "%d %lg\n", &row, &value) == 2) {
      vector[i][row] = value;
      if(fgets(line, 256, fp) == NULL) {
        break;
      }
    }
  }
  *rhs = vector;

  return num_vectors;
}

void output_array_int(char* filename, int* int_array, int num) {
  FILE* fp;
  int i;

  fp = fopen(filename, "w");
  if(fp == NULL) {
    printf("cannot open %s.\n", filename);
    exit(1);
  }

  for(i=0; i<num; i++) {
    fprintf(fp, "%d\n", int_array[i]);
  }

  fclose(fp);
}

void output_array_double(char* filename, double* double_array, int num) {
  FILE* fp;
  int i;

  fp = fopen(filename, "w");
  if(fp == NULL) {
    printf("cannot open %s.\n", filename);
    exit(1);
  }

  for(i=0; i<num; i++) {
    fprintf(fp, "%e\n", double_array[i]);
  }

  fclose(fp);
}

void gen_rhs(int n, double* v) {
```

```
  int i;

  for (i=0; i<n; i++) {
    v[i] = (double)i / 1000.0;
  }
}

void output_solution(char* file, double** sol, int vec, int length) {
  FILE *fp;
  int i, j;

  fp = fopen(file, "w");
  if (fp == NULL) {
    printf("cannot open file %s\n", file);
    return;
  }

  for(i=0; i<vec; i++) {
    fprintf(fp, "*** solution %d\n", i);
    for(j=0; j<length; j++) {
      if (sol[i][j] != 0) {
        fprintf(fp, "%16d %e\n", j, sol[i][j]);
      }
    }
  }

  fclose(fp);

  return;
}
```

solver_matrix.c

```
#include "cs.h"
#include "solver.h"

extern int mysize;

cs** split_matrix(cs* A, int unit) {
  int proc;
  int ind_row;
  int ind_col;
  cs* At;

  int* num_row;
  cs** tmp_part_matrix;
  cs** ret;

  At = cs_transpose(A, TRUE);
  tmp_part_matrix = (cs**)malloc(mysize*sizeof(cs*));
  ret = (cs**)malloc(mysize*sizeof(cs*));
  num_row = (int*)calloc(mysize, sizeof(int));

  for(proc=0; proc<mysize; proc++) {
    tmp_part_matrix[proc] = cs_spalloc(1, 1, 1, TRUE, TRUE);
  }
```

```
  for(ind_row=0; ind_row<A->n; ind_row++) {
    int pnum = proc_number_solves_row(ind_row, unit);
    for(ind_col=At->p[ind_row]; ind_col<At->p[ind_row+1]; ind_col++) {
      cs_entry(tmp_part_matrix[pnum], num_row[pnum], At->i[ind_col], At->x[ind_col]);
    }
    num_row[pnum]++;
  }

  for(proc=0; proc<mysize; proc++) {
    ret[proc] = cs_compress(tmp_part_matrix[proc]);
    cs_spfree(tmp_part_matrix[proc]);
  }
  free(tmp_part_matrix);
  cs_spfree(At);

  return ret;
}

int proc_number_solves_row(int ind_row, int unit) {
  int mod_line;
  int mod_unit;
  int procno;

  mod_line = ind_row % (unit*mysize);
  mod_unit = mod_line % unit;

  procno = mod_line / unit;

  return procno;
}

int get_matrix_index(int proc, int ind_row, int unit) {
  int line;
  int line_head;
  int proc_head;
  int offset;

  line = ind_row / unit;
  line_head = line*unit*mysize;
  proc_head = proc*unit;
  offset = ind_row % unit;

  return line_head+proc_head+offset;
}

int number_of_row(int matrix_size, int proc, int unit) {
  int num_line;
  int unit_size;
  int partial_unit;
  int ret;

  unit_size = matrix_size / unit;
  partial_unit = matrix_size % unit;
  num_line = number_of_line(matrix_size, proc, unit);

  if(partial_unit == 0) {
    ret = num_line * unit;
```

```
  } else {
    if(unit_size%mysize == proc) {
      ret = (num_line-1)*unit + partial_unit;
    } else {
      ret = num_line * unit;
    }
  }

  return ret;
}

int number_of_line(int matrix_size, int proc, int unit) {
  int div;
  int mod;
  int ret;

  div = matrix_size / (mysize*unit);
  mod = matrix_size % (mysize*unit);

  if(mod == 0) {
    ret = div;
  } else {
    int div_lastrow;
    int mod_lastrow;

    div_lastrow = mod / unit;
    mod_lastrow = mod % unit;

    if(mod_lastrow != 0) {
      div_lastrow += 1;
    }

    if(proc < div_lastrow) {
      ret = div+1;
    } else {
      ret = div;
    }
  }

  return ret;
}

int number_of_rhs_assigned(int num_rhs, int myid) {
  int div;
  int mod;

  div = num_rhs / mysize;
  mod = num_rhs % mysize;
  if(mod != 0) {
    if(myid < mod) {
      return div+1;
    }
  }
  return div;
}
```

solver_mpicomm.c

```
#include "cs.h"
#include "mpi.h"
#include "solver.h"

extern int mysize;
extern int myrank;

void distribute_matrix_send(cs* A, cs** pmatrix, int unit) {
  int proc;

  for(proc=1; proc<mysize; proc++) {
    MPI_Send(&pmatrix[proc]->nzmax, 1, MPI_INT, proc, 0, MPI_COMM_WORLD);
    MPI_Send(&pmatrix[proc]->m, 1, MPI_INT, proc, 1, MPI_COMM_WORLD);
    MPI_Send(&pmatrix[proc]->n, 1, MPI_INT, proc, 2, MPI_COMM_WORLD);

    MPI_Send(pmatrix[proc]->p, pmatrix[proc]->n+1, MPI_INT, proc, 3, MPI_COMM_WORLD);

    MPI_Send(pmatrix[proc]->i, pmatrix[proc]->nzmax, MPI_INT, proc, 4, MPI_COMM_WORLD);
    MPI_Send(pmatrix[proc]->x, pmatrix[proc]->nzmax, MPI_DOUBLE, proc, 5, MPI_COMM_WORLD);

    MPI_Send(&pmatrix[proc]->nz, 1, MPI_INT, proc, 6, MPI_COMM_WORLD);
  }

  return;
}

void distribute_matrix_recv(cs** Ap, int unit) {
  cs* tmpmat;
  MPI_Status stat;
  int nzmax;
  int m;
  int n;

  MPI_Recv(&nzmax, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &stat);
  MPI_Recv(&m, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &stat);
  MPI_Recv(&n, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, &stat);
  tmpmat = cs_spalloc(m, n, nzmax, TRUE, FALSE);
  MPI_Recv(tmpmat->p, n+1, MPI_INT, 0, 3, MPI_COMM_WORLD, &stat);
  MPI_Recv(tmpmat->i, nzmax, MPI_INT, 0, 4, MPI_COMM_WORLD, &stat);
  MPI_Recv(tmpmat->x, nzmax, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD, &stat);
  MPI_Recv(&tmpmat->nz, 1, MPI_INT, 0, 6, MPI_COMM_WORLD, &stat);

  tmpmat->nzmax = nzmax;
  tmpmat->m = m;
  tmpmat->n = n;
  *Ap = tmpmat;

  return;
}

double** distribute_rhs_array_send(double** array, int An, int num_vec) {
  double** ret;
  int i;
  int num_assign;
  int index;
```

```
  index = 0;
  num_assign = number_of_rhs_assigned(num_vec, myrank);
  ret = (double**)malloc(num_assign*sizeof(double*));

  for(i=0; i<num_vec; i++) {
    int proc;

    proc = i % mysize;
    if(proc != myrank) {
      MPI_Send(array[i], An, MPI_DOUBLE, proc, 0, MPI_COMM_WORLD);
    } else {
      ret[index] = array[i];
      index++;
    }
  }

  return ret;
}

double** distribute_rhs_array_recv(int An, int num_vec) {
  double** array;
  int num_assign;
  int i;

  num_assign = number_of_rhs_assigned(num_vec, myrank);
  array = (double**)malloc(num_assign*sizeof(double**));

  for(i=0; i<num_assign; i++) {
    double* tmp;
    MPI_Status stat;

    tmp = (double*)malloc(An*sizeof(double));
    MPI_Recv(tmp, An, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);

    array[i] = tmp;
  }

  return array;
}

cs* distribute_matrix_bcast(cs* A) {
  int nzmax, m, n, nz;
  cs* B;

  if(myrank == 0) {
    nzmax = A->nzmax;
    m = A->m;
    n = A->n;
    nz = A->nz;
  }
  MPI_Bcast(&nzmax, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&nz, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(myrank == 0) {
    MPI_Bcast(A->p, n+1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    MPI_Bcast(A->i, nzmax, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(A->x, nzmax, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return A;
  } else {
    B = cs_calloc(1, sizeof(cs));
    B->m = m;
    B->n = n;
    B->nzmax = nzmax;
    B->nz = nz;

    B->p = (int*)malloc((n+1)*sizeof(int));
    MPI_Bcast(B->p, n+1, MPI_INT, 0, MPI_COMM_WORLD);
    B->i = (int*)malloc(nzmax*sizeof(int));
    MPI_Bcast(B->i, nzmax, MPI_INT, 0, MPI_COMM_WORLD);
    B->x = (double*)malloc(nzmax*sizeof(double));
    MPI_Bcast(B->x, nzmax, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return B;
  }
}
```

A.2.3 Sequential Solver

seq_solver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

#define FALSE 0
#define TRUE -1

double start_time;
int myrank;
int mysize;

int main(int argc, char* argv[]) {
  cs* A;
  double* b;
  double* answer;
  struct timespec tp;
  long time1, time2, time3;

  // begin: preparation
  if(argc < 2) {
    if(myrank == 0) {
      printf("usage: ./seq_solver matrix.mtx\n");
    }
    return 0;
  }

  read_matrix(argv[1], &A);
```

```
  A = cs_compress(A);
  // end: preparation

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  // preparing right hand side
  b = (double*)malloc(A->n*sizeof(double));
  gen_rhs(A->n, b);

  // begin: core of the solver

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  answer = forward_subst_sequential(A, b, A->n);

  clock_gettime(CLOCK_REALTIME, &tp);
  time3 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  printf("total: %ld msec.\n", time3-time1);
  printf("subst: %ld msec.\n", time2-time1);

  output_array_double("answer.log", answer, A->n);
  free(b);

  return 0;
}
```

seq_subst.c

```
#include "cs.h"
#include "solver.h"

double* forward_subst_sequential(cs* A, double* b, int An) {
  int rowind;

  for(rowind=0; rowind<An; rowind++) {
    int rowind_pm;
    int colind;

    colind = A->i[A->p[rowind]];
    if(colind == rowind) {
      b[colind] /= A->x[A->p[rowind]];
    }

    for(rowind_pm=A->p[rowind]; rowind_pm<A->p[rowind+1]; rowind_pm++) {
      colind = A->i[rowind_pm];

      if(colind > rowind) {
        b[colind] -= A->x[rowind_pm]*b[rowind];
      }
    }
  }

  return b;
}
```

A.2.4 Sequential Multiple Substitutions

seq_psolver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

double start_time;

int main(int argc, char* argv[]) {
  int unit;

  cs* A;

  double** rhs;

  //pthread_init();

  // begin: preparation
  if(argc < 2) {
    printf("usage:\n ./seq_psolver matrix.mtx rhs.mtx\n");
    return 0;
  }

  int num_vec;

  read_matrix(argv[1], &A);
  A = cs_compress(A);
  num_vec = read_rhs_array(argv[2], &rhs, A->m);

  int i;
  double** solution;
  struct timespec tp;
  long time1, time2, tt;

  clock_gettime(CLOCK_REALTIME, &tp);
  tt = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  solution = (double**)malloc(num_vec*sizeof(double*));

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  for(i=0; i<num_vec; i++) {
    solution[i] = forward_subst_sequential(A, rhs[i], A->m);
  }

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("subst: %ld msec.\n", time2-time1);
  printf("total: %ld msec.\n", time2-tt);

  output_solution("seq_psolver_solution.log", solution, num_vec, A->m);
```

```
  return 0;
}
```

A.2.5 Parallel Multiple Substitutions in MPI

mpi_psolver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "cs.h"
#include "mpi.h"
#include "solver.h"

double start_time;
int myrank;
int mysize;
pthread_key_t key;

int main(int argc, char* argv[]) {
  cs* A;

  double** rhs;
  double** rhs_split;

  MPI_Init(&argc, &argv);
  //pthread_init();

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &mysize);

  // begin: preparation
  if(argc < 3) {
    if(myrank == 0) {
      printf("usage:\n mpirun -np n solver matrix.mtx rhs.dat\n");
    }
    MPI_Finalize();
    return 0;
  }

  double t1;
  t1 = MPI_Wtime();
  int num_vec;

  if (myrank == 0) {
    double td, tm;

    read_matrix(argv[1], &A);
    A = cs_compress(A);

    tm = MPI_Wtime();
    A = distribute_matrix_bcast(A);
    printf("(%d) matrix distribution: %e\n", myrank, MPI_Wtime()-tm);
    num_vec = read_rhs_array(argv[2], &rhs, A->m);
```

```
  td = MPI_Wtime();
  MPI_Bcast(&num_vec, 1, MPI_INT, 0, MPI_COMM_WORLD);
  rhs_split = distribute_rhs_array_send(rhs, A->m, num_vec);
  printf("(%d) rhs distribute: %e\n", myrank, MPI_Wtime()-td);
} else {
  double td1, td2, tm;

  tm = MPI_Wtime();
  A = distribute_matrix_bcast(A);
  printf("(%d) matrix distribution: %e\n", myrank, MPI_Wtime()-tm);

  t1 = MPI_Wtime();
  td1 = MPI_Wtime();
  MPI_Bcast(&num_vec, 1, MPI_INT, 0, MPI_COMM_WORLD);
  rhs_split = distribute_rhs_array_recv(A->m, num_vec);
  td2 = MPI_Wtime();
}

double** solution;
double** answer = NULL;
double ts1, ts2;
int num_assign;
int i;

num_assign = number_of_rhs_assigned(num_vec, myrank);
solution = (double**)malloc(num_assign*sizeof(double*));

ts1 = MPI_Wtime();
for(i=0; i<num_assign; i++) {
  solution[i] = forward_subst_sequential(A, rhs_split[i], A->m);
}
ts2 = MPI_Wtime();
printf("(%d) subst %e\n", myrank, ts2-ts1);

// collecting solutions
if(myrank == 0) {
  int ind_arr;

  printf("(%d) collecting solutions\n", myrank);
  answer = (double**)malloc(num_vec*sizeof(double*));
  for(ind_arr=0; ind_arr<num_vec; ind_arr++) {
    int proc;
    MPI_Status stat;

    proc = ind_arr%mysize;
    answer[ind_arr] = (double*)malloc(A->m*sizeof(double));
    if(proc == 0) {
      memmove(answer[ind_arr], solution[ind_arr/mysize], A->m*(sizeof(double)));
    } else {
      MPI_Recv(answer[ind_arr], A->m, MPI_DOUBLE,
               proc, ind_arr, MPI_COMM_WORLD, &stat);
    }
  }
} else {
  int ind_sol;
  int num_assign = number_of_rhs_assigned(num_vec, myrank);
```

```
    for(ind_sol=0; ind_sol<num_assign; ind_sol++) {
      MPI_Send(solution[ind_sol], A->m, MPI_DOUBLE,
               0, ind_sol*mysize+myrank, MPI_COMM_WORLD);
    }
  }
  printf("(%d) total: %e\n", myrank, MPI_Wtime() - t1);

  if(myrank == 0) {
    printf("output the solution...\n");
    output_solution("psolver_solution.log", answer, num_vec, A->m);
  }

  for(i=0; i<num_vec; i++) {
    free(answer[i]);
  }
  free(answer);
  free(solution);

  MPI_Finalize();

  return 0;
}
```

A.2.6 Parallel Multiple Substitutions in POSIX Threading

pt_psolver.c

```
#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

double start_time;
int mysize;
pthread_key_t key;
pthread_barrier_t barrier;

cs* A;
double** rhs;
int num_rhs;

int main(int argc, char* argv[]) {
  int An;
  pthread_t* thread;
  struct timespec tp;
  long time1, time2;

  //pthread_init();

  // begin: preparation
```

```
if(argc < 4) {
  printf("usage:\n pt_psolver matrix.mtx rhs.dat number_of_thread\n");
  return 0;
}
mysize = atoi(argv[3]);
if(mysize <= 0) {
  printf("The number of thread should be more than 0\n");
  return 0;
}

int index_thread;

thread = (pthread_t*)malloc(mysize*sizeof(pthread_t));
read_matrix(argv[1], &A);
An = A->n;
A = cs_compress(A);
num_rhs = read_rhs_array(argv[2], &rhs, A->m);

clock_gettime(CLOCK_REALTIME, &tp);
time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

clock_gettime(CLOCK_REALTIME, &tp);

// begin: core of the solver
pthread_key_create(&key, NULL);
char*** str;

pthread_barrier_init(&barrier, NULL, mysize);

str = (char***)malloc(mysize*sizeof(char**));
for(index_thread=0; index_thread<mysize; index_thread++) {
  str[index_thread] = (char**)malloc(1*sizeof(char*));
  str[index_thread][0] = (char*)malloc(8*sizeof(char));
  sprintf(str[index_thread][0], "%d", index_thread);
  pthread_create(&thread[index_thread], NULL,
         thread_task_forward_pt_psolver, str[index_thread]);
}

for(index_thread=0; index_thread<mysize; index_thread++) {
  pthread_join(thread[index_thread], NULL);

  free(str[index_thread][0]);
  free(str[index_thread]);
}

free(str);
pthread_barrier_destroy(&barrier);

clock_gettime(CLOCK_REALTIME, &tp);
time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
printf("total: %ld msec.\n", time2-time1);

printf("output...\n");
output_solution("pt_psolver_soltion.log", rhs, num_rhs, A->m);
printf("done\n");

free(thread);
```

```
  cs_spfree(A);

  return 0;
}
```

pt_psubst.c

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include "cs.h"
#include "solver.h"

extern pthread_key_t key;
extern int num_rhs;
extern double** rhs;
extern int mysize;
extern cs* A;

double* forward_multi_subst_sequential_threading(cs* A, double* b, int An) {
  int rowind;

  for(rowind=0; rowind<An; rowind++) {
    int rowind_pm;
    int colind;
    colind = A->i[A->p[rowind]];

    if(colind == rowind) {
      b[colind] /= A->x[A->p[rowind]];
    }

    for(rowind_pm=A->p[rowind]; rowind_pm<A->p[rowind+1]; rowind_pm++) {
      colind = A->i[rowind_pm];

      if(colind > rowind) {
        b[colind] -= A->x[rowind_pm]*b[rowind];
      }
    }
  }

  return b;
}

void* thread_task_forward_pt_psolver(void* param) {
  int index;
  void** str;
  struct timespec tp;
  long time1, time2;
  int i, num_assign;

  pthread_setspecific(key, ((void**)param));
  str = (void**)pthread_getspecific(key);
  index = atoi((char*)str[0]);
  num_assign = number_of_rhs_assigned(num_rhs, index);

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
```

```
  for(i=0; i<num_assign; i++) {
    rhs[mysize*i+index] =
      forward_multi_subst_sequential_threading(A, rhs[mysize*i+index], A->m);
  }

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("(%d) subst: %ld msec.\n", index, time2-time1);

  return NULL;
}
```

A.2.7 Parallel Multiple Substitutions with L/B in MPI

mpi_plbsolver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "cs.h"
#include "mpi.h"
#include "solver.h"

double start_time;
int myrank;
int mysize;
pthread_key_t key;

int main(int argc, char* argv[]) {
  cs* A;

  double** rhs;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &mysize);

  // begin: preparation
  if(argc < 3) {
    if(myrank == 0) {
      printf("usage:\n mpirun -np n solver matrix.mtx rhs.dat\n");
    }
    MPI_Finalize();
    return 0;
  }

  double t1;
  int num_vec;

  if (myrank == 0) {
    double td;
    int* assigned_list;
    double* rhs_buf;
```

```
read_matrix(argv[1], &A);
num_vec = read_rhs_array(argv[2], &rhs, A->m);
A = cs_compress(A);

t1 = MPI_Wtime();
td = MPI_Wtime();
A = distribute_matrix_bcast(A);
printf("(%d) matrix dist: %e\n", myrank, MPI_Wtime() - td);

int i;
MPI_Status stat;

printf("(%d) %d vectors\n", myrank, num_vec);

td = MPI_Wtime();
MPI_Bcast(&num_vec, 1, MPI_INT, 0, MPI_COMM_WORLD);

assigned_list = (int*)calloc(mysize-1, sizeof(int));
rhs_buf = (double*)malloc(A->m*sizeof(double));

// if mysize is too large...
if(mysize-1>A->m) {
  mysize = A->m+1;
}

for(i=1; i<mysize; i++) {
  assigned_list[i] = i-1;
  MPI_Send(rhs[i-1], A->m, MPI_DOUBLE, i, i-1, MPI_COMM_WORLD);
    //first assignments
}

for(i=mysize-1; i<num_vec; i++) {
  double* swap;
  MPI_Recv(rhs_buf, A->m, MPI_DOUBLE, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
  swap = rhs[assigned_list[stat.MPI_SOURCE]];
  rhs[assigned_list[stat.MPI_SOURCE]] = rhs_buf;
  rhs_buf = swap;

  assigned_list[stat.MPI_SOURCE] = i;

  MPI_Send(rhs[i], A->m, MPI_DOUBLE, stat.MPI_SOURCE, i, MPI_COMM_WORLD);
}

for(i=0; i<mysize-1; i++) {
  double *swap;
  MPI_Recv(rhs_buf, A->m, MPI_DOUBLE, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &stat);

  swap = rhs[assigned_list[stat.MPI_SOURCE]];
  rhs[assigned_list[stat.MPI_SOURCE]] = rhs_buf;
  rhs_buf = swap;
}

for(i=1; i<mysize; i++) {
  MPI_Send(rhs_buf, A->m, MPI_DOUBLE, i, num_vec,
```

```
            MPI_COMM_WORLD); //the end message
    }
    printf("(%d) subst: %e\n", myrank, MPI_Wtime() - td);
    free(assigned_list);
    free(rhs_buf);
    printf("(%d) total: %e\n", myrank, MPI_Wtime() - t1);
    printf("(%d) output the solution...\n", myrank);
    output_solution("plbsolver_solution.log", rhs, num_vec, A->m);
    printf("(%d) output done\n", myrank);
  } else {
    double td1;
    MPI_Status stat;
    double* work_rhs;

    A = distribute_matrix_bcast(A);

    t1 = MPI_Wtime();
    td1 = MPI_Wtime();
    MPI_Bcast(&num_vec, 1, MPI_INT, 0, MPI_COMM_WORLD);

    work_rhs = (double*)malloc(A->m*sizeof(double));
    MPI_Recv(work_rhs, A->m, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    while(stat.MPI_TAG < num_vec) {
      work_rhs = forward_subst_sequential(A, work_rhs, A->m);
      MPI_Send(work_rhs, A->m, MPI_DOUBLE, 0, stat.MPI_TAG, MPI_COMM_WORLD);
      MPI_Recv(work_rhs, A->m, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    }

    free(work_rhs);
  }

  MPI_Finalize();

  return 0;
}
```

A.2.8 Parallel Multiple Substitutions with L/B in POSIX Threading

pt_plbsolber.c

```
#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

double start_time;
int mysize;
pthread_key_t key;
pthread_barrier_t barrier;
```

```
pthread_mutex_t mutex;
int next_rhs;

cs* A;
double** rhs;
int num_rhs;

int main(int argc, char* argv[]) {
  int An;
  pthread_t* thread;
  struct timespec tp;
  long time1, time2;

  //pthread_init();

  // begin: preparation
  if(argc < 4) {
    printf("usage:\n pt_plbsolver matrix.mtx rhs.dat number_of_thread\n");
    return 0;
  }
  mysize = atoi(argv[3]);
  if(mysize <= 0) {
    printf("The number of thread should be more than 0\n");
    return 0;
  }
  int index_thread;

  thread = (pthread_t*)malloc(mysize*sizeof(pthread_t));
  read_matrix(argv[1], &A);
  An = A->n;
  A = cs_compress(A);
  num_rhs = read_rhs_array(argv[2], &rhs, A->m);

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  clock_gettime(CLOCK_REALTIME, &tp);

  // begin: core of the solver
  pthread_key_create(&key, NULL);
  char*** str;

  pthread_barrier_init(&barrier, NULL, mysize);
  pthread_mutex_init(&mutex, NULL);
  next_rhs = 0;

  printf("threading start\n");
  str = (char***)malloc(mysize*sizeof(char**));
  for(index_thread=0; index_thread<mysize; index_thread++) {
    str[index_thread] = (char**)malloc(1*sizeof(char*));
    str[index_thread][0] = (char*)malloc(8*sizeof(char));
    sprintf(str[index_thread][0], "%d", index_thread);

    pthread_create(&thread[index_thread], NULL,
        thread_task_forward_pt_plbsolver, str[index_thread]);
  }
```

```
  for(index_thread=0; index_thread<mysize; index_thread++) {
    pthread_join(thread[index_thread], NULL);

    free(str[index_thread][0]);
    free(str[index_thread]);
  }

  free(str);
  pthread_barrier_destroy(&barrier);
  pthread_mutex_destroy(&mutex);

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("total: %ld msec.\n", time2-time1);

  printf("output...\n");
  output_solution("pt_plbsolver_soltion.log", rhs, num_rhs, A->m);
  printf("done\n");

  free(thread);

  return 0;
}
```

pt_plbsubst.c

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include <stdio.h>
#include "cs.h"
#include "solver.h"

extern pthread_key_t key;
extern pthread_mutex_t mutex;
extern int num_rhs;
extern double** rhs;
extern cs* A;
extern int next_rhs;

void* thread_task_forward_pt_plbsolver(void* param) {
  int index;
  void** str;
  struct timespec tp;
  long time1, time2;

  pthread_setspecific(key, ((void**)param));
  str = (void**)pthread_getspecific(key);
  index = atoi((char*)str[0]);

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  int i;
  i = get_next_rhs();

  while(i < num_rhs) {
    rhs[i] = forward_subst_sequential(A, rhs[i], A->m);
```

```
    i = get_next_rhs();
  }

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("(%d) subst: %ld msec.\n", index, time2-time1);

  return NULL;
}

int get_next_rhs() {
  int ret;

  pthread_mutex_lock(&mutex);
  ret = next_rhs++;
  pthread_mutex_unlock(&mutex);

  return ret;
}
```

A.2.9 Parallel Multiple Substitutions with L/B and Blocking in POSIX Threading

pt_plbsolber_blocking.c

```
#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

double start_time;
int mysize;
pthread_key_t key;
pthread_barrier_t barrier;
pthread_mutex_t mutex;
int next_rhs;

cs* A;
double** rhs;
int num_rhs;

int main(int argc, char* argv[]) {
  int An;
  pthread_t* thread;
  struct timespec tp;
  long time1, time2;

  //pthread_init();

  // begin: preparation
```

```
if(argc < 4) {
  printf("usage:\n pt_plbsolver matrix.mtx rhs.dat number_of_thread\n");
  return 0;
}
mysize = atoi(argv[3]);
if(mysize <= 0) {
  printf("The number of thread should be more than 0\n");
  return 0;
}
int index_thread;

thread = (pthread_t*)malloc(mysize*sizeof(pthread_t));
read_matrix(argv[1], &A);
An = A->n;
A = cs_compress(A);
num_rhs = read_rhs_array(argv[2], &rhs, A->m);

clock_gettime(CLOCK_REALTIME, &tp);
time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

clock_gettime(CLOCK_REALTIME, &tp);

// begin: core of the solver
pthread_key_create(&key, NULL);
char*** str;

pthread_barrier_init(&barrier, NULL, mysize);
pthread_mutex_init(&mutex, NULL);
next_rhs = 0;

printf("threading start\n");
str = (char***)malloc(mysize*sizeof(char**));
for(index_thread=0; index_thread<mysize; index_thread++) {
  str[index_thread] = (char**)malloc(1*sizeof(char*));
  str[index_thread][0] = (char*)malloc(8*sizeof(char));
  sprintf(str[index_thread][0], "%d", index_thread);
  pthread_create(&thread[index_thread], NULL,
      thread_task_forward_pt_plbsolver, str[index_thread]);
}

for(index_thread=0; index_thread<mysize; index_thread++) {
  pthread_join(thread[index_thread], NULL);

  free(str[index_thread][0]);
  free(str[index_thread]);
}

free(str);
pthread_barrier_destroy(&barrier);
pthread_mutex_destroy(&mutex);

clock_gettime(CLOCK_REALTIME, &tp);
time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
printf("total: %ld msec.\n", time2-time1);

printf("output...\n");
output_solution("pt_plbsolver_soltion.log", rhs, num_rhs, A->m);
```

```
  printf("done\n");

  free(thread);

  return 0;
}
```

pt_plbsolsubst_blocking.c

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include <stdio.h>
#include "cs.h"
#include "solver.h"

extern pthread_key_t key;
extern pthread_mutex_t mutex;
extern int num_rhs;
extern double** rhs;
extern cs* A;
extern int next_rhs;
extern int block;

void* thread_task_forward_pt_plbsolver_blocking(void* param) {
  int index;
  void** str;
  struct timespec tp;
  long time1, time2;
  int block_min, block_max;

  pthread_setspecific(key, ((void**)param));
  str = (void**)pthread_getspecific(key);
  index = atoi((char*)str[0]);

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  block_min = get_next_rhs_blocking();

  while(block_min < num_rhs) {
    int j;
    block_max = block_min+block <= num_rhs ? block_min+block : num_rhs;
    for(j=block_min; j<block_max; j++) {
      rhs[j] = forward_subst_sequential(A, rhs[j], A->m);
    }
    block_min = get_next_rhs_blocking();
  }

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("(%d) subst: %ld msec.\n", index, time2-time1);

  return NULL;
}

int get_next_rhs_blocking() {
  int ret;
```

```
  pthread_mutex_lock(&mutex);
  ret = next_rhs;
  next_rhs += block;
  pthread_mutex_unlock(&mutex);

  return ret;
}
```

A.2.10 Parallelized Forward Substitution in MPI

mpi_solver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "mpi.h"
#include "cs.h"
#include "solver.h"

double start_time;
int myrank;
int mysize;

int main(int argc, char* argv[]) {
  cs* A;
  double* b;
  int An;

  cs* pmat;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &mysize);

  // begin: preparation
  if(argc < 2) {
    if(myrank == 0) {
      printf("usage: mpirun -np n mpi_solver matrix.mtx\n");
    }
    MPI_Finalize();
    return 0;
  }

  double t1, t2;

  if (myrank == 0) {
    int proc;
    cs** partial_matrix;

    read_matrix(argv[1], &A);
    A = cs_compress(A);

    MPI_Barrier(MPI_COMM_WORLD);
```

```
    t1 = MPI_Wtime();
    An = A->n;
    double tp1, tp2;

    tp1 = MPI_Wtime();
    partial_matrix = split_matrix(A, 1);

    tp2 = MPI_Wtime();
    printf("(%d) split matrix: %e\n", myrank, tp2-tp1);

    MPI_Barrier(MPI_COMM_WORLD);
    tp1 = MPI_Wtime();
    distribute_matrix_send(A, partial_matrix, 1);
    tp2 = MPI_Wtime();
    printf("(%d) distribute matrix send: %e\n", myrank, tp2-tp1);

    pmat = partial_matrix[0];

    for(proc=1; proc<mysize; proc++) {
      cs_spfree(partial_matrix[proc]);
    }

  } else {
    double td1, td2;

    MPI_Barrier(MPI_COMM_WORLD);
    t1 = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD);
    td1 = MPI_Wtime();
    distribute_matrix_recv(&pmat, 1);
    td2 = MPI_Wtime();
    printf("(%d) distribute matrix receive: %e\n", myrank, td2-td1);
  }

  MPI_Bcast(&An, 1, MPI_INT, 0, MPI_COMM_WORLD);
  b = (double*)malloc(An*sizeof(double));
  gen_rhs(An, b);

  t2 = MPI_Wtime();
  printf("(%d) prepare: %e\n", myrank, t2-t1);
  // end: preparation

  // begin: core of the solver
  double* answer;

  double tc;
  tc = MPI_Wtime();
  answer = forward_subst_nonblocking(pmat, b, An);
  printf("(%d) core: %e\n", myrank, MPI_Wtime() - tc);
  t2 = MPI_Wtime();
  printf("(%d) total: %e\n", myrank, t2-t1);

  if(myrank == 0) {
    output_array_double("answer.log", answer, An);
    printf("\n");
  }
  free(b);
```

```
  MPI_Finalize();

  return 0;
}
```

mpi_subst.c

```
#include "mpi.h"
#include "cs.h"
#include "solver.h"

extern int myrank;
extern int mysize;

double* forward_subst_nonblocking(cs* Ap, double* b, int An) {
  int rowind;
  int colmax;
  double time1, acc;
  acc = 0;

  colmax = Ap->n;

  for(rowind=0; rowind<colmax; rowind++) {
    int proc_solve;
    int rowind_pm;

    proc_solve = proc_number_solves_row(rowind, 1);
    int colind_mat = get_matrix_index(myrank, Ap->i[Ap->p[rowind]], 1);

    if(colind_mat == rowind) {
      b[colind_mat] /= Ap->x[Ap->p[rowind]];
    }
    time1 = MPI_Wtime();
    MPI_Bcast(&b[rowind], 1, MPI_DOUBLE, proc_solve, MPI_COMM_WORLD);
    acc += MPI_Wtime() - time1;

    for(rowind_pm=Ap->p[rowind]; rowind_pm<Ap->p[rowind+1]; rowind_pm++) {
      colind_mat = get_matrix_index(myrank, Ap->i[rowind_pm], 1);

      if(colind_mat > rowind) {
b[colind_mat] -= Ap->x[rowind_pm]*b[rowind];
      }
    }
  }

  for(rowind=colmax; rowind<An; rowind++) {
    int proc_solve;

    proc_solve = proc_number_solves_row(rowind, 1);

    time1 = MPI_Wtime();
    MPI_Bcast(&b[rowind], 1, MPI_DOUBLE, proc_solve, MPI_COMM_WORLD);
    acc += MPI_Wtime() - time1;
  }

  printf("(%d) MPI time = %e\n", myrank, acc);
```

```
  return b;
}
```

A.2.11 Paralellized Forward Substitution in POSIX Threading

pt_solver.c

```
#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>

#include "cs.h"
#include "solver.h"

#define FALSE 0
#define TRUE -1

int mysize;
double start_time;
pthread_key_t key;
pthread_barrier_t barrier;

int main(int argc, char* argv[]) {
  int An;
  pthread_t* thread;
  struct timespec tp;
  long time1, time2;

  cs* A;
  double* b;

  //pthread_init();

  // begin: preparation
  if(argc < 3) {
    printf("usage:\n pt_solver matrix.mtx number_of_thread\n");
    return 0;
  }
  mysize = atoi(argv[2]);
  if(mysize <= 0) {
    printf("The number of thread should be more than 0\n");
    return 0;
  }

  cs** partial_matrix;
  int index_thread;

  thread = (pthread_t*)malloc(mysize*sizeof(pthread_t));
  read_matrix(argv[1], &A);
  An = A->n;
  A = cs_compress(A);
```

```
clock_gettime(CLOCK_REALTIME, &tp);
time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

partial_matrix = split_matrix(A, 1);
clock_gettime(CLOCK_REALTIME, &tp);
time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
printf("split: %ld msec.\n", time2-time1);

// begin: core of the solver

b = (double*)malloc(An*sizeof(double));
gen_rhs(An, b);
pthread_key_create(&key, NULL);
char*** str;

pthread_barrier_init(&barrier, NULL, mysize);

str = (char***)malloc(mysize*sizeof(char**));
for(index_thread=0; index_thread<mysize; index_thread++) {
  str[index_thread] = (char**)malloc(5*sizeof(char*));
  str[index_thread][0] = (char*)malloc(5*sizeof(char));
  str[index_thread][1] = (char*)malloc(30*sizeof(char));
  str[index_thread][4] = (char*)malloc(32*sizeof(char));

  sprintf(str[index_thread][0], "%d", index_thread);
  sprintf(str[index_thread][1], "thread %d\n", index_thread);
  str[index_thread][2] = (void*)partial_matrix[index_thread];
  str[index_thread][3] = (void*)b;
  sprintf(str[index_thread][4], "%d", An);
  pthread_create(&thread[index_thread], NULL,
      thread_task_forward_pt_solver, str[index_thread]);
}

for(index_thread=0; index_thread<mysize; index_thread++) {
  int i;

  pthread_join(thread[index_thread], NULL);
  printf("(%d) joined\n", index_thread);

  for(i=0; i<2; i++) {
    free(str[index_thread][i]);
  }
  free(str[index_thread][4]);

  free(str[index_thread]);
}

free(str);
pthread_barrier_destroy(&barrier);

clock_gettime(CLOCK_REALTIME, &tp);
time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
printf("total: %ld msec.\n", time2-time1);

output_array_double("answer.log", b, An);

free(b);
```

```
  free(thread);

  return 0;
}
```

pt_subst.c

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include <time.h>
#include "cs.h"
#include "solver.h"

extern pthread_barrier_t barrier;
extern pthread_key_t key;

double* forward_subst_nonblocking_threading
(cs* Ap, double* b, int An, int thread_index) {
  int rowind;
  int colmax;

  colmax = An;

  for(rowind=0; rowind<Ap->n; rowind++) {
    int proc_solve;
    int rowind_pm;

    proc_solve = proc_number_solves_row(rowind, 1);
    int colind_mat = get_matrix_index(thread_index, Ap->i[Ap->p[rowind]], 1);

    if(colind_mat == rowind) {
      b[colind_mat] /= Ap->x[Ap->p[rowind]];
    }

    pthread_barrier_wait(&barrier);

    for(rowind_pm=Ap->p[rowind]; rowind_pm<Ap->p[rowind+1]; rowind_pm++) {
      colind_mat = get_matrix_index(thread_index, Ap->i[rowind_pm], 1);
      if(colind_mat > rowind) {
        b[colind_mat] -= Ap->x[rowind_pm]*b[rowind];
      }
    }
  }

  for(rowind=Ap->n; rowind<colmax; rowind++) {
      pthread_barrier_wait(&barrier);
  }

  return b;
}

void* thread_task_forward_pt_solver(void* param) {
  int index;
  char** str;
  cs* Ap;
  double* b;
```

```
  int An;
  struct timespec tp;
  long time1, time2;

  pthread_setspecific(key, ((char**)param));
  str = (char**)pthread_getspecific(key);
  index = atoi(str[0]);

  Ap = (cs*)str[2];
  b = (double*)str[3];
  An = atoi(str[4]);

  clock_gettime(CLOCK_REALTIME, &tp);
  time1 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;

  forward_subst_nonblocking_threading(Ap, b, An, index);

  clock_gettime(CLOCK_REALTIME, &tp);
  time2 = ((long)(tp.tv_sec))*1000+((long)(tp.tv_nsec))/1000000;
  printf("(%d) subst: %ld msec.\n", index, time2-time1);

  return NULL;
}
```

Bibliography

[1] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 1–6, 1987.

[2] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, 2006.

[4] MPI Forum. Message Passing Interface Forum Web Page. *http://www.mpi-forum.org/*, 2008.

[5] J. R. Gilbert and T. Peierls. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM Journal on Scientific Computing*, 9:pp. 862–874, 1988.

[6] GNU. GNU portable threads, `http://www.gnu.org/software/pth/`. 2006.

[7] A. Gupta and V. Kumar. Parallel Algorithms for Forward and Back Substitution in Direct Solution of Sparse Linear Systems. *Proceedings of the IEEE/ACM Conferrence on Supercomputing*, page No. 74, 1995.

[8] Algonne National Laboratory. MPICH Home Page, `http://www-unix.mcs.anl.gov/mpi/mpich1/`. 2008.

[9] S. Parter. The Use of Linear Graphs in Gauss Elimination. *SIAM Rev.*, 3:pp. 119–130, 1961.

[10] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, 2nd edition edition, 1994.

[11] R. Schreiber. A New Implementation of Sparse Gaussian Elimination. *ACM Trans. Math. Software*, 8:pp. 256–276, 1982.

[12] Wolfgang Schreiner. Personal Communication on 16th June, 2008.

[13] Silicon Graphics, Inc. SGI Altix 4700 Datasheet, `http://www.sgi.com/pdfs/3867.pdf`. 2008.

[14] Silicon Graphics, Inc. SGI Official Webpage, `http://www.sgi.com/`. 2008.

[15] Gabor Bodnar Peter Stadelmeyer. Personal Communication on 6th May, 2008. 2008.

[16] Volker Strassen. Gausian Elimination is not Optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[17] Robin J. Wilson. *Introduction to Graph Theory*. Longman, 3rd ed. edition.

Kenji Miyamoto

30th June 2008

Studentenheim des OÖ Studentenwerks 4210,
Softwarepark 23, A-4232 Hagenberg, Austria

Phone: +43-7236-7973-4210 – Cell: +43-699-1300-9065
Email Address: i07005@isi-hagenberg.at

Date of birth	31st August, 1979
Citizenship	Japanese
Education	Master Course Student (From October 2007 to present) International School for Informatics, Johannes Kepler University, Austria PhD study at Kyoto University, Kyoto, Japan, (from April 2004 to March 2007) Master of Informatics Kyoto University, Kyoto, Japan Supervisor: Masahiko Sato Thesis title: A Modal Foundation for Secure Information Flow Graduation date: March 2004 Bachelor of Engineering Keio University, Tokyo, Japan Major: Electronics and Electrical Engineering Graduation date: March 2002
Publications	Kenji Miyamoto and Atsushi Igarashi. A modal foundation for secure information flow., *Proceedings of the Workshop on Foundations of Computer Security (FCS'04)*, pages 187–203, 2004. Takahiro Seino, Izumi Takeuti and Kenji Miyamoto. A verification of a database management system by π-calculus. (in Japanese), *Proceedings of the 23rd Symposium of Japan Society for Software Science and Technology*, 2006.

Eidesstattliche Erklärung:

Ich erkläre an Eides statt, das ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäßentnommenen Stellen als solche kenntlich gemacht habe.