LINEAR STATIC ANALYSIS
OF
LARGE STRUCTURAL MODELS ON PC CLUSTERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SEMİH ÖZMEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CIVIL ENGINEERING

JULY 2009

Approval of the thesis:

LINEAR STATIC ANALYSIS
OF
LARGE STRUCTURAL MODELS ON PC CLUSTERS

submitted by **SEMİH ÖZMEN** in partial fullfillment of the requirements for the degree of **Master of Science in Civil Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Güney Özcebe _____
Head of Department, **Civil Engineering**

Asst. Prof. Dr. Özgür Kurç _____
Supervisor, **Civil Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Uğur Polat _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Özgür Kurç _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Ayşegül Askan Gündoğan _____
Civil Engineering Dept., METU

Inst. Dr. Afşin Sarıtaş _____
Civil Engineering Dept., METU

Asst. Prof. Dr. Nilay Sezer Uzol _____
Mechanical Engineering Dept., TOBB ETU

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name	:	Semih Özmen
Signature	:	

ABSTRACT

LINEAR STATIC ANALYSIS
OF
LARGE STRUCTURAL MODELS ON PC CLUSTERS

Özmen, Semih

M.S, Department of Civil Engineering

Supervisor: Asst. Prof. Dr. Özgür Kurç

July 2009, 106 pages

This research focuses on implementing and improving a parallel solution framework for the linear static analysis of large structural models on PC clusters. The framework consists of two separate programs where the first one is responsible from preparing data for the parallel solution that involves partitioning, workload balancing, and equation numbering. The second program is a fully parallel finite element program that utilizes substructure based solution approach with direct solvers.

The first step of data preparation is partitioning the structure into substructures. After creating the initial substructures, the estimated imbalance of the substructures is adjusted by iteratively transferring nodes from the slower substructures to the faster ones. Once the final substructures are created, the solution phase is initiated. Each processor assembles its substructure's stiffness matrix and condenses it to the interfaces. The interface equations are then solved in parallel with a block-cyclic dense matrix solver. After computing the interface unknowns, each processor calculates the internal displacements and element stresses or forces. Comparative tests were done to demonstrate the performance of the solution framework.

Keywords: Linear Static Analysis, High Performance Computing, Substructuring, Workload Balancing, Direct Solvers

ÖZ

BİLGİSAYAR KÜMELERİ KULLANILARAK BÜYÜK YAPI MODELLERİNİN DOĞRUSAL STATİK OLARAK ÇÖZÜMLENMESİ

Özmen, Semih

Yüksek Lisans, İnşaat Mühendisliği Bölümü

Tez Yöneticisi: Yard. Doç. Dr. Özgür Kurç

Temmuz 2009, 106 sayfa

Bu çalışma, bilgisayar kümeleri kullanılarak büyük yapı modellerinin doğrusal statik çözümlemesini yapabilen bir yazılımın geliştirilmesini hedeflemektedir. Çözümleme yazılımı iki ayrı yazılımdan oluşmaktadır. İlk yazılım, paralel çözümleme yazılımına veri hazırlamaktadır. İkinci yazılım ise alt-yapılar üzerinde direk çözümleme tekniği kullarak bütünüyle paralel çözümleme yapabilen bir sonlu elemanlar programıdır.

Veri hazırlama yazılımı alt-yapıları çözüm için olabilecek en uygun şekilde oluşturmaya çalışır. Alt-yapıların ilk sefer parçalanmasının ardından, alt-yapıların çözümlenmesi için gerekecek işlem adedi, her bir bilgisayarın işlem gücü ve o alt-yapıyı oluşturan elemanların adedi kullanılarak alt-yapılar arası iş yükü farkı hesaplanır ve bu fark yavaş çözümlenen alt-yapılardan hızlı çözümlenenlere doğru düğüm noktası aktarılarak dengelenmeye çalışılır. Alt-yapıların oluşturulmasından sonra direngenlik matrisi denklemleri en uygun şekilde sıralanır ve paralel çözümlemeye başlanır. Öncelikle, alt-yapı direngenlik matrisleri oluşturulur ve bu matrisler sınır düğümlerine indirgenir. Sonrasında, sınır denklemlerinin çözümü paralel olarak blok-çevrimsel yoğun matris çözücü ile gerçekleştirilir ve her bilgisayar bu çözümün sonuçlarını kullanarak düğüm deplasmanlarını ve eleman kuvvetlerini hesaplar. Çözüm yazılımının bütününün başarımını irdelemek amacıyla örnek problemler çözülmüş ve sonuçları değerlendirilmiştir.

Anahtar Kelimeler: Doğrusal Statik Çözümleme, Yüksek Başarımlı Hesaplama, Alt-yapılara Bölme, İş Yükü Bölüştürme, Direk Çözümleme

To my dearest love

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor, Asst. Prof. Dr. Özgür Kurç, because of his great guidance, support, and complete confidence in me during the whole research, publications, and presentations. Without his existence and experience, this research could not be completed.

Another person that supports this research in collaboration with his thesis research is my dear friend Tunç Bahçecioğlu. In any obstacle, I always felt comfortable with his support.

I wish to thank Asst. Prof. Dr. Julien Langou for his assistance about the LAPACK and ScaLAPACK routines, their optimizations, and binary compilations.

Another friend that gave a hand indirectly is Lasse Reinhold with his compression algorithm and suggestions on optimization of it.

Thanks to Eyyüp Volkan Çektimur, for his suggestions and comments on some critical points of the algorithm.

I would like to thank ComoSYS developers and especially Mr. Ateeq Ahmad for their understanding and support due to the fact that I spent some of my working hours to this study.

I want to exhibit my special thanks to my wife Özlem for her support and patience during this heavy work and also for her existence which gives me strength.

Moreover, I wish to thank my friends and my family for their existence, thus, I could find the opportunity to relax.

Thanks to *Scientific and Technological Research Council of Turkey* (TÜBİTAK), because of their financial support during this research.

This study is supported by Grant No: BAP-2007-03-03-09 from Middle East Technical University Scientific Research Program.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ v
ACKNOWLEDGMENTS vii
TABLE OF CONTENTS viii
LIST OF FIGURES xii
LIST OF TABLES xiv
CHAPTERS

1 INTRODUCTION 1
1.1 Problem Definition 1
1.2 Related Work 3
1.2.1 Matrices and Matrix Storage Schemes 3
1.2.2 Solution Methods for System of Linear Equations 5
1.2.3 Direct Solution Methods 6
1.2.3.1 Background 6
1.2.3.2 Direct Solvers for Dense Matrices 10
1.2.3.2.1 Cholesky Factorization in LAPACK 12
1.2.3.2.2 Cholesky Factorization in ScaLAPACK 13
1.2.3.3 Direct Solvers for Sparse Matrices 14
1.2.4 Substructuring 17
1.2.4.1 Partitioning 18
1.2.4.2 Workload Balancing 19
1.2.4.3 Substructure based Solution Methods 20
1.2.5 PC Clusters and Communication Interfaces 23

1.3 Objectives and Scope 24
1.4 Thesis Outline 24

2 AN OVERVIEW OF THE PARALLEL SOLUTION FRAMEWORK 26
2.1 Introduction 26
2.2 Algorithm 26
2.3 Mathematical Background 27
2.3.1 Condensation 28
2.3.2 Substructure based solution 29

3 DATA PREPARATION 33
3.1 Introduction 33
3.2 Overview 33
3.2.1 Partitioning Libraries 35
3.2.1.1 METIS 36
3.2.1.2 PARMETIS 37
3.3 Workload Balancing Algorithm 37
3.4 Test Results 40
3.4.1 Illustrative Example 41
3.4.2 Actual Testing Models 42

4 PARALLEL SOLUTION 48
4.1 Introduction 48
4.2 Parallel Solution Algorithm 48
4.3 Condensation 51
4.3.1 Algorithm 51
4.3.2 Test Results 53
4.4 Interface System 55
4.4.1 Interface System Assembly 56
4.4.1.1 Data Mapping 58
4.4.1.1.1 Block Cyclic Data Distribution 59
4.4.1.1.2 Data Mapping for Block Cyclic Distribution 60
4.4.1.2 Communication Scheme 64

4.4.1.3 Data Compression 65
4.4.2 Interface System Solution 66
4.4.2.1 Implementation 66
4.4.2.2 Improvements for Speed-up 67
4.4.2.3 Execution Time and Communication Volume Estimations 68
4.4.3 Test Results 69
4.4.3.1 Performance of The Interface System Assembly 70
4.4.3.1.1 Performance Improvement by Runtime Data Compression 71
4.4.3.2 Performance of The Interface System Solution 71
4.4.3.2.1 Execution Time Estimation and Comparison with Actual Timings 71
4.4.3.2.2 Speed-up and Efficiency 73

5 OVERALL PERFORMANCE OF PARALLEL SOLUTION FRAMEWORK 76
5.1 Method 76
5.2 Test Results 77
5.2.1 Cube30 77
5.2.2 GroM 79
5.2.3 StRegis 81
5.2.4 Shared Memory Simulation 83
5.3 Discussion of Overall Results 85

6 CONCLUSION AND FUTURE PLANS 87
6.1 Conclusion 87
6.2 Future Plans 89
REFERENCES 91
APPENDICES

A PERFORMANCE ASSESSMENT OF PARALLEL ALGORITHMS 99
A.1 General features of parallel processing 99
A.2 Time measures 100
A.3 Computational speed 101

A.4 Parallel measurement 101

B STRUCTURAL MODELS USED FOR PERFORMANCE TESTS 104

B.1 Cube30 104

B.2 GroM 105

B.3 StRegis 106

LIST OF FIGURES

FIGURES

Figure 1.1 Memory access patterns for variants of Cholesky Decomposition . 8
Figure 1.2 ScaLAPACK Software Hierarchy [1] 11
Figure 1.3 A snapshot of block Cholesky factorization[2] 13

Figure 2.1 Parallel Solution Flow Chart Demonstration 27
Figure 2.2 An Example Profile of a Stiffness Matrix Assembled for Condensation 28
Figure 2.3 Internal and Interface dofs of Substructures 30

Figure 3.1 Graph Representation of an arbitrary structure 36
Figure 3.2 Multilevel Approach [3] 37
Figure 3.3 Nodal Graph and Subgraphs during iterations 38
Figure 3.4 The Flow Chart of the Workload Balancing Algorithm 39
Figure 3.5 Substructures of 2DMesh Model at each workload balancing step 43
Figure 3.6 Substructures of 2DMesh Model at each workload balancing step (cont.) 44
Figure 3.7 Final Partitioning of Cube30 Model 45
Figure 3.8 Final Partitioning of GroM Model 46
Figure 3.9 Final Partitioning of StRegis Model 47

Figure 4.1 Parallel Solution Flow Chart 50
Figure 4.2 Non-zero element pattern for interface stiffness matrix 56
Figure 4.3 The contributions of each substructure to interface stiffness matrix 56
Figure 4.4 Redistribution over cluster 57
Figure 4.5 On the Fly Assembly 58
Figure 4.6 Example of a block cyclic data distribution [2] 60
Figure 4.7 2D Block Cyclic Mapping Parameters 61

Figure 4.8 2D Block Cyclic Mapping 63
Figure 4.9 Data buffers to be sent to relevant computers 63
Figure 4.10 Runtime Data Compression 65
Figure 4.11 Interface Solution Prediction vs Actual Timings 73
Figure 4.12 Interface Solution Speed-Up Graphs 74
Figure 4.13 Interface Solution Efficiency Graphs 75

Figure 5.1 Test Results for Cube30 Model on HPCE cluster 78
Figure 5.2 Test Results for Cube30 Model on HPCE2 cluster 78
Figure 5.3 Speed-ups for Cube30 Model 79
Figure 5.4 Test Results for GroM Model on HPCE cluster 80
Figure 5.5 Test Results for GroM Model on HPCE2 cluster 80
Figure 5.6 Speed-up Graphs for GroM Model 81
Figure 5.7 Test Results for StRegis Model on HPCE cluster 82
Figure 5.8 Test Results for StRegis Model on HPCE2 cluster 82
Figure 5.9 Speed-up Graphs for StRegis Model 83
Figure 5.10 Total Solution Times on a Multi-processor Computer in HPCE2 Cluster 84

Figure A.1 Data Transfer vs Time Graphs 100

Figure B.1 Cube30 Model 105
Figure B.2 GroM Model 105
Figure B.3 StRegis Model 106

LIST OF TABLES

TABLES

Table 1.1 Matrix Storage Schemes 4

Table 3.1 Workload Balancing Iteration Results for 2DMesh Model 41
Table 3.2 Workload Balancing Iteration Results for StRegis Model 42
Table 3.3 Workload Balancing Iteration Results for GroM Model 45

Table 4.1 Condensation Timings 53
Table 4.2 Condensation Speeds for HPCE Computers 54
Table 4.3 Condensation Speeds for HPCE2 Computers 55
Table 4.4 Forming Communication Scheme 64
Table 4.5 Interface System Assembly Timings for HPCE2 70
Table 4.6 Speed-up obtained by Runtime Data Compression 71
Table 4.7 Execution time and communication volume estimations on 4 computers of HPCE 72
Table 4.8 Execution time and communication volume estimations on 4 computers of HPCE2 72

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

A few decades ago, like any other discipline, structural engineers and companies started to use computers for structural analyses as computers and related software became more available. At that time, considerable amount of time was spent to input the mathematical model to the computer and the computation could last hours or even nights for medium size structural models, having 100-1000 nodes. In time, computers got faster in clock time speeds and had larger memory capacities. Such technological developments improved the computational experiences of engineers. Initially, increasing computational capabilities of computers allowed software developers to build graphical user interfaces which were not only allowed inputting very complex structures with many elements but also capable of rendering objects in 3D. In addition, increase in the speed of processors shortened overnight analysis to seconds for medium size models. In this case, however, structural engineers felt more comfortable to create larger models by using advanced graphical user interfaces and also they tended to utilize more advanced and complex finite elements, more load combinations, and even more detailed analysis methods. Today, it is very common to see structural models having hundreds of thousands of nodes created for analysis and design of structures. Moreover, for very special projects, structural engineers could create models that required the solution of billion equations [4]. Thus, the computation demand of engineers keeps increasing together with the improvement in the computer technology.

Just a few years ago, the increase in the clock speeds of processors almost stopped due to the space limitations, power, and cooling requirements for processors. As Herb Sutter says "*light isn't getting any faster, free lunch is over.*" [5]. These physical limi-

tations forced the processor manufacturers to change their direction. They started to produce processors having more than one processing unit rather than trying to increase the clock speed of a single one. Therefore, to balance the expectations of the structural engineers, new solution strategies that can utilize the available multi-processor systems more efficiently are necessary. Thus, although it appears for more than two decades, parallel computing techniques are one of the remedy to this problem. These techniques are mainly composed of using the power of more than one processor to solve a single problem.

The necessity for new solution strategies become more apparent when the design of a structure is investigated in detail. The design of a structure is an iterative process of analysis and design stages. This process can be subjected to more repetitions due to the possible modifications in architectural, economical, and manufacturing requirements. For each of these modifications, the structural model may need to be updated, re-analyzed and re-designed. Analysis step is the one that consumes considerable amount of computational resource. When the size and the details of the model become larger, the finite element procedure requires more time to solve the system. Depending on the problem, different kinds of solution techniques are implemented with finite element methods. Most of the analysis methods, have the similar solution procedure that can be grouped into three major steps [6]. First one is the generation of element stiffness matrices, and equivalent nodal loads and assembling them into structural stiffness and force matrices. Besides, if dynamic analysis is performed, mass and damping matrices must also be computed. Second step is the solution of the following linear system of equations;

$$[K] \, . \, \{u\} = \{F\} \tag{1.1}$$

for u. In this linear system of equations, K is the $n \times n$ positive-definite symmetric stiffness matrix, F is a vector of size n, alternatively right hand side (RHS) vector, representing the loading at each DOF, and u is a vector of size n, respresenting the unknown displacements corresponding to each loading. The final step is the computation of element forces and stresses using the calculated displacements.

In order to decrease the time spent during analysis step, readily available systems can be utilized. When the structural engineering design offices are considered, the readily available systems are usually the network of PC's running Windows OS. A parallel solution system that can use this system can be considerably useful for them. This way,

not only the time required for the analysis will decrease but also existing computational power in these offices will be utilized more efficiently, without purchasing any additional hardware [6].

As a result, the structural engineering industry will benefit significantly from a solution algorithm that utilizes the existing computational power of the design offices, which is optimized for linear static solutions with multiple loading conditions and which is able to decrease the analysis time notably. Because of that, this study will focus on a parallel linear static solution of large structures on PC clusters.

1.2 Related Work

Linear system of equations can be represented by matrices and corresponding load vectors. These matrices and load vectors are stored in the computer's memory with different storage schemes. The storage scheme is an important factor for the memory utilization and speed of the computation. Another factor that mainly affects the performance of the computation is the method that is utilized for the solution of system of equations. Different solution methods, their advantages, and disadvantages will be discussed in detail. Besides, parallel implementations of these methods and parallel solution environments proposed as a remedy for increasing the computational efficiency will also be presented.

1.2.1 Matrices and Matrix Storage Schemes

Matrices can be classified according to the occurrence of their non-zero terms. Frequently encountered matrix types are dense, band, and sparse matrices. Dense matrices has few number of non-zero terms or even not at all. On the contrary, "*a sparse matrix is a matrix populated primarily with zeros*" [7]. As a special form of sparse matrix, band matrix is the one whose non-zero entries are confined to a diagonal band, comprising the main diagonal and zero or more diagonals on either side.

Depending on the matrix type, matrix storage schemes, namely how the matrices are stored in computer memory, differ. The conventional way of storing a matrix is utilizing a two-dimensional array. This storage may be reasonable when the matrix is dense. However, if the matrix is sparse, such storage is consumptive as the majority of the elements of the matrix are zero and need not to be stored explicitly. For sparse matrices, the common practice is to store only the non-zero entries and to keep track

of their locations in the matrix through an indexing scheme. There are a variety of specialized indexing schemes utilized to store sparse matrices. These specialized schemes not only reduce memory consumption but also yield computational savings [8]. Since the locations of the non-zero elements in the matrix are known explicitly, unnecessary computations involving zeros can be avoided. However, the indexing based storage schemes increase the number of non-contiguous memory accesses and this may increase the application execution time [9]. Like these storage schemes, by using symmetry of the matrix or any pattern of the non-zero terms, further minimization of the memory required for storage can be achieved.

In literature, there are different types of sparse matrix storage schemes. Frequently used ones are *Coordinate Storage (CS), Compressed Row Storage (CRS), Compressed Column Storage (CCS), Blocked Compressed Row Storage (BCRS), Compressed Diagonal Storage (CDS), Jagged Diagonal Scheme (JDS)* and *Skyline Storage* (SS).

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix} \tag{1.2}$$

Table 1.1: Matrix Storage Schemes

Coordinate Storage (CS)	Compressed Row Storage (CRS)
$Value = \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} & a_{33} & a_{34} & a_{43} & a_{44} \end{pmatrix}$ $RowIndeces = \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \end{pmatrix}$ $ColumnIndeces = \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 3 & 2 & 3 \end{pmatrix}$	$Value = \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} & a_{33} & a_{34} & a_{43} & a_{44} \end{pmatrix}$ $ColumnIndeces = \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 3 & 2 & 3 \end{pmatrix}$ $RowStartingIndeces = \begin{pmatrix} 0 & 2 & 4 & 6 \end{pmatrix}$
Compressed Column Storage (CCS)	**Blocked CRS (BCRS)**
$Value = \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} & a_{33} & a_{34} & a_{43} & a_{44} \end{pmatrix}$ $RowIndeces = \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 3 & 2 & 3 \end{pmatrix}$ $ColumnIndeces = \begin{pmatrix} 0 & 2 & 4 & 6 \end{pmatrix}$	$Value = \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} & a_{33} & a_{34} & a_{43} & a_{44} \end{pmatrix}$ $ColumnIndeces = \begin{pmatrix} 0 & 2 \end{pmatrix}$ $RowIndeces = \begin{pmatrix} 0 & 2 \end{pmatrix}$
Compressed Diagonal Storage (CDS)	**Jagged Diagonal Storage (JDS)**
$Value = \begin{pmatrix} 0 & a_{12} & 0 & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \\ a_{21} & 0 & a_{43} & 0 \end{pmatrix}$	$Value = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$ $ColumnIndeces = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 2 & 3 \\ 2 & 3 \end{pmatrix}$
Skyline Storage (SS)	
$UpperValues = \begin{pmatrix} a_{11} & a_{22} & a_{12} & a_{33} & a_{44} & a_{34} \end{pmatrix}$	$UpperStartIndeces = \begin{pmatrix} 0 & 1 & 3 & 4 \end{pmatrix}$
$LowerValues = \begin{pmatrix} a_{11}^* & a_{22}^* & a_{21} & a_{33}^* & a_{44}^* & a_{43} \end{pmatrix}$	$LowerStartIndeces = \begin{pmatrix} 0 & 1 & 3 & 4 \end{pmatrix}$

In all of the above sparse matrix storage schemes, with the exception of the CDS scheme which is also known as packed banded matrix storage [10], the indexing is handled by additional data structures like arrays or matrices that stores indices of values. These different data structures typically necessitate non-contiguous multiple memory system accesses which hinders performance. In order to access a_{ij}, first the place of the value in value list should be calculated by accessing and using index lists. Therefore, these multiple indirect accesses are difficult for the compiler to optimize for fast memory accesses and resulting in poor performance [11]. If it is possible to implement such a specialized algorithm that can utilize CDS, no additional accesses are required. In contrast, two index list accesses are required to obtain a_{ij} in CS. Although implementation of this scheme is easier than CRS or CCS, it requires more memory space. Also, skyline storage is generally advantageous because after an initial access to index list, no accesses is required through that active column or row.

1.2.2 Solution Methods for System of Linear Equations

The system of linear equations arising from a linear solution of a structural model with n degree of freedoms (DOF) is represented by Equation 1.1 as

$$[K] \, . \, \{u\} = \{F\}$$

In case of multiple loading conditions, there are multiple right hand sides, and the solution is performed for each right hand side. Therefore, for each right hand side, corresponding displacements are obtained.

The stiffness matrices resulted from the linear finite element method are symmetric and positive definite. Besides, the stiffness matrices are generally sparse. In literature, there are mainly two kinds of methods for the solution of linear system of equations:

- **Direct Methods**: These methods give the exact solution of a linear system of equations by performing known number of operations. There are mainly two different approaches in direct methods: first is finding the inverse of stiffness matrix K and just multiplying it with F vector, second is the transforming the coefficient matrix into triangular or diagonal form in order to eliminate the coupling between equations. First method requires a lot of operations so it is seldom used. The most commonly used transformation based direct methods are Gauss elimination and LU decomposition in general, LL^T decomposition for symmetric, positive-definite coefficient matrices and LDL^T decomposition for symmetric coefficient matrices.

- **Iterative Methods**: These methods are similar to trial and error. They start with an initial guess and try to converge to the result by refining the solution at each iteration step. Due to their iterative behaviour, preconditioning techniques are utilized to reduce the number of iterations and to guarantee the convergence of solution. Mostly used examples are Jacobi Method, Gauss-Seidel Method and Conjugate Gradient Method.

For parallel solution, iterative methods are scalable and require less memory compared to direct methods, allowing the solution of very large problems with limited computational resources. However, the convergence of iterative methods depend on the preconditioner used for a problem, and the runtime of iterative methods is unpredictable due to their iterative nature. Additionally, iterative methods can be inefficient for analyzing structures for multiple load cases since the entire solution start from scratch for each right hand side vector.

The direct methods, on the other hand, factorize stiffness matrix for only once, then, the system of equations can be solved efficiently for multiple right hand side vectors without the need of any additional factorization. The sparsity of a system is used to minimize the arithmetic operation and data storage required for the solution. These methods have high numerical precision and guarantee the solution within a predictable amount of time if computational resources are adequate. Direct methods are often the method of choice because finding and computing a good preconditioner for an iterative method can be computationally more expensive than using a direct method [12]. Because of these advantages, direct methods are preferred in most commercial structural analysis software.

1.2.3 Direct Solution Methods

1.2.3.1 Background

Direct methods are based on the factorization of the stiffness matrix. They can be classified according to the way of the factorization is performed, for example in LU decomposition, the stiffness matrix is factorized as

$$A = LU = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ L_{2,1} & \ddots & 0 & \vdots \\ \vdots & . & \ddots & 0 \\ L_{n,1} & \cdots & L_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} \\ 0 & \ddots & . & \vdots \\ \vdots & 0 & \ddots & U_{n-1,n} \\ 0 & \cdots & 0 & U_{n,n} \end{bmatrix} \tag{1.3}$$

where U is upper triangular coefficient matrix obtained at the end of forward elimination, and L is lower triangular matrix formed by multipliers that is used during forward elimination. For a symmetric stiffness matrix A, this decomposition becomes

$$A = LL^T = \begin{bmatrix} L_{1,1} & 0 & \cdots & 0 \\ L_{2,1} & \ddots & 0 & \vdots \\ \vdots & . & \ddots & 0 \\ L_{n,1} & \cdots & L_{n,n-1} & L_{n,n} \end{bmatrix} \begin{bmatrix} L_{1,1} & L_{1,2} & \cdots & L_{1,n} \\ 0 & \ddots & . & \vdots \\ \vdots & 0 & \ddots & L_{n-1,n} \\ 0 & \cdots & 0 & L_{n,n} \end{bmatrix} \tag{1.4}$$

which is known as Cholesky's decomposition. Following formulas apply for the entries of L;

$$L_{i,j} = \frac{1}{L_{j,j}}(A_{i,j} - \sum_{k=1}^{j-1} L_{i.k}L_{j.k}), \text{ for i<j} \tag{1.5}$$

and

$$L_{i,i} = \sqrt{A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2} \tag{1.6}$$

where indices for entries of L, $i = 1...n$, and $j = 1...n$.

Here, computation of square root hinders the performance of the procedure. To avoid taking square roots, following alternative can be utilized;

$$A = LDL^T = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ L_{2,1} & \ddots & 0 & \vdots \\ \vdots & . & \ddots & 0 \\ L_{n,1} & \cdots & L_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} D_{1,1} & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & D_{n,n} \end{bmatrix} \begin{bmatrix} 1 & L_{1,2} & \cdots & L_{1,n} \\ 0 & \ddots & . & \vdots \\ \vdots & 0 & \ddots & L_{n-1,n} \\ 0 & \cdots & 0 & 1 \end{bmatrix} \tag{1.7}$$

where D is a diagonal matrix. So, formulations for the entries of D and L are;

$$L_{i,j} = \frac{1}{D_j}(A_{i,j} - \sum_{k=1}^{j-1} L_{i.k} L_{j.k} D_k), \text{ for i<j} \tag{1.8}$$

and

$$D_i = A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2 D_k \tag{1.9}$$

where indices for entries of L, $i = 1...n$, and $j = 1...n$.

For all decomposition methods mentioned above, forward elimination followed by backward substitution completes the solution process for each given right hand side vector.

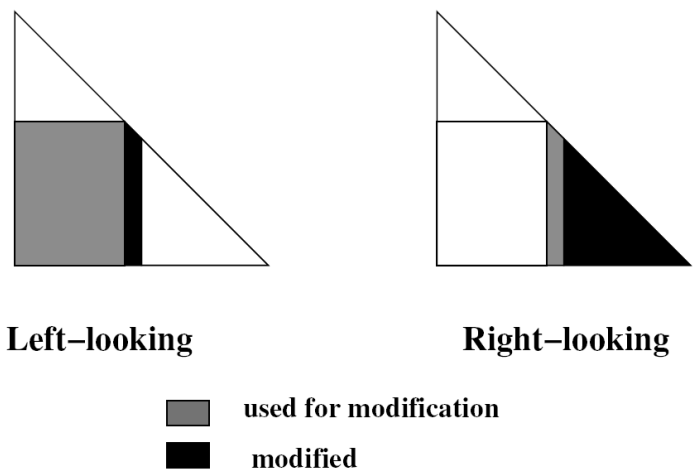


Figure 1.1: Memory access patterns for variants of Cholesky Decomposition

According to the way a matrix is factorized , direct solution methods can be examined into two groups as left-looking and right-looking (Figure 1.1). Both of these computations can be typically expressed in terms of two primitives. First, *cmod(j,k)* to add into column j a multiple of column k and second *cdiv(j)* : divide column j by a scalar.

- Left-looking (or fan-in) algorithms; where updates are performed on each column in turn by all the previous columns that contribute to it, then the pivot is chosen in that column and the multipliers calculated. In other words, the general structure of the left-looking approach is as follows, with the kth term iterating over columns to the left of column j in the matrix.

Algorithm 1.1 Left-Looking Algorithm

```
for j = 1 to n do
    cdiv(j)
    for each k that modifies j do
        cmod(j, k)
```

- Right-looking (or fan-out) algorithm; where, after the calculation of pivot and multipliers, that column is immediately used to update all future columns that it modifies. Namely, the general structure of the right-looking approach is as follows, with the j term iterating over column to the right of column k.

Algorithm 1.2 Right-Looking Algorithm

```
for k = l to n do
    cdiv (k)
    for each j modified by k do
        cmod(j, k)
```

In both cases, j iterates over destination columns and k iterates over source columns. Note also that the *cmod()* operation is performed a number of times per column while the *cdiv()* operation is performed only once. Therefore , the *cmod()* operation dominates the computation time.

The tests over the efficiency of both of the algorithms reveal that left-looking algorithm is more efficient because it enables the usage of contiguous data in memory. However, this difference in performance is negligible when the improvement achieved by the use of blocked algorithms considered [13].

During factorization step, various linear algebra routines like matrix-vector or matrix-matrix operations are executed numerous times [2, 13, 14]. To handle such common operations, BLAS (Basic Linear Algebra Subprograms) are developed and distributed in public domain. BLAS are subdivided into three levels, each of which offers increased scope for exploiting performance. This subdivision corresponds to three different types of basic linear algebra operations:

- Level 1 BLAS : for vector operations, such as $y \leftarrow \alpha x + y$,
- Level 2 BLAS : for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$,

- Level 3 BLAS : for matrix-matrix operations, such as $C \leftarrow \alpha AB + \beta C$.

Here, A, B, and C are matrices, x and y are vectors, and α and β are scalars [10].

The main obstacle to obtaining high performance is the bottleneck in getting data from the main memory to the functional units. Many machines have multiple caches usually organized hierarchically from fastest-smallest to slowest-largest. Therefore, to obtain high performance relative to the peak of the machine, it is necessary to reuse data in the cache as much as possible to amortize the cost of getting it to the cache from main memory. In this context, performance of the routines in BLAS increases with the increase in its level because more contiguous data will be transferred to cache and used repetitively [11]. Therefore, the most suitable and widely used kernels are Level 3 BLAS for $O(n^3)$ operations involving matrices of order n [1, 10, 15, 16]. Likewise, instead of factorizing a single column, factorizing a blocked column is more efficient. For example, Level 2.5 BLAS is designed as the multiplication of a set of vectors by a matrix where the vectors cannot be stored in two-dimensional arrays. In other words, source data can be held in cache and applied to the target columns or blocks of columns of the target data, thus getting a high degree of reuse of data and a performance similar to the Level 3 BLAS [17].

Currently, various solvers, not only utilize methods discussed here but also by slightly modifying these methods, try to enhance computational efficiency on different computational environments like parallel solution environments.

1.2.3.2 Direct Solvers for Dense Matrices

With the great achievement of BLAS to become a *de facto* standard for linear algebra computations, an upper level library, LAPACK, or Linear Algebra PACKage, is built over BLAS routines [11]. This library is a collection of routines for linear system solution, linear least squares problems, and eigenproblems. The associated matrix factorizations (LU, Cholesky, QR, SVD, etc..) are also provided. Dense and banded matrices are handled. High performance is attained by using algorithms that perform most of their work in calls to the BLAS, with an emphasis on matrix-matrix multiplication [18].

Because of their high performance, dense matrix kernels also have a widespread usage in parallel linear algebra. Besides, native parallel dense solver packages such as ScaLAPACK are also provided in public domain. To have a parallel dense solver

library, communication libraries are required in addition to the kernels. BLACS, or Basic Linear Algebra Communication Subroutines provides point-to-point or collective communication subroutines.

ScaLAPACK is a library of high performance linear algebra routines for PC clusters (refer Section 1.2.5). It is based on LAPACK and PBLAS (a set of parallel version of BLAS) which uses BLACS for communication and calls the BLAS. Like LAPACK, the ScaLAPACK routines are based on block partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. In the ScaLAPACK routines, all inter-processor communication occurs within the PBLAS and the BLACS. As it can be seen from Figure 1.2, libraries below the dashed line are serial, in other words, sequential libraries that does not require any communication among computers.

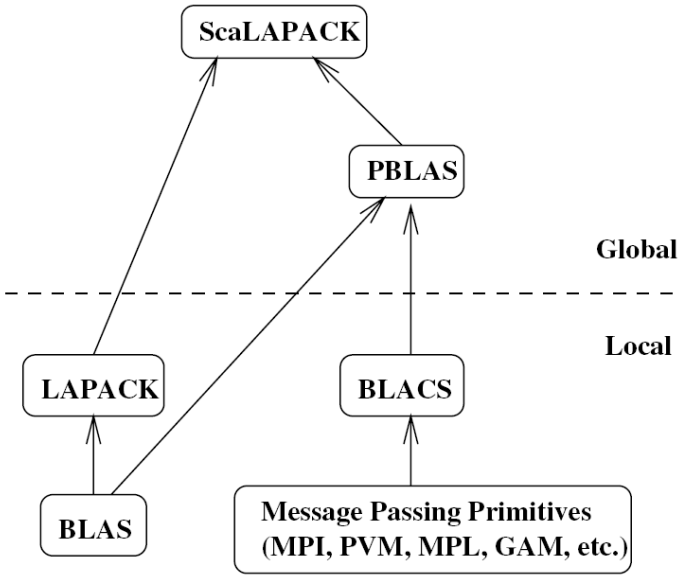


Figure 1.2: ScaLAPACK Software Hierarchy [1]

The goals of ScaLAPACK project are *efficiency*, so that the computationally intensive routines execute as fast as possible; *scalability* as the problem size and number of processors grow; *reliability*, including the return of error bounds; *portability* across machines; *flexibility* so that users may construct new routines from well-designed components; and *ease of use* [1].

Scalability demands that a program be reasonably effective (refer to Appendix A) over a wide range of numbers of processors. The scalability of parallel algorithms over a range of architectures and numbers of processors requires that the granularity of computation be adjustable. To accomplish this, block partitioned algorithms are

provided with adjustable block sizes [19].

The performance of the ScaLAPACK drivers, are dependent to the performance of each computer in the cluster, computational workload assigned to each computer, data distribution type and block size. The ScaLAPACK software assumes that the user's input data has been distributed on a two-dimensional grid of processors according to the block cyclic data distribution (4.4.1.1.1). For a given number of processors, the parameters of this family of data distributions are the shape of the processor grid and the size of the block used to partition and distribute the matrix entries over the processor grid. These parameters affect the number of messages exchanged during the operation, the aggregated volume of data communicated, and the computational load balance [1].

1.2.3.2.1 Cholesky Factorization in LAPACK

Cholesky factorization factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). It is assumed that the lower triangular portion of A is stored in the lower triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A. At the kth step, the $n \times n$ matrices $A(k)$, $L(k)$, and $L^T(k)$ are partitioned, and the system is written as

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix} \tag{1.10}$$

where the block A_{11} is $nb \times nb$, A_{21} is $(n - nb) \times nb$, and A_{22} is $(n - nb) \times (n - nb)$. L_{11} and L_{22} are lower triangular.

The block-partitioned form of Cholesky factorization may be inferred inductively as follows. If it is assumed that L_{11}, the lower triangular Cholesky factor of A_{11}, is known, the block equations can be rearranged as;

$$L_{21} \leftarrow A_{21}(L_{11}^T)^{-1} \tag{1.11}$$

and

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T \tag{1.12}$$

A snapshot of the block Cholesky factorization algorithm in Figure 1.3 shows how the column panel $L^{(k)}$ (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22}

is updated. The factorization can be done by recursively applying the steps outlined above to the *(n - nb)×(n - nb)* matrix $\tilde{A_{22}}$.

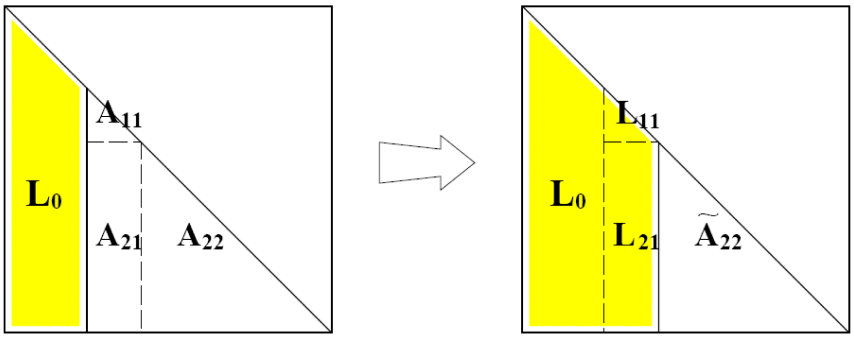


Figure 1.3: A snapshot of block Cholesky factorization[2]

In the right-looking version of the LAPACK routine, the computation of the above steps involves the following operations:

1. **DPOTF2**: Compute the Cholesky factorization of the diagonal block A_{11}.

$$A_{11} \rightarrow L_{11}L_{11}^T \tag{1.13}$$

2. **DTRSM**: Compute the column panel L_{21},

$$L_{21} \leftarrow A_{21}(L_{11}^T)^{-1} \tag{1.14}$$

3. **DSYRK**: Update the rest of the matrix,

$$\tilde{A_{22}} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T \tag{1.15}$$

1.2.3.2.2 Cholesky Factorization in ScaLAPACK

The parallel implementation of the Cholesky factorization in ScaLAPACK proceeds as follows:

1. **PDPOTF2**: The processor P_i, which has the $nb \times nb$ diagonal block A_{11}, performs the Cholesky factorization of A_{11}.

   - P_i performs $A_{11} \rightarrow L_{11}L_{11}^T$, and sets a flag if A_{11} is not positive definite.
   - P_i broadcasts the flag to all other processors so that the computation can be stopped if A_{11} is not positive definite.

2. **PDTRSM**: L_{11} is broadcast columnwise by P_i down all rows in the current column of processors, which computes the column of blocks of L_{21}.

3. **PDSYRK**: the column of blocks L_{21} is broadcast rowwise across all columns of processors and then transposed. Now, processors have their own portions of L_{21} and L_{21}^T. They update their local portions of the matrix A_{22}.

1.2.3.3 Direct Solvers for Sparse Matrices

The resulting system of equations produced by the finite element method are generally sparse and in literature there is plenty of research on direct solution of sparse matrices.

One of the fundamental concepts in sparse matrix factorization is the elimination tree. The elimination tree is defined for any sparse matrix whose sparsity pattern is symmetric. For a sparse matrix of order n, the elimination tree is a tree on n nodes such that node j is the father (or parent) of node i if $entry(i, j)$, $j > i$ is the first entry below the diagonal in column i of the lower triangular factor. Similarly, node i is called as leaf (or descendant) of node j. Each node is connected with an edge.

For example, Sparse Cholesky factorization by columns can be represented by an elimination tree. This can either be a left-looking or a right-looking algorithm. Either way, the dependency between columns which will be updated and columns will be used for update, is determined by the elimination tree. If each node of the tree is associated with a column, a column can only be modified by columns corresponding to nodes that are descendants of the corresponding node in the elimination tree.

Even though they have different solution procedures and approaches in global, most of the popular sparse equation solvers are using dense matrix kernels for the core of computation. This is because of avoiding indirect addressing and trying to manipulate blocked algorithms for high performance purposes [11, 15, 16, 20].

One approach to using higher level BLAS in sparse direct solvers is a generalization of a sparse column factorization. Higher level BLAS can be used if columns with a common sparsity pattern are considered together as a single block or supernode and algorithms are termed column-supernode, supernode-column, and supernode-supernode depending on whether target, source, or both are supernodes. This approach is named as supernodal approach.

Sparse direct methods solve systems of linear equations by factorizing (described in Section 1.2.3) the coefficient matrix A. An ordering method which is interchanging the rows and/or the columns of sparse matrix and can be utilized with an elimination tree in order to minimize both the storage requirement and amount of computations performed. Sparse direct solvers usually have numerous distinct phases [12] that are

summarized below:

1. An ordering phase that determines a factorization order which reduces both storage requirements and the number of floating-point operations required.

2. An analysis phase (which is sometimes referred to as the symbolic factorization step [15]) that the matrix structure is examined in order to determine the amount of storage for factorization and solution phases. Furthermore, elimination tree which governs the calculations during the factorization and subsequent solution phase is created. The tree is also used to schedule the parallel tasks, since the nodes of the tree can be viewed as representing computations and the edges representing transfer of data.

3. A numerical factorization phase that uses the elimination tree to factorize the matrix.

4. A triangular solution phase that performs forward elimination followed by back substitution using the computed factors at previous stages.

According to Dongarra et al. [11], three levels parallelism can be achieved in direct sparse solvers;

- System Level Parallelism; can be exploited by dividing the problem into subproblems that can be solved independently. Then the contributions of each problem form an interconnecting system problem which is quite smaller than the original problem.

- Matrix Level Parallelism; can be achieved by using the sparsity pattern of the coefficient matrix. Usually elimination trees are used for determining the independent computations.

- Sub-matrix Level Parallelism; can be achieved by performing series of dense matrix operations over sub-matrices. PBLAS and ScaLAPACK (refer to Section 1.2.3.2) can be used for this purpose.

In literature, there are numerous research on parallel direct solvers for sparse systems because of their robustness and guaranteed success over different types of sparse matrices resulting from different types of problems. Although they have slightly different strategies and performances, all of the modern sparse solvers are trying to take the advantage of dependency among the sparse data to be processed and utilize blocked dense

solvers in lower levels of computation, allowing the exploitation of higher-level BLAS. Some state-of-art parallel sparse direct solvers will be discussed next.

SuperLU DIST, as the name implies, the distributed (parallel) version of SuperLU algorithm which uses supernodal approach for factorization. SuperLU DIST, utilizes a directed acyclic graph (elimination tree) to reduce the memory requirement and a static task scheduling according to the elimination tree. Right-looking formulation (described in 1.2.3) is used to carry out the elimination updates. Immediately after factorization of a block of columns (namely, supernode) corresponding to a node in the tree, data are sent to update blocks corresponding to ancestors in the tree [15, 21, 22].

MUMPS utilizes a multifrontal approach where the Gaussian eliminations are carried out on dense frontal matrices corresponding to children nodes in the elimination tree. The resulting Schur complements (condensed matrices) are thereupon sent for assembly to the parent nodes. Thus, the elimination structure utilized by MUMPS is called as assembly tree. MUMPS exploits parallelism arising from the sparsity in the matrix and parallelism available for dense matrices. Additionally, large computational tasks are divided into smaller subtasks to enhance parallelism. MUMPS uses a distributed dynamic scheduling technique that allows numerical pivoting and the migration of computational tasks to lightly loaded processors. Asynchronous communication is used to overlap communication with computation. By default, MUMPS automatically chooses the ordering algorithm depending on the packages installed, the size of the matrix, and the number of processors available [15, 20].

These two solvers, MUMPS and SuperLU Dist are compared by various studies [15, 22]. According to the results of these studies, the block sizes occurring during factorization within SuperLU Dist were smaller than those utilized within MUMPS, giving less efficient use of the Level 3 BLAS kernels and hence generally produced slower factorization and solution speeds. On the other hand, it was also observed that efficiency of MUMPS dropped because of the memory problems and increasing communication overhead when the number of processors increased.

In the study conducted by Guermouche et al. [23] problems of multifrontal methods were discussed. They compared the effect of five different reordering algorithms, AMD [24], AMF [25], PORD [26], METIS [3], and SCOTCH [27], on the shape of the corresponding assembly trees, hence, their effects on the memory usage. The METIS [3] and SCOTCH [27] libraries produced wide, well balanced trees where the others produced very deep unbalanced trees with a large number of nodes. In terms of memory usage,

deep unbalanced trees were found to be better than the wide ones. They concluded that, however, for parallel cases, the computational scheduling had to be considered since it also had a significant effect on the memory requirement. In conclusion, there are two main problems in multifrontal methods; they may require large memory space for in-core storage which is not always possible. Secondly, their parallel efficiency depends on how the elimination trees are constructed.

TAUCS is another state-of-art parallel direct solver which is based on multifrontal supernodal sparse Cholesky factorization. The multifrontal supernodal method factors the matrix using recursion on the assembly tree. Each node in the tree is associated with a set of columns of the Cholesky factor L (unknowns in the linear system). The method works by factorizing the columns associated with all the columns associated with proper descendants of a parent node, then updating the coefficients of the unknowns associated with this node, and factorizing the columns of it. The updates and the factorization are performed using calls to the dense Level 3 BLAS kernels. TAUCS is currently used in Mathematica [14].

1.2.4 Substructuring

Popular parallel direct sparse solvers like MUMPS and SuperLU are not explicitly performing a parallelization on system level [15, 22]. Their main concern is solving a linear system of equations as fast as possible. However, in finite element analysis, pre-processing and post-processing stages may take considerable amount of time especially with the enhanced speed of sparse solvers. Pre-processing stage involves the formation of element stiffness matrices and assembly of the structural stiffness matrix and load vectors. Similarly, after the solution of the system, results are used in post-processing stage in which element forces and stresses are computed. Within this framework, substructuring can be used to extend the parallelism to cover pre-processing and post-processing computations of a direct solution of finite element analysis problem.

Another advantage of substructuring is that it can reduce the number of equations to manageable size by dividing a structure into non-overlapping substructures. The element stiffness matrices of each element that lies on each substructure is assembled to generate the substructure equations. By treating each substructure as a super-element with many internal and external (interface) nodes, and using static condensation (refer to Section 2.3.2), the equations of substructure are reduced to a form involving only the interface nodes of that particular substructure. The reduced substructure equations can

then be assembled to obtain the overall system equations involving only the interface unknowns of the various substructures. The number of these system equations is much less compared to the total number of unknowns. The solution of the system equations gives values of the interface unknowns of each substructure. The known interface nodal values can then be used as prescribed interface conditions for each substructure to solve for the respective internal nodal unknowns.

In substructure based parallel solution methods (explained in Section 1.2.4.3), the performance of the solution is very sensitive to the way the structure is partitioned into substructures. The optimum substructuring [6] for a particular structure should;

- guarantee that the parallel solution time is less than the serial solution time
- balance the workloads for each processor
- minimize the communication among the processors

1.2.4.1 Partitioning

All the requirements mentioned in Section 1.2.4, for an optimum substructuring to analyze a structure, increase the complexity of the substructuring problem. Hence, several partitioning approaches are developed. These methods can be examined in two groups; static partitioning and dynamic partitioning.

Static partitioning methods are preferred when the workload and communication requirements are known before the actual computation initiates. Kurç [6] classified these methods in four groups as geometric, topological, graph based, and hybrid methods. Geometric based methods divide the domain by using the geometric properties of each object, i.e. nodal coordinates, elements etc. Recursive Coordinate Bisection Method [28], Unbalanced Recursive Bisection Method [29], and Recursive Inertial Bisection [30] can be given as examples for this group. Second group is topological based methods in which the connectivity information among the objects are used for partitioning. Greedy Algorithm [31], Bandwidth Reduction Approach [32], and Octree Partitioning [33] are examples to this group. Third group is graph based algorithms. In such methods, graph models of a computation are prepared and all the partitioning computations are performed on these models. Examples for this group are Recursive Spectral Bisection Method [34] and Multilevel Approaches [35].

Dynamic partitioning methods can be used in cases that arise where it is either impossible to calculate the computational loads initially or the computational requirement

varies during solution in an unpredictable way. One way to handle this problem is using the new information about the computational loads to repartition the mesh (Scratch-Remap Algorithms [36,37]). However, it should be guaranteed that the new partition is similar to the previous one, otherwise, huge amounts of data should be transferred among processors. The other option is transferring nodes among the processors in order to balance the load by shifting the interfaces (Diffusion Algorithms [38]). In this case, interface shifting might cause considerable increase in the interface size which will increase the communication volume. Therefore, having a transfer algorithm that will balance the workload while keeping the edge-cut as small as possible is very important for such methods [6].

1.2.4.2 Workload Balancing

Workload imbalance is one of the most important phenomenon that reduces the efficiency of any parallel solution algorithm. When direct solution methods are utilized, the number of operations required to solve a linear system of equations can be predicted before the solution initiates. Hence, in literature there are various methods that attempts to balance the computational loads of processors that participated in solution.

Yang and Hsieh [39] proposed an iterative partition optimization method for direct substructuring. After finding the initial partitions, the number of arithmetic operations required for condensation is computed using the symbolic factorization. The weights of the elements within a substructure are adjusted according to the operation counts found for each substructure. Later the partitions are modified by using the partitioning packages JOSTLE [40] and METIS [3]. While METIS [3] restarts the partitioning from the scratch, JOSTLE [40] has the feature to adjust the partitions from the previous iterations. Iteratively refining the partitions using JOSTLE [40] generally provides balanced partitions with less iteration. Refining the partitions by moving small number of elements between the partitions shows similarities to the dynamic partitioning that tries to minimize the number of objects moved between partitions.

Kurç and Will [41] proposed a workload balancing scheme for the condensation of the substructures. METIS [3] partitioning library is utilized for the initial partitioning of the nodal graph representation of a structure. Later the node weights of partitions are adjusted according to the estimated operation counts. The PARMETIS [42] library is used for repartitioning according to the adjusted node weights. The diffusion and scratch-remap repartitioning algorithms are investigated. It was concluded that scratch-

remap produced computationally more balanced substructures. Moreover, the number of interface equations was smaller with scratch-remap algorithm compared to diffusion algorithm. They also stated that time spent during workload balancing iterations was insignificant compared to the improvements obtained in the condensation times. Test results indicated that workload balancing iterations reduce the total solution timings considerably.

1.2.4.3 Substructure based Solution Methods

Substructuring offers several advantages such as enabling the parallelization of every step of the solution, from formation of element stiffness matrices to the computation of element forces and stresses and minimizing the communication by requiring data transfer only during the solution of interface equations. These features make them very suitable for PC Clusters that have relatively low communication speed with respect to their computation power. Besides, with the use of direct solvers, they may enhance the solution of systems that have multiple loading conditions.

Duff and Scott [43] applied substructuring methods to multi-fronts scheme. In such a case, instead of creating elimination trees, the underlying domain was partitioned into subdomains and frontal decompositions were performed on each domain separately. In conclusion, they stated that domain partitioning could reduce operation counts, factorization and solution times. Besides, they observed that costs of communication and interface problem solution did not dominate the overall solution time and they obtained reasonable speed-ups.

Farhat et al. [44] implemented substructuring for parallel finite element solution. Their method initiated by partitioning the structure into subdomains. For each subdomain, the stiffness matrix and force vectors were formed by first numbering the internal degrees of freedom and then the interface degrees of freedom. The internal equations were transferred to the interfaces by static condensation. For the solution of interface problem, the row-wise LDL^T decomposition was utilized. They observed that efficiency of the method is dropping with the increase in the number of processors. They, however, stated that with the increase in problem size the efficiency drop could be compensated.

Bjørstad et al. [45] implemented a direct solution algorithm, based on processing substructures in parallel. Proposed algorithm was to divide substructures into smaller substructures in a multilevel fashion. At any given level in this procedure, the unknowns were divided into two disjoint sets, the internal variables and the external (interface)

variables. Before the algorithm proceeds to the next level, all internal variables were eliminated by static condensation. At the next level the retained variables from the previous level were again split into two sets and this process repeated until one reached the highest level where all remaining variables in the problem would be in the internal set. An elimination tree was constructed before the computation and according to this tree, pool of tasks where information about all the tasks including execution schedule and time estimates were formed. With this approach they ensured a symmetric and well load balanced substructures.

In another study, Baugh and Sharma [46] implemented the domain decomposition method to solve linear equations on a network of workstations. They compared five different algorithms based on direct, iterative, and hybrid methods. In the direct approach, the partitions were first condensed with a direct static condensation method and a direct solution was performed at the interface. In the iterative approach, they solved the system globally by using two different versions of the conjugate gradient method. The hybrid approach used direct condensation and parallel and sequential versions of conjugate gradient method for the interface problem. The test results showed that the iterative solution methods were outperformed by the direct solution methods for the solution of a rectangular membrane problem on a workstation environment that was connected with an ordinary LAN.

Fulton and Su [47] implemented the substructuring method on a shared memory parallel computer. They used active column storage scheme to store the substructure level stiffness matrix. During the condensation, the internal equations were first numbered and then the interface equations. The interface stiffness matrix was kept in the shared memory and the contribution of each substructure was assembled to the interface stiffness matrix according to the correct location determined during the renumbering phase. In order to balance the various computational loads for the condensation phase of each substructure, more processors were assigned to the substructures which were estimated to require more computation. The proposed approach performed much better than the parallel global solution algorithm.

The other paper by Synn and Fulton [48] searched the answers for the following issues: direct versus iterative solution, the optimum number of processors for the parallel matrix decomposition, workload balancing, and which solution type to be utilized for a particular problem. They recommended the direct solution methods even though iterative methods were more scalable. The load balancing during condensation step

was provided by assigning more processors to the subdomains estimated to have larger number of equations and bandwidths. Moreover, they derived operation count equations to estimate the optimum number of processors and to choose whether to use the global solution instead of substructure based solution.

An object-oriented database structure was proposed by Hsieh et al. [49] that could be used in parallel finite element codes for structural engineering applications. They preferred substructuring approach with direct solvers in their code which utilized the parallel matrix library developed by Modak et al. [50]. In this library, the linear solution algorithm was based on active column matrices that utilized Cholesky decomposition method. Within the view of test cases, small speed-ups were obtained for the factorization phase during the interface solution. Moreover, the forward and back substitution times remained constant as the number of processors increased.

Escaig et al. [51] presented a multilevel domain decomposition method with a direct solver for the interface problem. They first partitioned the structure in such a way that the number of subdomains was larger than the number of processors. During the parallel solution, the subdomains were condensed by the first available processor. This way it was possible to balance the workload among the processors. However, as the number of subdomains increased, the size of the interface problem also increased. They tested their algorithm both in shared and distributed memory architectures. Although they obtained reasonable results for shared memory architectures, the performance dropped as the number of processors increased for distributed architectures.

An analytical study performed by Nikishkov et al. [52] examined the parallel performance of the domain decomposition method with LDU based condensation and solution algorithms. They first calculated the number of operations and the communication volumes and estimated the solution times of each algorithm for a square domain problem. Then, they compared the time estimations with the actual values. The predicted values mostly agreed with the actual ones ($<5\%$) and good parallel efficiency 95% with 6 processors, 85% with 8 processors, was obtained.

Kurç [6] proposed a substructure based parallel solution framework for solving linear systems with multiple loading conditions. To overcome the work balancing problem and enhance the performance of the parallel solution, he suggested a framework that consists of two steps where the first step was preparing data for the parallel solution that involves partitioning, workload balancing, and equation numbering. The second step was fully parallel solution of substructured finite element model with direct solvers.

During the data preparation step, best possible partitioning for parallel solution was obtained by iteratively transferring nodes from the substructures having slower condensation time estimations to the faster ones. After data preparation step, each processor assembles its substructure's stiffness matrix and condenses it to the interfaces by using a profile solver. The interface equations were then solved by a variable band solver. In conclusion, he stated that workload balancing was able to decrease not only the local factorization time but also the local forward and back substitution times. Comparative tests were presented for various numbers of computers to demonstrate the performance and efficiency of the overall solution framework on PC clusters that were connected with ordinary ethernet.

Kurç and Will [53] extended the framework to a heterogeneous PC cluster environment. At the beginning of the data preparation step, a cluster recognition step was added to obtain processor speeds compared to each other. This tuning was tested on example problems and it was observed that balancing the substructures according to obtained solution time-operation count ratios reduced the solution time considerably. Besides, they enhanced the condensation step that consumes the most of solution time by utilizing a sparse solver rather than an active column one. This enhancement revealed that interface system solution and communication at that phase was a governing factor over the solution time as the number of processors increased.

1.2.5 PC Clusters and Communication Interfaces

PC Clusters are groups of computers which have their own processors and memories and are simply connected to each other with routers, hubs, or switches. Data is transferred among the computer by this network. This is one of the cheapest ways to obtain a parallel computation environment and they can be upgraded and extended for a very low price. However, it is difficult to obtain a sufficient performance on clusters since the communication speed among the computers are relatively slow [6].

PC Clusters can be classified as homogeneous and heterogeneous clusters. Homogeneous clusters are the ones that are composed of computers which have exactly the same hardware and configuration. Thus, they have similar computation characteristics. However, in heterogeneous clusters, computers may have different hardware. Parallel solution algorithms vary depending on the PC cluster type on which they are running.

Communication interfaces are the tools for forming a similar language ground among the computers with different structures and operating systems. Walker proposed that

MPI [54] as the de facto standard for communication among processors. It is portable and easy to use. MPI includes point-to-point and collective communication routines like MPI_Send or MPI_Receive either in blocking or non-blocking forms [55]. Blocking means participating applications are halted until the job requested by MPI command is finalized. Thus, non-blocking means a possible overlap of message transmittal with computation or the overlap of multiple message transmittals with one another [56]. Different implementations of MPI like MPICH [57], LAM [58] and Open MPI [59] are available in public domain. Among these MPI implementations, MPICH was found to be performed slightly better than others [60].

1.3 Objectives and Scope

The main purpose of this study is to develop an efficient parallel solution algorithm for the linear solution of large structures on PC Clusters. This study is based on the concepts suggested by Kurç [6, 53, 61, 62]. The ultimate goal is to show that parallel algorithms can be used as practical tools in design offices by utilizing the existing system without the need for buying any additional hardware.

Hence, the research objectives are as follows;

- Inserting the stiffness assembly time parameter into workload balancing step, to enhance workload balancing,
- Improving the solution of interface system by using dense solvers,
- Coding a mapping and communication scheme for the efficient assembly of interface system,
- Testing the performance of the framework on actual civil engineering problems,

The target environment is the homogeneous PC Clusters connected with ordinary hubs. For the communications among the computers, MPICH2 [63] is used. BLAS and ScaLAPACK [2] is the main interest area for the numerical computations.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 presents the main topics of solution framework [6,53,61,62] with the improvements implemented by this study and

the mathematical background of the solution method. Then, in Chapter 3, preparation of substructures by partitioning and workload balancing are discussed in detail and some test results are presented. Chapter 4 is devoted to the parallel solution. In this chapter, condensation and interface system solution is discussed in detail and performance of these solution algorithms is investigated. Chapter 5, is composed of the case studies by using the framework presented. Chapter 6, is the final chapter which summarizes the efficiency and future plans of this study. Two appendices follows the final chapter. The first one presents a general idea about how to assess the performance of a parallel algorithm and the second one introduces the structural models utilized for the case studies.

CHAPTER 2

AN OVERVIEW OF THE PARALLEL SOLUTION FRAMEWORK

2.1 Introduction

This study focuses on improving the performance of a substructure based parallel solution framework that utilizes direct solvers for the linear solution of large structural models on PC Clusters. The cluster can be composed of identical computers. In this chapter, general algorithm and mathematical background for this method is presented.

2.2 Algorithm

One of the main challenges of substructure based parallel solution methods is to find a partitioning where the substructures have balanced condensation times. Otherwise, the parallel solution is governed by the substructure having the slowest condensation time. In order to overcome such problems and thus enhance performance of the parallel solution, a data preparation step is added to the solution framework that is responsible from preparing data for the parallel solution which involves partitioning, workload balancing, and equation numbering.

The first step of data preparation is partitioning the structure into substructures where the number of substructures is equal to the number of processors. After creating the initial substructures, the workload balancing step is initiated where the estimated imbalance of the substructures is adjusted by iteratively transferring nodes from the slower substructures to the faster ones. The imbalance was computed by using the total time estimation for the assembly and the condensation of each substructure. All the iterations were performed in parallel to speed-up the workload balancing step.

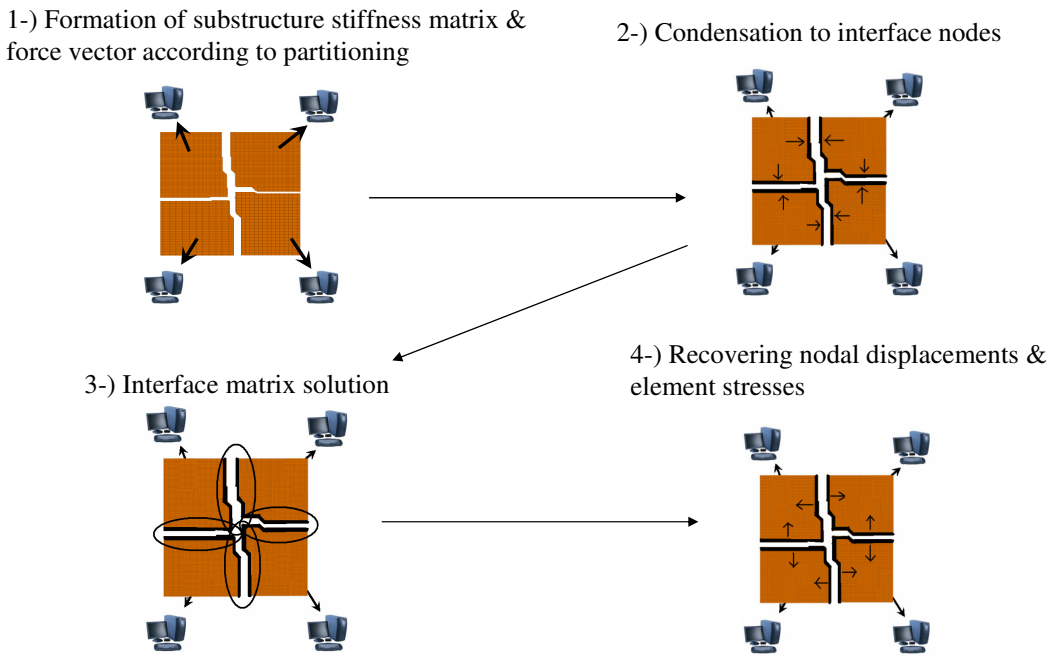


Figure 2.1: Parallel Solution Flow Chart Demonstration

Once the final substructures are created, the solution phase is initiated (Figure 2.1). Each processor assembles its substructure's stiffness matrix and condenses it to the interfaces by using a sparse matrix solver. Then, contributions of each substructure is assembled on the interface system. The interface equations are then solved by parallel dense and banded solver routines of ScaLAPACK library [2]. After computing the interface unknowns, each processor calculates the internal displacements and element stresses or forces of the substructures assigned to them.

2.3 Mathematical Background

The solution approach utilized throughout this study can be summarized as the solution of the condensed structural system of each substructure. After the structure is partitioned into substructures, stiffness effects of each substructure is condensed to interface nodes. Stiffness contributions from all substructures, are gathered into interface nodes and form an interface system. Solution of this interface system results in displacements of the interface nodes. Therefore, by using the results of the interface system solution, the results for the whole system can be obtained.

The condensation part is one of the main steps of the parallel solution. Thus, a general definition for condensation and then the mathematical background for substructure based parallel solution method will be presented.

2.3.1 Condensation

The term condensation refers to the contraction in size of a system of equations by reflecting the contributions or effects of some preselected degrees of freedoms to the rest of the degrees of freedom [64]. Suppose that the degrees of freedom of an arbitrary substructure is numbered starting from the internal dofs and then number the interface nodes. Expected stiffness matrix pattern with skyline storage scheme will be similar to Figure 2.2.

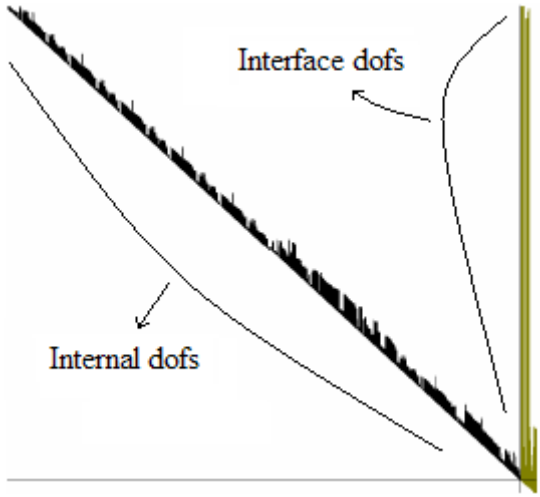


Figure 2.2: An Example Profile of a Stiffness Matrix Assembled for Condensation

The stiffness matrix of a substructure can be represented by a partitioned form of a linear system as follows;

$$\begin{bmatrix} K_{cc} & K_{ci} \\ K_{ic} & K_{ii} \end{bmatrix} \begin{Bmatrix} \Delta_c \\ \Delta_i \end{Bmatrix} = \begin{Bmatrix} P_c \\ P_i \end{Bmatrix} \tag{2.1}$$

where c index represents internal dofs and i index represents interface dofs. In Equation 2.1 and the following equations, K designates the stiffness, P designates load applied and Δ designates the displacement of nodes.

After the condensation, this system is reduced to

$$\left[\hat{K_{ii}}\right] \{\Delta_i\} = \left\{\hat{P_i}\right\} \tag{2.2}$$

To reduce Equation 2.1 into Equation 2.2, first row of Equation 2.1 is expanded and

solved for $\{\Delta_c\}$. Thus

$$\{\Delta_c\} = -\left[K_{cc}\right]^{-1}\left[K_{ci}\right]\{\Delta_i\} + \left[K_{cc}\right]^{-1}\{P_c\} \tag{2.3}$$

Substituting this value of $\{\Delta_c\}$in the expanded second row of Equation 2.1 yields

$$-\left[K_{ic}\right]\left[K_{cc}\right]^{-1}\left[K_{ci}\right]\{\Delta_i\} + \left[K_{ii}\right]\{\Delta_i\} = \{P_i\} - \left[K_{ic}\right]\left[K_{cc}\right]^{-1}\{P_c\} \tag{2.4}$$

Letting

$$\left[K_{ii}\right] - \left[K_{ic}\right]\left[K_{cc}\right]^{-1}\left[K_{ci}\right] = \left[\hat{K}_{ii}\right] \tag{2.5}$$

and

$$\{P_i\} - \left[K_{ic}\right]\left[K_{cc}\right]^{-1}\{P_c\} = \left\{\hat{P}_i\right\} \tag{2.6}$$

Equation 2.4 becomes identical to Equation 2.2. Therefore the stiffness effects of internal dofs of the substructure is reflected on the interface dofs and internal dofs are eliminated. Also, it should be clear that the "eliminated" dofs are not discarded. They are expressed as functions of the corresponding forces, the remaining dofs and the coefficients of the equations, with the result substituted in the original equations.

The concepts of condensation have already been used although they were not identified by that term. In literature, the disciplines other than structural engineering use this method with the name of Schur Complement Matrix [65]. Suppose A, B, C, D are respectively $p \times p$, $p \times q$, $q \times p$ and $q \times q$ matrices, and D is invertible. Let

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{2.7}$$

so that M is a $(p+q)\times(p+q)$ matrix. Then the Schur complement of the block D of the matrix M is the $p \times p$ matrix $A - BD^{-1}C$ [66] which is equivalent to $\hat{K}_{ii}$ of Equation2.5.

2.3.2 Substructure based solution

Consider a structural model that is partitioned into two, Substructure A and Substructure B as shown in Figure 2.3. Degrees of freedom of the whole structure can be divided into three sets as [64];

- a: internal dofs of Substructure A
- b: internal dofs of Substructure B

- i: interface dofs between Substructure A and B

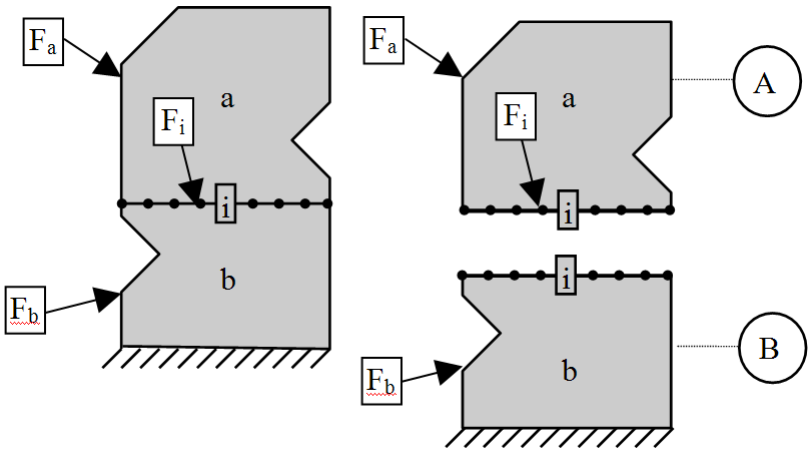


Figure 2.3: Internal and Interface dofs of Substructures

Therefore, stiffness matrices of each substructure can be written as follows;

$$K_A = \begin{bmatrix} K_{aa} & K_{ai} \\ K_{ia} & K_{ii}^A \end{bmatrix} \qquad K_B = \begin{bmatrix} K_{ii}^B & K_{ib} \\ K_{bi} & K_{bb} \end{bmatrix} \tag{2.8}$$

and similarly load vectors can be written as follows;

$$F_A = \begin{bmatrix} F_a \\ F_i^A \end{bmatrix} \qquad F_B = \begin{bmatrix} F_i^B \\ F_b \end{bmatrix} \tag{2.9}$$

When the stiffness matrices of each substructure is assembled, the structural stiffness matrix will become:

$$\begin{bmatrix} K_{aa} & K_{ai} & 0 \\ K_{ia} & K_{ii} & K_{ib} \\ 0 & K_{bi} & K_{bb} \end{bmatrix} \begin{Bmatrix} \Delta_a \\ \Delta_i \\ \Delta_b \end{Bmatrix} = \begin{Bmatrix} F_a \\ F_i \\ F_b \end{Bmatrix} \tag{2.10}$$

where

$$K_{ii} = K_{ii}^A + K_{ii}^B \tag{2.11}$$

and

$$F_i = F_i^A + F_i^B \tag{2.12}$$

and $\triangle$ represents corresponding displacement and F represents the corresponding load vector.

When static condensation is applied to internal dofs of substructure B, that means they will be eliminated. On the other hand, rest of the dofs which can be demonstrated by $a{+}i$ will remain. So, stiffness matrix and load vector are partitioned in the following way:

$$\left[\begin{array}{cc|c} K_{aa} & K_{ai} & 0 \\ K_{ia} & K_{ii} & K_{ib} \\ \hline 0 & K_{bi} & K_{bb} \end{array}\right] = \left[\begin{array}{c|c} K_{\alpha\alpha} & K_{\alpha\beta} \\ \hline K_{\beta\alpha} & K_{\beta\beta} \end{array}\right] \qquad \left\{\begin{array}{c} F_a \\ F_i \\ \hline F_b \end{array}\right\} = \left\{\begin{array}{c} F_\alpha \\ \hline F_\beta \end{array}\right\} \tag{2.13}$$

where remaining dofs and eliminated dofs are indicated by α and β, respectively. Applying the expressions 2.5 and 2.6 to partitioned Equation 2.13 :

$$\left[\hat{K}_{(a+i)}\right] = \begin{bmatrix} K_{aa} & K_{ai} \\ K_{ia} & K_{ii} \end{bmatrix} - \begin{bmatrix} 0 \\ K_{ib} \end{bmatrix} [K_{bb}]^{-1} \begin{bmatrix} 0 & K_{bi} \end{bmatrix} \tag{2.14}$$

$$\left\{\hat{F}_{(a+i)}\right\} = \begin{Bmatrix} F_a \\ F_i \end{Bmatrix} - \begin{bmatrix} 0 \\ K_{ib} \end{bmatrix} [K_{bb}]^{-1} \{F_b\} \tag{2.15}$$

that result in:

$$\left[\hat{K}_{(a+i)}\right] = \begin{bmatrix} K_{aa} & K_{ai} \\ K_{ia} & K_{ii} - K_{ib}K_{bb}^{-1}K_{bi} \end{bmatrix} \tag{2.16}$$

$$\left\{\hat{F}_{(a+i)}\right\} = \begin{Bmatrix} F_a \\ F_i - K_{ib}K_{bb}^{-1}F_b \end{Bmatrix} \tag{2.17}$$

where $\hat{K}_{(a+i)}$ and $\hat{F}_{(a+i)}$ respectively represent the stiffness matrix and load vector of equivalent condensed system after the condensation of internal dofs of substructure B. Decomposing each matrix as sum of two terms:

$$\left[\hat{K}_{(a+i)}\right] = \begin{bmatrix} K_{aa} & K_{ai} \\ K_{ia} & K_{ii}^{A} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & K_{ii}^{B} - K_{ib}K_{bb}^{-1}K_{bi} \end{bmatrix} = K_A + \hat{K_{ii}} \tag{2.18}$$

$$\left\{\hat{F}_{(a+i)}\right\} = \begin{Bmatrix} F_a \\ 0 \end{Bmatrix} + \begin{Bmatrix} 0 \\ F_i - K_{ib}K_{bb}^{-1}F_b \end{Bmatrix} = \begin{Bmatrix} F_a \\ 0 \end{Bmatrix} + \left\{\hat{F}_i\right\} \tag{2.19}$$

The reduced system of equations to dofs set (a+i) has the following form:

$$\left(K_A + \hat{K}_{ii}\right) \left\{ \begin{array}{c} \Delta_a \\ \Delta_i \end{array} \right\} = \left\{ \begin{array}{c} F_a \\ 0 \end{array} \right\} + \hat{F}_i \tag{2.20}$$

$$\left[\hat{K}_{(a+i)}\right] \left\{ \begin{array}{c} \Delta_a \\ \Delta_i \end{array} \right\} = \left\{\hat{F}_{(a+i)}\right\} \tag{2.21}$$

Equivalent stiffness matrix is composed of the stiffness matrix of substructure A and the condensed stiffness matrix of substructure B to the interface dof (set i) expanded (completed with zeros) to *(a+i)* size. In the same way, equivalent load vector is the applied load to internal dofs of substructure A plus the condensed load vector of substructure B to set i, and expanded again to *(a+i)* size. Similarly, by applying condensation to internal dofs of substructure A, whole system can be reduced to an equivalent interface system of $\hat{K}_{ii}$. Hence, results of this interface system solution can be used to recover the whole system solution.

As a summary, condensation of a substructure reflects the static behaviour of a substructure at its interface dofs. During condensation, the information about other substructures are not required, thus, there is no need for data transfer among processors if each substructure is assigned to a particular processor for condensation. This attribute of substructure based solution reduces the amount of communication by requiring data transfer during interface solution only.

CHAPTER 3

DATA PREPARATION

3.1 Introduction

In the substructure based solution approach, structure is partitioned into substructures and each computer assembles its substructure's stiffness matrix and force vectors and then condenses internal nodes to the interface nodes. The condensation step is followed by the interface solution. The interface solution, however, can not initiate until the condensation of all substructures are finalized. In other words, the time spent during condensation is governed by the substructure that requires the most computation. Thus, any imbalance among the condensation times of each substructure reduces the efficiency of the parallel solution because of having idle processors. Hence, the partitioning of the structure into substructures is vital for such methods.

This chapter is dedicated to data preparation step of a substructure based parallel solution framework that can be used for solving large linear systems on PC clusters. Data preparation step mainly aims to balance the workload for assembly and condensation of each substructure. The method iteratively searches for more balanced substructure workloads by modifying them according to their estimated stiffness matrix assembly and condensation times (workload balancing). Moreover, all the computations during the workload balancing iterations are performed in parallel. This way, the time consumed during the workload balancing step is decreased and the algorithm becomes more suitable for large linear static problems.

3.2 Overview

The first step of a substructure based parallel solution method is partitioning the structure into a number of substructures. Automatic partitioning algorithms are generally

utilized for this purpose. The goal of such partitioning algorithms is to balance the computational workloads of each processor while keeping the size of the substructure interfaces as low as possible. When the workloads of processors are balanced, the processors would be more efficiently utilized. In other words, none of the processors will stay idle while waiting for other processors to finalize their computations. When the size of the substructure interfaces is low, less time will be spent for data transfers among substructures. Thus, the ultimate goal of partitioning is actually to decrease the overall computation time.

Currently, there are mainly two different partitioning approaches, static and dynamic. Static partitioning algorithms are mostly utilized in problems where the workload is computable before the solution and remains unchanged during the solution. For such problems, the computational workload is usually represented as a single integer value assigned to the nodes or the elements of a structure. Hence, once the sum of the weight values of each partition is balanced, it is assumed that the computational workloads for each processor are also balanced.

The dynamic partitioning algorithms, on the other hand, are developed for problems in which the computational loads can not be known prior to partitioning or the computational loads of processors change during the solution. Dynamic partitioning algorithms mainly modify the substructures according to the new computational loads in such a way that the loads are balanced, the differences between the previous and newly formed substructures are minimized, and the interface size of the new partitions has not significantly increased when compared with the interface size of the previous substructures.

In the literature, various partitioning approaches exist [67]. The basic goal of many of the partitioning approaches is to minimize the communication among processors while keeping a balanced number of elements or nodes in each partition. This goal can be achieved if the computational cost can be represented by a single weight value assigned to a node or an element. However, when a direct condensation method is used, such weight definitions are insufficient to provide a balanced distribution of the computational load [68]. There are other variables that affect the condensation time such as equation numbering, the non-zero term pattern of the stiffness matrix, and the number of the internal and interface equations. Moreover, the variables that affect the condensation time depend on the way in which the structure is partitioned. In other words, the computational load of each substructure can only be estimated after partitioning.

Secondly, the non-zero term pattern of the stiffness matrix depends on how the equations are numbered. Such algorithms are based on heuristic approaches. Therefore, it is very difficult to predict the effect of any partition changes on the equation numbering and hence on the condensation time. Thus, it is rather complicated to partition a structure while balancing the workload for direct condensation. In conclusion, utilizing only static partitioning algorithms are not enough for this study. Therefore, static and dynamic algorithms are used together.

3.2.1 Partitioning Libraries

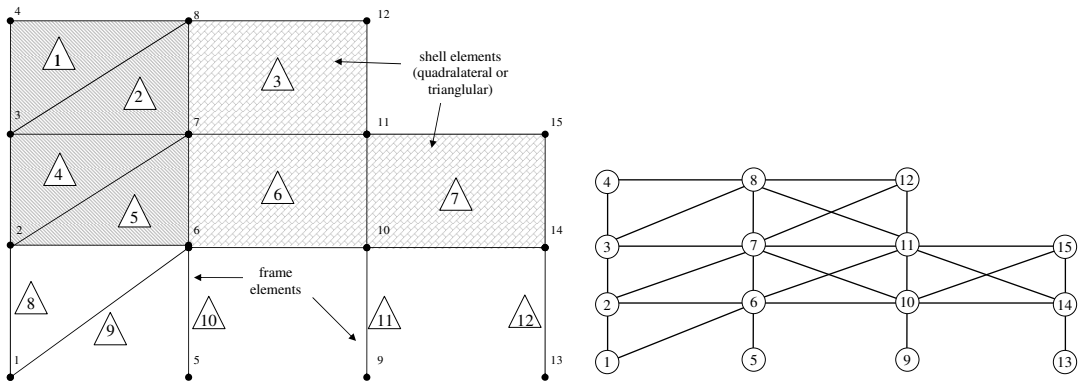
In this study, METIS library [3] is utilized for the initial partitioning of substructures prior to the parallel solution. PARMETIS library [42] is employed as dynamic partitioning utility to repartition the structure in order to balance the total times of assembly and condensation of each substructure.

Both of these libraries; METIS [3] and PARMETIS [42], are graph partitioning methods. In other words, they work with the graph representation of a structure which describes the structure in terms of vertices and edges as shown in Figure 3.1b. Each vertex is actually a solution point and its weight shows the computational weight of that point. An edge is used to define interactions between the vertices. Therefore, a graph partitioning algorithm attempts to keep the vertex weights balanced in each partition while keeping the edges at the domain interfaces as small as possible. One approach to accomplish this is to create a mathematical definition of the partitioning problem and attempt to solve it. In the literature, this mathematical definition is an NP-complete problem which has a computable solution. The exact solution, however, could be very expensive for some types of the analysis methods since the exact solution requires the computation of the second smallest eigenvalue and corresponding eigenvector of the system [69].

The other approach is to target a reasonably good solution instead of the best one with the application of various heuristic methods. The multilevel scheme is a commonly used approach where the size of the graph is reduced and partitioning is performed on a relatively smaller graph. Although the partition quality decreases due to coarsening, the partitioning time drops considerably. Both METIS [3] and PARMETIS [42] libraries utilizes multilevel graph partitioning approaches.

3.2.1.1 METIS

METIS [3] is a software package developed for partitioning large irregular graphs. It utilizes a multilevel approach to speed-up the partitioning process and allows single or multiple vertex and edge weight definitions. METIS [3] works with the graph representation of a structure and accepts any kind of graph, like nodal or dual graph. An example of a nodal graph for an arbitrary structure (Figure 3.1) is given in Figure 3.1b. In the nodal graph, each node corresponds to a vertex in the graph. The vertices are joined with an edge if the corresponding nodes are connected by an element. Because of its easy generation, nodal graph representation is used in this study.



(a) An Arbitrary Structure (b) Nodal Graph

Figure 3.1: Graph Representation of an arbitrary structure

METIS [3] partitioning algorithms are based on the multilevel approach. As illustrated in Figure 3.2, multilevel approaches are composed of three phases: graph coarsening, initial partitioning, and uncoarsening/refinement. In the graph coarsening phase, a series of graphs is constructed by collapsing together adjacent vertices of the input graph in order to form a coarser graph that resembles the properties of the original graph. The collapsed vertices are described by a multinode that contains the vertex weight of the contributing vertices. Computation of the initial partitioning is performed on the coarsest (and hence smallest) of these graphs, and thus is faster. Then, partition refinement is performed on each level graph, where the partitions of the coarser graph are projected back to the original graph by going through finer and finer graphs.

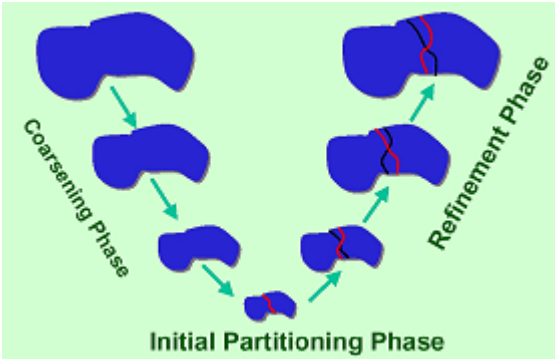


Figure 3.2: Multilevel Approach [3]

3.2.1.2 PARMETIS

Dynamic partitioning or in other words repartitioning approaches are required if the computational load changes as the solution proceeds or it is not possible to compute the computational load of partitions before partitioning. There are several types of repartitioning algorithms but they can be classified into two groups according to their use of the original partition. The first group of algorithms, e.g. scratch-remap, first partition the graph from scratch according to the new vertex weights and use the existing partitioning information to minimize the difference between the original and the new partitions. The other group of algorithms first calculates the imbalance of the original partitions and then attempts to balance them by migrating vertices from the overweight partitions to the under-weight ones. This type of algorithms are called diffusion algorithms because they consider the formentioned problem analogous to the diffusion process where an initial uneven temperature in space drives the movement of heat, and eventually reaches equilibrium [69].

PARMETIS [42] is a parallel multilevel graph partitioning and repartitioning library that implements both scratch-remap and diffusion based repartitioning algorithms. It also uses a multilevel scheme to speed-up the partitioning process. Moreover, it performs all the computations in parallel that enables it to partition and repartition very large sized graphs in a very short amount of time.

3.3 Workload Balancing Algorithm

The flow chart of the workload balancing algorithm is presented in Figure 3.4. The algorithm initiates by converting the structural information into the nodal graph representation. Then, the nodal graph is partitioned into 'n' parts by using METIS [3]

where 'n' is equal to the number of available processors. After assigning a single substructure to each processor, the nodal graph and the initial partitioning information are distributed to every processor. The processors first extract their assigned substructure's subgraphs from the nodal graph using the partitioning information. A subgraph is actually the nodal graph of a substructure with links to adjacent substructures as shown in Figure 3.3.

Then, each processor optimizes the equation numbering of their substructure and calculates the operation counts for condensation. During these computations, it is assumed that there is a single degree of freedom per node.

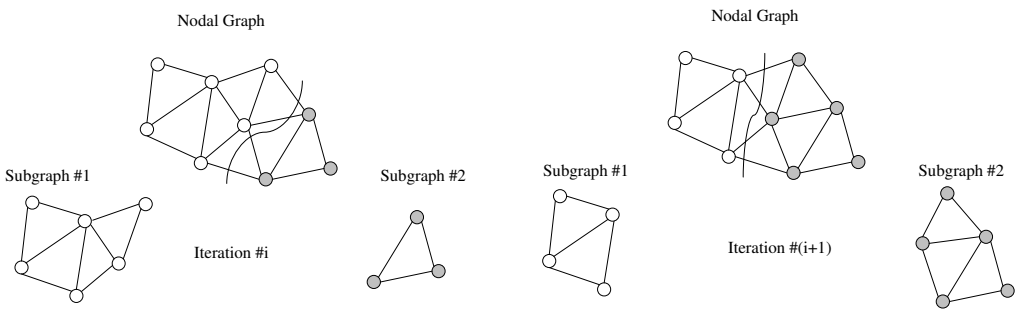


Figure 3.3: Nodal Graph and Subgraphs during iterations

Next, the master computer, which performed the initial partitioning, collects the operation count values and calculates the condensation time estimation of each substructure by dividing the operation count with the condensation speed of the corresponding computer. Then, by approximately estimating the number of elements in each substructure, master computer also computes the amount of time required to generate and assemble the local stiffness matrix for each computer. Then, the master processor checks whether the workload from assembly and condensation of each substructure is balanced or the maximum number of iterations has been reached. If so, the iterations are finalized and the structural data for the solution is prepared. Otherwise, the master processor calculates the imbalance factor according to Equation 3.1 for each substructure and distributes it to other processors.

$$Rt(j) = \frac{Lst(j)}{\sum_{i=1}^{p} Lst(i)} \tag{3.1}$$

where p is the number of computers, $Lst(j)$ is the local solution time estimation for j^{th}

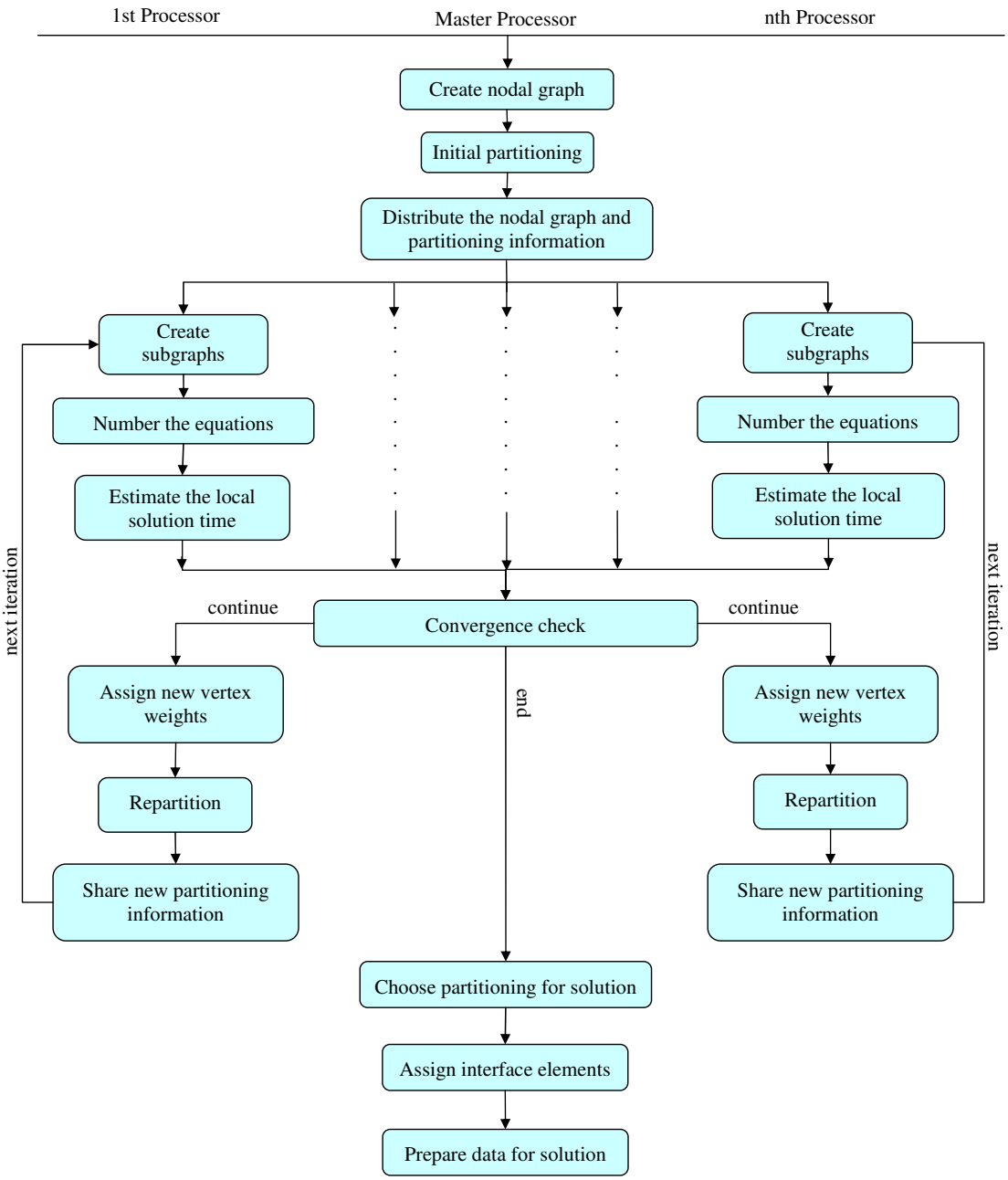


Figure 3.4: The Flow Chart of the Workload Balancing Algorithm

substructure, and $Rt(j)$ is the relative local solution time ratio of the jth substructure. Then, each processor calculates new vertex weights of their substructures by utilizing Equation 3.2 and updates the weights of its vertices.

$$Wv(j) = \frac{Rt(j) \,/\, n(j)}{\sum_{i=1}^{p} Rt(i) \,/\, n(i)} \tag{3.2}$$

In the above equation, $n(j)$ represents the number of vertices in the jth substructure and $Wv(j)$ represents the new vertex weight for the jth substructure. Equation 3.2

calculates the vertex weights in such a way that the relative ratios of the sum of the vertex weights of substructures are equal to the relative ratios of the local solution time estimations.

Then, the current partitioning information is stored and repartitioning is initiated. The substructures are repartitioned by using the scratch-remap type repartitioning algorithms of the PARMETIS [42] library. Once the new partitioning information is obtained, it is distributed to all other computers and the next iteration starts.

When the iterations are finalized, the master processor scans all partitioning results created during the iterations and chooses the one that provides the best estimate of the solution time. The solution time estimation can consider the estimation of either assembly plus condensation times only or the total solution time including the interface solution time.

The final step is the preparation of the structural data where the subgraphs are converted into nodes and element definitions. During that process, the interface elements, whose nodes are on two or more substructures, are assigned to one of their adjacent substructures.

3.4 Test Results

Implemented algorithm is tested on two different PC Clusters. First one, HPCE, is composed of eight identical computers which have Intel P4 3.2 GHz processors and 1 GB RAMs. On the otherhand, second cluster, HPCE2, is composed of eight identical computers which have Intel Core2Quad Q9300@2.5 GHz processors and 3.23 GB RAMs. Intel Core2Quad family processors involve four processors that are theoretically working at 2.5 GHz and are able to share the memory. However, this feature of this cluster is not utilized during these tests. Both of these clusters are connected with ordinary 1 GBit network switches and all computers at both clusters are running Windows XP Professional.

The test results are presented in terms of three different parameters; imbalance ratio, improvement and local solution time. Imbalance ratio can be computed as;

$$Imbalance\,Ratio = \frac{Slowest\,Local\,Solution\,Time}{Fastest\,Local\,Solution\,Time} \tag{3.3}$$

where local solution time is composed of assembly and condensation times. To obtain the improvement at each workload balancing iteration, initial and i^{th} local solution

times are compared as follows;

$$Improvement(i) = \frac{Lst(i)}{Lst(0)} \tag{3.4}$$

where *Lst(i)* denotes the local solution timing for i^{th} workload balancing iteration and *Lst(0)* denotes the local solution timing for initial partitioning. *Improvement(i)* $>$ *1.0*, indicates that initial partitioning is faster than partitioning at i^{th} step.

3.4.1 Illustrative Example

Figure 3.5 and Figure 3.6 illustrate the substructures created during the workload balancing iterations. Scratch-remap is utilized as repartitioning method. For this example problem, a 2D square mesh having 25,600 quadrilateral elements was partitioned into 8 substructures.

Initially, imbalance ratio between the fastest and slowest local solution of substructures was 1.13 and after ten iterations, workload balancing algorithm terminated. 6th iteration was chosen as the best partitioning since it was estimated that this iteration has the fastest local solution timing (refer to Table 3.1). The final imbalance ratio value was 0.33 and the decrease in the governing local solution time (assembly and condensation times) improved around 27%. Thus, balancing the local solution times of the substructures decreases the governing local solution time, but having more balanced local solution times does not always mean that the governing local solution time is faster. There may be another partitioning which the substructures are less balanced but the governing local solution time is faster.

Table 3.1: Workload Balancing Iteration Results for 2DMesh Model

Iteration #	Imbalance Ratio	Improvement	Local Solution Time (s)
Initial	1.13	1.00	2.35
1	0.58	0.90	2.12
2	0.55	0.79	1.86
3	0.77	0.96	2.26
4	0.80	0.87	2.05
5	0.48	0.79	1.87
6	0.59	0.71	1.66
7	0.39	0.72	1.71
8	0.32	0.81	1.90
9	0.19	0.76	1.78
10	0.33	0.73	1.72

When the substructures of chosen iteration (6th) is investigated (refer to Figure 3.6),

there were four similar substructures at the corners having larger number of internal nodes. On the other hand, the smallest substructure was located at the center of the mesh. It had the least number of nodes because all its edges are at the interfaces. The rest of the substructures which were midsize substructures, have interface nodes at their three sides. As a result, the iterations resulted in a decrease in the number of nodes of the substructures which contained more interface nodes.

3.4.2 Actual Testing Models

Table 3.2 presents workload balancing iteration results of StRegis Model on two computers of HPCE cluster. Imbalance ratio at initial partitioning was 1.63 and local solution time was ∼540 seconds. After six iterations, workload balancing algorithm terminates because of no improvement in consecutive iterations. Best partitioning was chosen as the second iteration since it was reduced local solution time to 330 seconds which means 210 seconds of short execution of condensation and assembly steps. Although first iteration was more balanced (Imbalance ratios: $0.03 < 0.04$) than second iteration, latter produced shorter local solution time (Table 3.2). Workload balancing for this special case was completed in 7 seconds which is insignificant when the improvement obtained is considered.

Table 3.2: Workload Balancing Iteration Results for StRegis Model

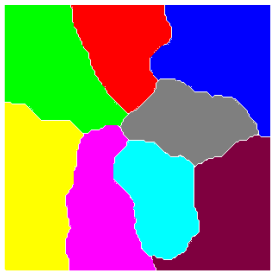
Iteration #	Imbalance Ratio	Improvement	Local Solution Time (s)
Initial	1.63	1.00	539.79
1	0.03	0.66	358.44
2	0.04	0.61	330.82
3	0.27	0.87	470.79
4	1.37	1.09	596.15
5	0.18	0.67	364.46
6	0.18	0.67	364.46

Workload balancing iteration results of GroM Model on four computers of HPCE cluster are given in Table 3.3. Imbalance ratio at initial partitioning was 1.19 and local solution time was ∼486 seconds. Workload balancing procedure spent 16 seconds to find a better partitioning. Procedure terminated since it reached the maximum number of iterations and then had chosen the fifth iteration as the best partitioning and it was the most balanced partitioning, too. With this partitioning, local solution time reduced to ∼336 seconds.

Initial Partitioning

Imbalance Ratio : 1.13

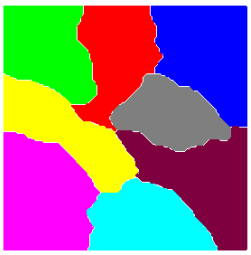
Improvement : 1.00



Iteration #1

Imbalance Ratio : 0.58

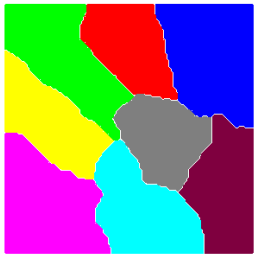
Improvement : 0.90



Iteration #2

Imbalance Ratio : 0.55

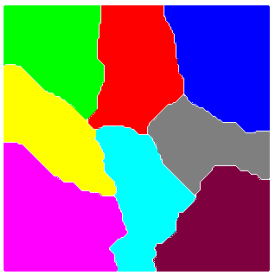
Improvement : 0.79



Iteration #3

Imbalance Ratio : 0.77

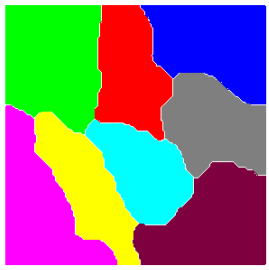
Improvement: 0.96



Iteration #4

Imbalance Ratio : 0.80

Improvement : 0.87

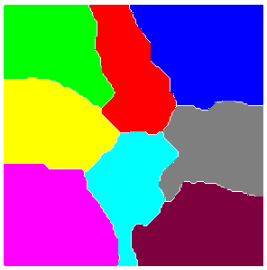


Iteration #5

Imbalance Ratio : 0.48

Improvement : 0.79

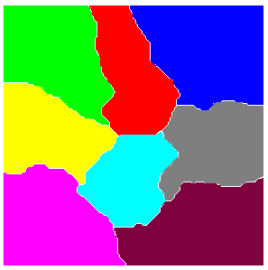
Figure 3.5: Substructures of 2DMesh Model at each workload balancing step



Iteration #6

Imbalance Ratio : 0.59

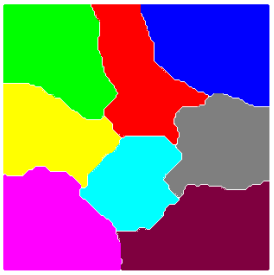
Improvement : 0.71



Iteration #7

Imbalance Ratio : 0.39

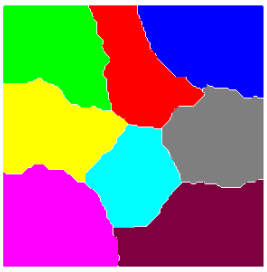
Improvement : 0.72



Iteration #8

Imbalance Ratio : 0.32

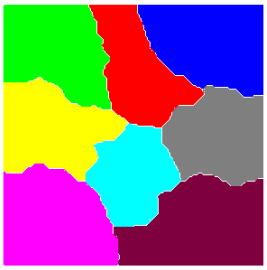
Improvement : 0.81



Iteration #9

Imbalance Ratio : 0.19

Improvement : 0.76



Iteration #10

Imbalance Ratio : 0.33

Improvement : 0.73

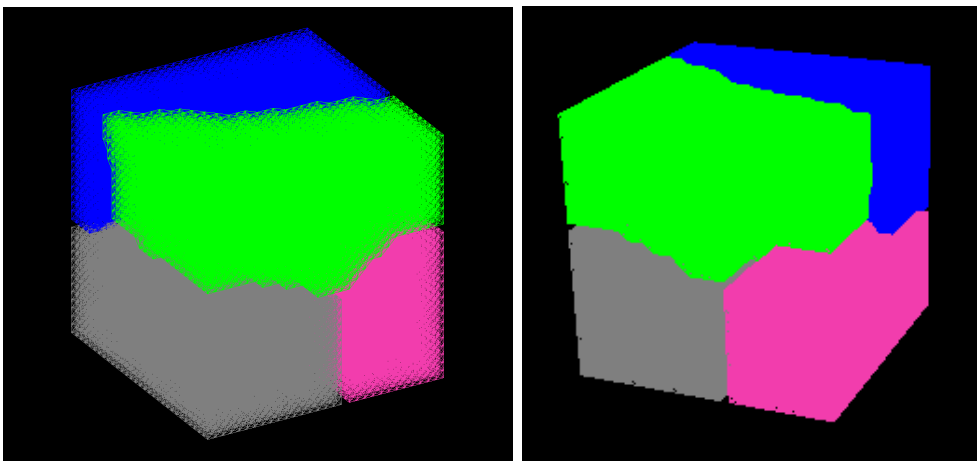
Figure 3.6: Substructures of 2DMesh Model at each workload balancing step (cont.)

Table 3.3: Workload Balancing Iteration Results for GroM Model

Iteration #	Imbalance Ratio	Improvement	Local Solution Time (s)
Initial	1.19	1.00	486.40
1	0.43	0.68	360.32
2	0.35	0.75	383.11
3	0.43	0.64	371.83
4	0.36	0.76	398.69
5	0.14	0.61	335.94
6	0.40	0.73	376.92
7	1.80	1.29	624.46
8	2.18	0.95	490.01
9	1.08	0.98	473.35
10	0.71	0.77	412.69

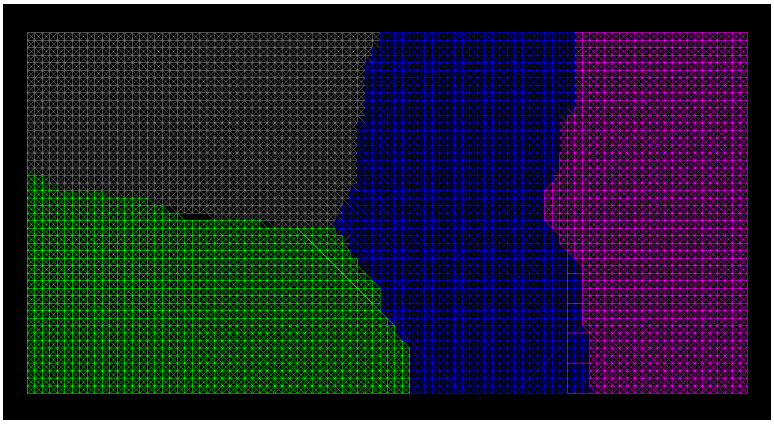
Workload balancing algorithm, could not suggest better partitioning for Cube30 models on both clusters except from a few runs with two and four computers on HPCE cluster. Since Cube30 model is a symmetric and uniform model with same elements, initial partitioning is most of time the best partitioning. In case of GroM model, with the introduction of different type of elements, uniformity of the model is disturbed. Therefore, workload balancing iteration could yield better partitioning. For example, similarly, StRegis Model is highly non-uniform and non-symmetric. Therefore, the improvement obtained in StRegis Model by workload balancing algorithm is generally larger than the one obtained in GroM model.

Three different structural models (Cube30, GroM, and StRegis) which are introduced in Appendix B were partitioned into four substructures for demonstration purposes. Views from different points are demonstrated for Cube30, GroM, and StRegis in Figure 3.7, Figure 3.8, and Figure 3.9, respectively.

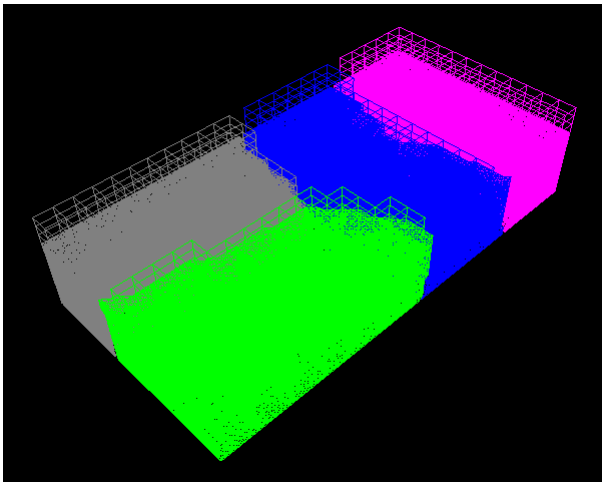


(a) 3D View (b) Perspective View

Figure 3.7: Final Partitioning of Cube30 Model

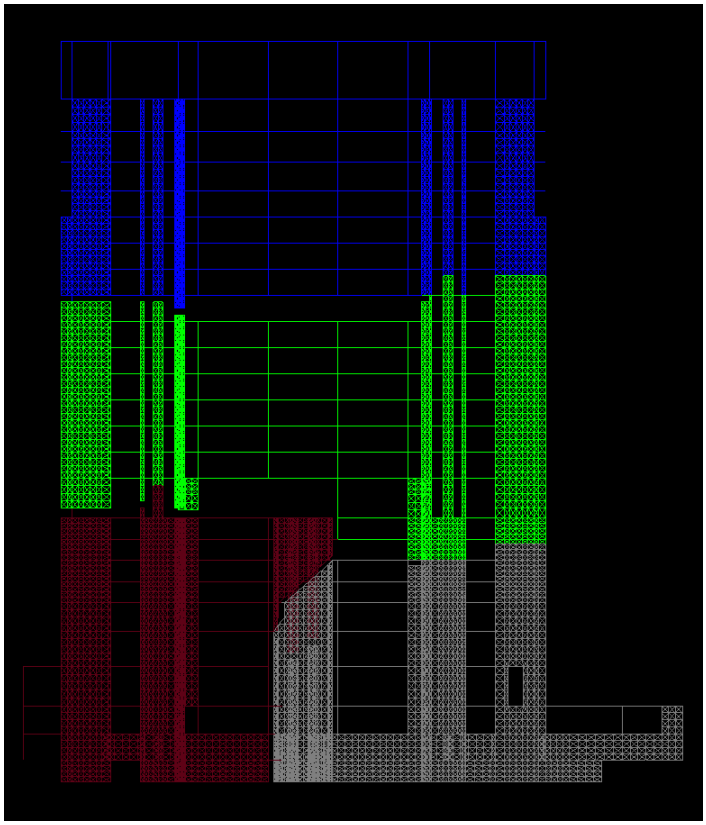


(a) Plan View

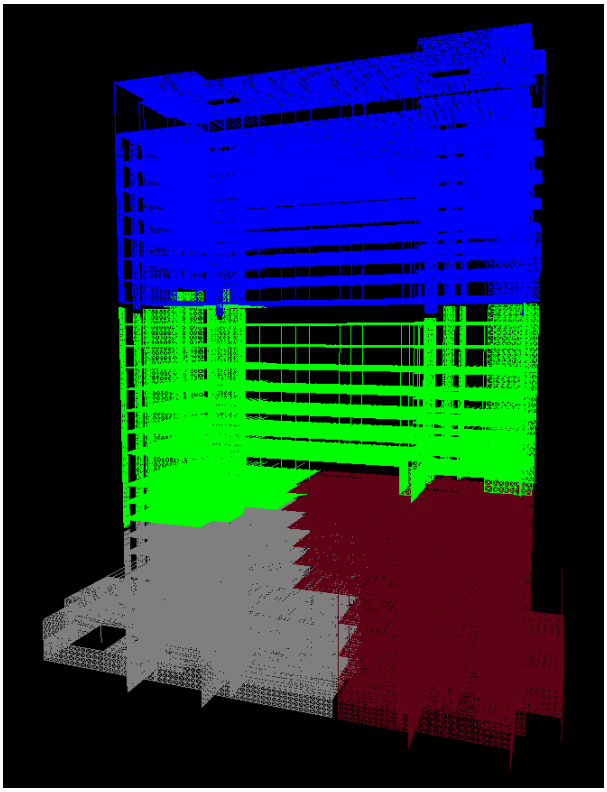


(b) Perspective View

Figure 3.8: Final Partitioning of GroM Model



(a) Elevation View



(b) Perspective View

Figure 3.9: Final Partitioning of StRegis Model

CHAPTER 4

PARALLEL SOLUTION

4.1 Introduction

This chapter focuses on the parallel solution of the structural model which was partitioned into substructures in data preparation step. Initially, general algorithm which is composed of two main stages is presented. This chapter is devoted to these two main topics. First stage is condensation. Condensation algorithm is presented and then test results for this algorithm is discussed. After the condensation, second stage initiates with assembling the contributions of each substructure on interface system. Three important steps in interface system assembly are introduced; data mapping for reducing the amount of data transfer, communication scheme for optimizing the communication order of processors, and runtime data compression for reducing the size of transferred data. Then interface system solution in parallel is presented. Optimization and runtime estimations are discussed. Finally, test results for interface system assembly and solution is presented and discussed in detail.

4.2 Parallel Solution Algorithm

In Figure 4.1, the flow chart for parallel solution is presented. The parallel solution initiates by creating separate structural data at each computer. First, the master computer reads the input file prepared by the data preparation program and sends the nodal, element connectivity, and loading information of each substructure to their corresponding computer. Then, the local solution initiates. Each computer assigns degrees of freedom to its nodes. The nodes of each substructure were written into the input file according to their optimized order and they were placed into the arrays in the same order. Hence, during the assignment process, each node is visited one by one and the nodes' active

degrees of freedom are numbered consecutively. After that, stiffness matrices and force vectors are assembled.

The next step is condensation. The condensations are performed by using a sparse solver which performs LDL^T factorization of symmetric matrices in row-wise fashion. Only the non-zero elements of the lower-triangular part of the stiffness matrix are stored in the compressed column format to minimize the memory usage. Up to this point, neither communication nor synchronization among computers are required.

Then, the interface stiffness matrix is assembled where each computer sends and receives some portion of the interface stiffness matrix. The interface matrix is distributed as 2D rectangular blocks in cyclic manner (block cyclic distribution) to utilize the parallel dense matrix solver of ScaLAPACK [10] library. In this assembly approach, first each computer prepares a data distribution scheme and data buffers that involves the part of the matrix that will be sent to a particular computer. Then, the data transfer initiates in such a way that none of the computers stay idle. As the distribution of the interface stiffness matrix is finalized, it is solved and the displacements are obtained.

Once the interface displacements are obtained, they are distributed in such a way that each computer receives the displacements belonging to their interface nodes. In the final step, the local solution is finalized by recovering the internal displacements and computing the stresses for each element in the substructure.

In the parallel solution method presented in Figure 4.1, all the steps other than interface solution are fully parallel. Each computer run the algorithms over its owned substructure without requiring any information from other substructures, in other words, these procedures are local. However, during the interface solution, due to high data dependency, resulting interface problem is solved by using parallel dense solvers in global. To sum up, all of the steps in solution are parallelized.

After the completion of assembly, system of equations at the substructure interfaces, which are composed of the interface stiffness matrix and load vectors, are distributed over the cluster according to block cyclic distribution. The interface stiffness matrix can vary from a full matrix to a variable band matrix depending on the number of substructures. Thus, two different parallel solvers of ScaLAPACK [10] are utilized for the solution of parallel full matrix solver and band solver. Details of these parallel solvers will be discussed in Section 4.4.2.

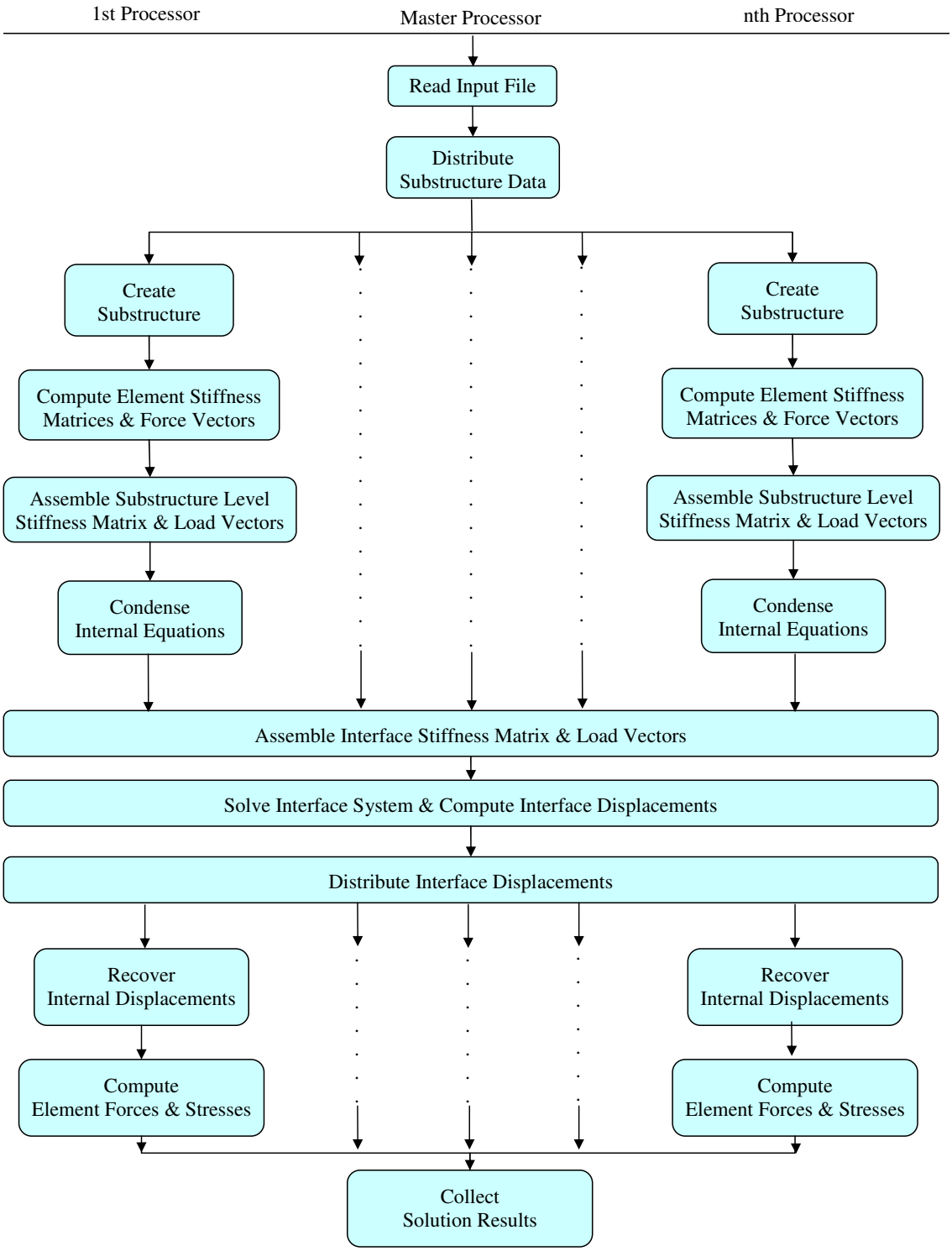


Figure 4.1: Parallel Solution Flow Chart

4.3 Condensation

In substructure based methods, after the assembly of stiffness matrices and force vectors, internal nodes are condensed to interface nodes. The algorithm and test results for condensation, will be discussed in following sections.

4.3.1 Algorithm

In the current version of the substructure based parallel solution method, the condensations are performed by using a sparse solver which performs LDL^T factorization of symmetric matrices in row-wise fashion. This condensation algorithm is the adjusted version of the sparse Cholesky solver supplied by Timothy A. Davis [70]. In this algorithm, only the non-zero elements of the lower-triangular part of the stiffness matrix are stored in the compressed column format to minimize the memory usage. Condensation algorithm that is implemented is as follows and explanations about how does the algorithm proceeds is commented inside the code;

```
int ldl_condense (  //returns n if successful, k if D (k,k) is zero
 unsigned int n,  //A and L are n-by-n, where n >= 0
 unsigned int gdof,
 unsigned int Ap[],  //input of size n+1, not modified
 unsigned int Ai[],  //input of size nz=Ap[n], not modified
 double Ax[], //input of size nz=Ap[n], not modified
 int Lp[],  //input of size n+1, not modified
 int Parent[], //input of size n, not modified
 int Lnz[], //output of size n, not defn. on input
 int Li[],  //output of size lnz=Lp[n], not defined on input
 double Lx[], // output of size lnz=Lp[n], not defined on input
 double D[], //output of size n, not defined on input
 double Y[], //workspace of size n, not defn. on input or output
 int Pattern[], //workspace of size n, not defn. on input or output
 int Flag[], //workspace of size n, not defn. on input or output
 int P[],  //optional input of size n
 int Pinv[]) //optional input of size n
{
 double yi = 0., l_ki  = 0.;
 int p = 0, kk = 0, p2 = 0, len = 0;
```

```
 for (unsigned int k = 0 ; k < n ; k++) {
  //compute nonzero Pattern of kth row of L, in topological order
  Y [k] = 0.0;        //Y(0:k) is now all zero
  unsigned int top = n; //stack for pattern is empty
  Flag [k] = k;        //mark node k as visited
  Lnz [k] = 0;        //count of nonzeros in column k of L
  kk = (P) ? (P [k]) : (k) ;  //kth original, or permuted, column
  p2 = Ap [kk+1];
  for (p = Ap [kk] ; p < p2 ; p++)  {
   unsigned int i = (Pinv) ? (Pinv [Ai [p]]) : (Ai [p]) ;
   if (i <= k)    {
    Y [i] += Ax [p];  //scatter A(i,k) into Y (sum duplicates)
    for (len = 0 ; Flag [i] != k ; i = Parent [i]) {
     Pattern [len++] = i;    //L(k,i) is nonzero
     Flag [i] = k;       //mark i as visited
    }
    while (len > 0)  Pattern [--top] = Pattern [--len];
   }
  }
  //compute numerical values kth row of L (a sparse triangular solve)
  D [k] = Y [k]; Y [k] = 0.0;       //get D(k,k) and clear Y(k)
  for ( ; top < n ; top++)  {
   int i = Pattern [top];//Pattern [top:n-1] is pattern of L(:,k)
   yi = Y [i];  Y [i] = 0.0;    //get and clear Y(i)
   p2 = Lp [i] + Lnz [i];
   if (i<gdof) {
    for (p = Lp [i] ; p < p2 ; p++)
     Y [Li [p]] -= Lx [p] * yi;
    l_ki = yi / D [i];       //the nonzero entry L(k,i)
    D [k] -= l_ki * yi;
   }  else  {  p = p2;  l_ki = yi;    }
   Li [p] = k;       //store L(k,i) in column form of L
   Lx [p] = l_ki; Lnz [i]++; //increment count of nonzeros in col i
  }
  if (D [k] == 0.0) return (k);       //failure, D(k,k) is zero
 }
 return (n); //success, diagonal of D is all nonzero
}
```

Mathematical background about condensation indicates an important feature of condensation which is that condensation is a local process. In other words, during condensation neither communication nor synchronization among computers are required. Therefore, performance of this process is only depends on the computational speed of the computer, performance of the algorithm and the type of problem.

4.3.2 Test Results

Implemented condensation algorithm was tested on two different PC Clusters. First one, HPCE, is composed of eight identical computers which have identical Intel P4 3.2 GHz processors and 1 GB RAMs. On the otherhand, second cluster, HPCE2, is composed of eight identical computers which have Intel Core2Quad Q9300@2.5 GHz processors and 3.23 GB RAMs. Intel Core2Quad family processors involve four processors that are theoretically working at 2.5 GHz and are able to share a single memory. Both of these clusters are connected with ordinary 1 GBit network switches and all computers at both clusters are running Windows XP Professional. Besides, communication, coordination and computation parameters measured for them are presented in Appendix A. Structural models analyzed by implemented algorithms are introduced in Appendix B.

Table 4.1: Condensation Timings

(a) For HPCE

# of Computers	Condensation Time (s)		
	Cube30	**GroM**	**StRegis**
2	901.0	522.0	283.0
4	226.7	260.8	130.0
6	119.0	160.5	80.0
8	74.0	98.5	55.0

# of Computers	Condensation Time (s)		
	Cube30	**GroM**	**StRegis**
2	459.6	314.8	165.0
4	149.0	172.7	87.8
6	78.5	74.8	60.9
8	55.0	53.2	38.9

(b) For HPCE2

Table 4.1 presents the time spent during the condensation of all substructures on HPCE and HPCE2 clusters. All results are given in seconds. For GroM and StRegis

models, condensation timings are reduced more or less in scale with the increasing number of computers. For example, condensation of GroM model by 2 computers of HPCE cluster requires almost two times of the time spent during condensation of same model by 4 computers. Similarly, the time spent during condensation of StRegis model by 6 computers of HCPE2 cluster is approximately 2/3 of the time spent for condensation of same model by 4 computers.

When the test results for Cube30 model is investigated, it is obvious that condensation timings are reduced more than the ratio of the number of computers utilized. This is reasonable when the properties of Cube30 model are considered. Since Cube30 model is generated by dividing an imaginary solid cube by 30 equal distances at each dimension, it is composed of brick elements that are connected each other on almost all edges. Therefore, resultant mathematical model is symmetric and has significantly large bandwidth. Utilizing substructuring on this model results in a significant reduction in bandwidth of each substructure and super speed-up values can be reasonably occurred.

Table 4.2: Condensation Speeds for HPCE Computers

HPCE	**Condensation Speed (MFlops)**		
	Cube30	**GroM**	**StRegis**
CEC1	433.0	441.5	410.4
CEC2	417.2	417.0	405.0
CEC3	413.2	423.1	417.0
CEC4	437.7	434.4	409.5
CEC5	415.2	422.9	410.5
CEC6	435.3	439.2	423.7
CEC7	432.5	434.4	427.2
CEC8	416.3	427.4	394.8

Table 4.2 and Table 4.3, demonstrates the measured condensation speeds for computers at HPCE cluster and HPCE2 cluster, respectively. These speeds are calculated by dividing the floating-point operation count during condensation algorithm by the measured time spent for condensation on each computer. As it can be noticed, although the computers in each cluster are identical, condensation times are not identical but fairly lies between 410∼430 $MFlops$ for computers at HPCE cluster and 730∼760 $MFlops$ for computers at HPCE2 cluster. This reasonable, when the differences of the substructures and fill-in occurrence during computation are considered.

Table 4.3: Condensation Speeds for HPCE2 Computers

HPCE2	Condensation Speed (MFlops)		
	Cube30	**GroM**	**StRegis**
CEC14	764.6	751.0	735.1
CEC15	733.6	771.3	733.7
CEC16	764.6	769.3	738.2
CEC17	756.0	767.0	747.3
CEC18	763.9	768.9	761.5
CEC19	760.2	749.9	728.5
CEC20	754.3	763.9	735.7
CEC21	759.5	760.9	741.9

To sum up, obtained computational speeds for condensation algorithm are considerably slower than the theoretical computational speeds of computers. This situation can be originated from two problems explained in previous sections. First one is indirect addressing. As it is presented in condensation algorithm implemented, there are numerous attempts to access List[index] through the algorithm and each of these indices are kept in additional lists such as index arrays (refer to Section 1.2.1). Therefore, access time for variables of any computation is reducing the overall algorithm speed. Second reason for degraded computational speed is not having the data as a whole in cache memory (computational unit). In contrast to dense block solvers, sparse solvers are accessing the different location of the matrix so computer is spending more time for transferring data to the computational unit than computing the operation. However, dense block solvers, brings a supernode or a subblock of a matrix into computational unit once and operates on it as much as required.

4.4 Interface System

In substructure based methods, first the contributions of each substructure to the interface system are obtained by static condensation. Then, the condensed substructure stiffness matrices are assembled and equilibrium equations are solved for the displacements at the interface nodes. Therefore, the interface system solution can be divided into two main sections as assembly and solution.

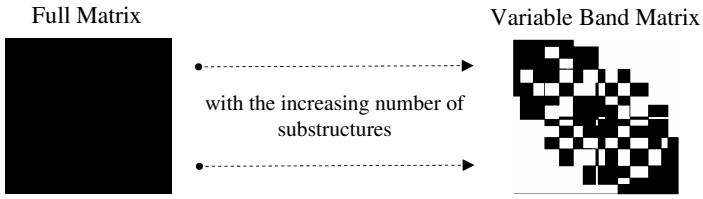


Figure 4.2: Non-zero element pattern for interface stiffness matrix

Interface stiffness matrix varies from variable band matrix to full matrix (Figure 4.2), depending on the number of substructures. For example, in case of two substructures, the interface stiffness matrix is a full matrix since both of the substructures has contributions to all of the interface nodes. However, with the increasing number of substructures, the density of the interface stiffness matrix decreases. Although, the bandwidth of the interface stiffness matrix is reduced for the fastest solution time during the data preparation step, a parallel interface solver that is capable of efficiently solving both banded matrices and dense matrices will improve the performance of the solution framework significantly.

4.4.1 Interface System Assembly

As demonstrated with an illustrative example in Figure 4.3, after the condensation, each computer has some portion of the interface stiffness matrix and load vectors.

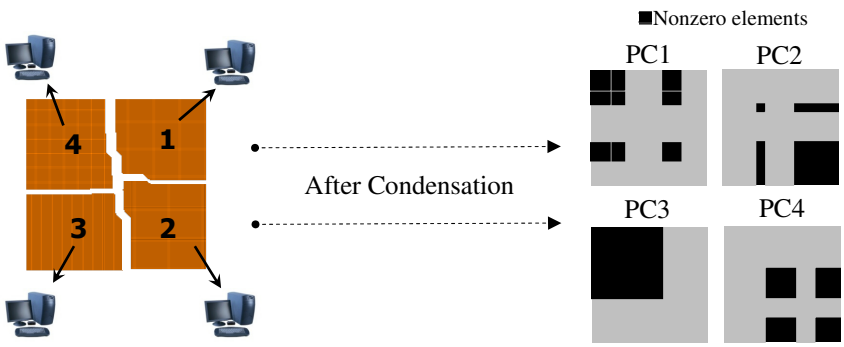


Figure 4.3: The contributions of each substructure to interface stiffness matrix

As shown in Figure 4.4, the contributions of each substructure are irregularly occurring. Besides, more than one substructure have contributions on a single interface node. Therefore, before the solution initiates, it is necessary to assemble all the contributions and then distribute the assembled system over the cluster according to a valid distribution scheme utilized by the interface solution algorithm.

One way of assembling the interface stiffness matrix is to gather all the contributions

and then distribute them to other computers. Such an approach requires considerable amount of data transfer that highly reduce the performance of the solution framework (more detailed explanation is presented in Appendix A). Because of this reason, an improved procedure named as "*On the Fly Assembly*" was developed.

Figure 4.4: Redistribution over cluster

A short summary of this procedure is presented in Figure 4.5. First, each processor allocates sufficient memory to store the resultant local matrix at the end of this procedure. Resultant local matrix is the part of the interface system which will be stored before interface system solution initiates. Dimensions of this local matrix can be computed by using data mapping equations which will be discussed in following sections. After allocating memory for the local matrix, each processor creates data buffers for all other processors. Then, for each contribution of current substructure, data is mapped over cluster by using 2D block cyclic mapping. If that contribution lies on local matrix of another processor according to mapping scheme, that data which should be sent to that processor is added to corresponding data buffer. On the other hand, if that contribution lies on the territory of the current processor, it is appended to a particular place of the local matrix. After the completion of this loop, data buffers are interchanged among the processors according to the prepared communication scheme. During the communication, data buffers are compressed before the data is sent and decompressed after the data is received.

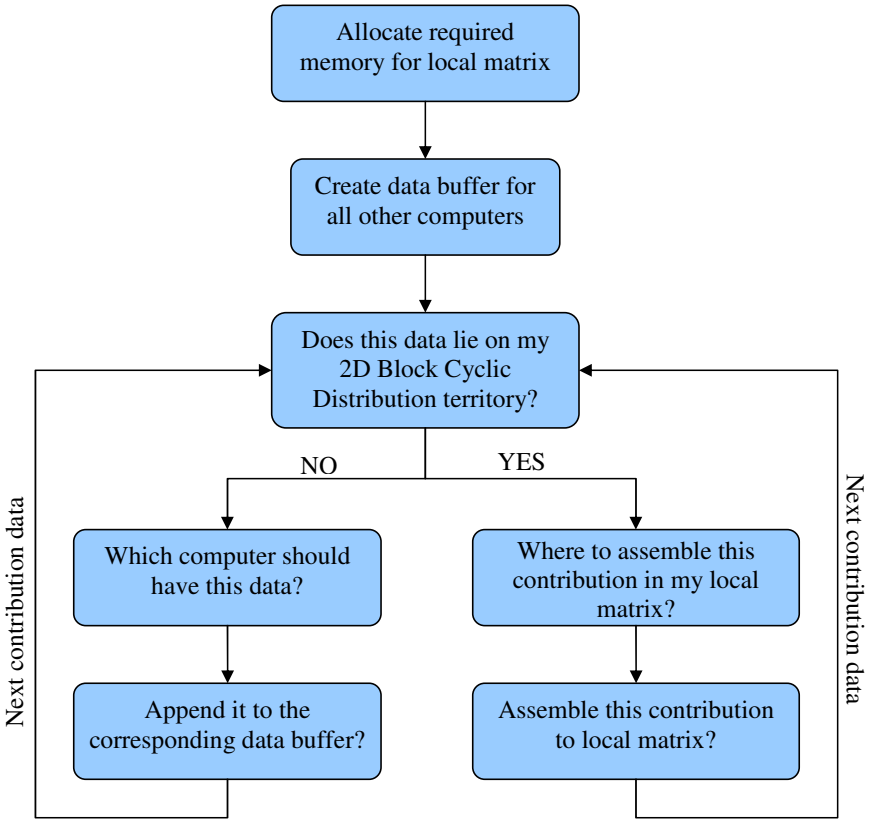


Figure 4.5: On the Fly Assembly

Basically, the *"On the Fly Assembly"* procedure is mainly composed of three stages; data mapping for reducing the amount of data transfer, communication scheme for optimizing the communication order of processors, and runtime data compression for reducing the size of transferred data.

4.4.1.1 Data Mapping

The ScaLAPACK software [10] assumes that the user's input data has been distributed on a two-dimensional grid of processors according to the block cyclic scheme. This means that subblocks (rather than single elements) of the matrix are distributed to processors in a wraparound fashion along the processor grid. For a given number of processors, the parameters of this family of data distributions are the shape of the processor grid and the size of the block used to partition and distribute the matrix entries over the processor grid. These parameters affect the number of messages exchanged during the operation, the aggregated volume of data communicated, and the computational load balance [1].

4.4.1.1.1 Block Cyclic Data Distribution

The way a matrix is distributed over the processors has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The block cyclic distribution provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers [2]. In block cyclic data distribution, blocks separated by a fixed stride in the column and row directions are assigned to the same processor.

Suppose that, in a 1D matrix, there are M values indexed by the integers $0, 1, \ldots, M-1$. In the block cyclic data distribution, the mapping of the global index, m, can be expressed as $m \mapsto \langle p, b, i \rangle$, where p is the logical processor number, b is the block number in processor p, and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is m_b, the block cyclic data distribution may be written as follows:

$$m \mapsto \left\langle s \, mod \, P, \left\lfloor \frac{s}{P} \right\rfloor, m \, mod \, m_b \right\rangle \tag{4.1}$$

where $s = \lfloor m/m_b \rfloor$ and P is the number of processors. The distribution of a block-partitioned matrix can be regarded as the tensor product of two such mappings: one that distributes the rows of the matrix over *NPROW* processors, and another that distributes the columns over *NPCOL* processors. That is, the matrix element indexed globally by *(m; n)* can be written as

$$(m, n) \mapsto \langle (p, q), (b, d), (i, j) \rangle \tag{4.2}$$

where n, $\langle q, d, j \rangle$ are the corresponding parameters of the column mapping over for m, $\langle p, b, i \rangle$ parameters of the row mapping.

Figure 4.6 (a) presents an example of the block cyclic data distribution, where a matrix with 12×12 blocks is distributed over a 2×3 grid. The numbered squares represent blocks of elements, and the number indicates at which location in the processor grid the block is stored. Thus, all blocks labeled with the same number are stored in the same processor. The *slanted* numbers, on the left and on the top of the matrix, represent indices of a row of blocks and of a column of blocks, respectively. Figure 4.6 (b) reflects the distribution from a processor point-of-view. Each processor has 6×4 blocks.

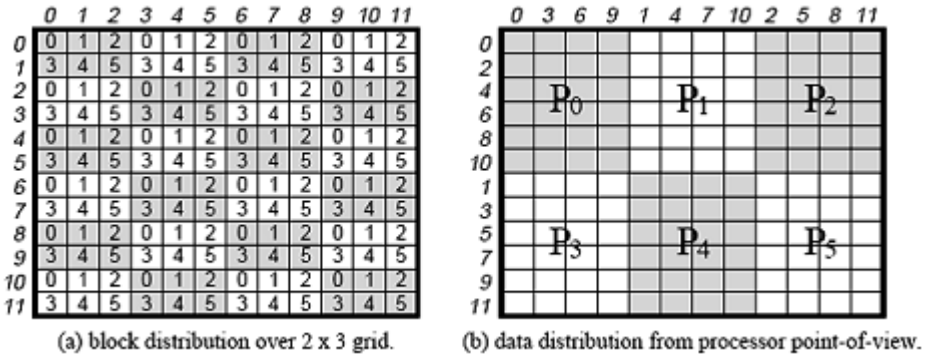


Figure 4.6: Example of a block cyclic data distribution [2]

The block cyclic data distribution can reproduce most data distributions used in linear algebra computations. For example, one-dimensional distributions over rows or columns are obtained by choosing $NPROW$ or $NPCOL$ to be equal to 1.

The non-scattered decomposition (or pure block distribution) is just a special case of the cyclic distribution in which the block size is given by $MB = \lceil M/NPROW \rceil$ and $NB = \lceil N/NPCOL \rceil$. That is,

$$(m,\, n) \mapsto \left\langle \left(\left\lfloor \frac{M}{MB} \right\rfloor,\, \left\lfloor \frac{N}{NB} \right\rfloor \right),\, (0,\, 0),\, (M \bmod MB,\, N \bmod NB) \right\rangle \tag{4.3}$$

Similarly a purely scattered decomposition (or two dimensional wrapped distribution) is another special case in which the block size is given by $m_b = n_b = 1$,

$$(m,\, n) \mapsto \left\langle (M \bmod NPROW,\, N \bmod NPCOL),\, \left(\left\lfloor \frac{M}{NPROW} \right\rfloor,\, \left\lfloor \frac{N}{NPCOL} \right\rfloor \right),\, (0,\, 0) \right\rangle \tag{4.4}$$

However, in literature it is experimentally tested and theoretically proved that for blocked algorithms, a 2D block-cyclic distribution is superior to a 1D block-cyclic distribution [1, 2, 19, 71].

4.4.1.1.2 Data Mapping for Block Cyclic Distribution

By using the fundamentals of the block cyclic distribution described in the previous section, a mapping can be obtained as follows [72]. Suppose that matrix A, which has dimensions of $M \times N$, will be mapped on a processor grid of $NPROW$ number of processor rows and $NPCOL$ number of processor columns ($NPROW \times NPCOL = NP$ number of processors totally) with subblocks that has dimensions of $MB \times NB$ where $(1 \leq MB \leq M)$, $(1 \leq NB \leq N)$.

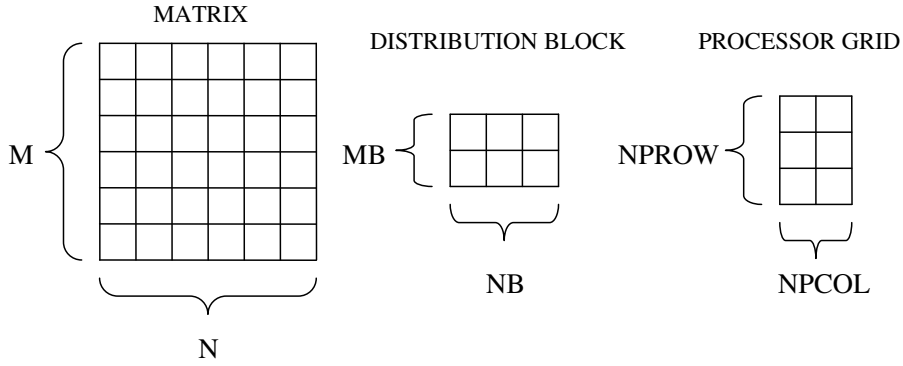


Figure 4.7: 2D Block Cyclic Mapping Parameters

These parameters which are demonstrated in Figure 4.7 define the 2D block cyclic mapping. Processor grid may be a logical grid that is formed virtually. Matrix dimensions are fixed, however the remaining parameters should be chosen carefully based upon an analysis of the cluster and a knowledge of the blocked algorithm that will be used. Once the number of processors, logical processor grid dimensions, and the block sizes for the decomposition are decided, the local memory per allocated processor is determined. The number of rows from the global $M{\times}N$ matrix A that processors in logical process row, $prow$ $(0 \leq prow < nprow)$ own is defined by m_{prow}. The number of columns from the global $M{\times}N$ matrix A that processors in logical processor column $pcol$ own is defined by n_{pcol}. For processor ($prow$, $pcol$) from the logical processor grid, the local matrix $A_{prow,pcol}$ has dimensions $m_{prow} \times n_{pcol}$ and $M = \sum_{prow=0}^{NPROW-1} m_{prow}$, $N = \sum_{pcol=0}^{NPCOL-1} n_{pcol}$ where m_{prow},n_{pcol} are calculated as follows:

$$m_{prow} = \begin{cases} (\left\lfloor \frac{\lfloor \frac{M}{MB} \rfloor}{NPROW} \right\rfloor + 1) * MB \text{ , if} & prow < (\lfloor \frac{M}{MB} \rfloor \bmod NPROW), \\ \left\lfloor \frac{\lfloor \frac{M}{MB} \rfloor}{NPROW} \right\rfloor * MB + M \bmod MB \text{ , if} & prow = (\lfloor \frac{M}{MB} \rfloor \bmod NPROW), \\ \left\lfloor \frac{\lfloor \frac{M}{MB} \rfloor}{NPROW} \right\rfloor * MB \text{ , if} & prow > (\lfloor \frac{M}{MB} \rfloor \bmod NPROW). \end{cases} \tag{4.5}$$

$$n_{pcol} = \begin{cases} (\left\lfloor \frac{\lfloor \frac{N}{NB} \rfloor}{NPCOL} \right\rfloor + 1) * NB \text{ , if} & pcol < (\lfloor \frac{N}{NB} \rfloor \text{ mod } NPCOL), \\ \left\lfloor \frac{\lfloor \frac{N}{NB} \rfloor}{NPCOL} \right\rfloor * NB + N \text{ mod } NB \text{ , if} & pcol = (\lfloor \frac{N}{NB} \rfloor \text{ mod } NPCOL), \\ \left\lfloor \frac{\lfloor \frac{N}{NB} \rfloor}{NPCOL} \right\rfloor NB \text{ , if} & pcol > (\lfloor \frac{N}{NB} \rfloor \text{ mod } NPCOL). \end{cases} \tag{4.6}$$

In global to local 2D block cyclic index mappings in which a given (i,j) index for the global matrix A is transformed into 2D block cyclically mapped indexes $(i_{prow,pcol}, j_{prow,pcol})$ for local matrix $A_{prow,pcol}$. First, the processor $(prow, pcol)$ that owns the specific global element (i,j) is determined by

$$prow = \left\lfloor \frac{i}{MB} \right\rfloor \text{mod NPROW} \tag{4.7}$$

$$pcol = \left\lfloor \frac{j}{NB} \right\rfloor \text{mod NPCOL} \tag{4.8}$$

Then, the indices of the local matrix on processor $(prow, pcol)$ can be labeled as $(i_{prow,pcol}, j_{prow,pcol})$ where $0 \leq i_{prow} < m_{prow}$, $0 \leq i_{pcol} < n_{pcol}$. The assignment is

$$i_{prow} = \left\lfloor \frac{\lfloor \frac{i}{MB} \rfloor}{NPROW} \right\rfloor * MB + (i \text{ mod MB}) \tag{4.9}$$

$$j_{pcol} = \left\lfloor \frac{\lfloor \frac{j}{NB} \rfloor}{NPCOL} \right\rfloor * NB + (j \text{ mod NB}) \tag{4.10}$$

The opposite is local to global 2D block cyclic index mapping. In this mapping, a specific element $(i_{prow,pcol}, j_{prow,pcol})$ where $0 \leq i_{prow} < m_{prow}$, $0 \leq i_{pcol} < n_{pcol}$ in local matrix $A_{prow,pcol}$ on a particular processor $(prow, pcol)$ can be located by (i, j) indices in global matrix A that distributed over the processor grid dimensions $(NPROW, NPCOL)$ by the block dimensions (MB, NB). The assignment for (i, j) is as follows;

$$i = i_{prow} * MB + \left\lfloor \frac{i_{prow}}{MB} \right\rfloor * NPROW * MB + (i_{prow} \text{ mod MB}) \tag{4.11}$$

$$j = j_{pcol} * NB + \left\lfloor \frac{j_{pcol}}{NB} \right\rfloor * NPCOL * NB + (i_{pcol} \text{ mod NB}) \tag{4.12}$$

By using the fundamentals above the global matrix A is mapped into local matrices $A_{nprow,npcol}$ at each computer with minimum overhead as shown in Figure 4.8.

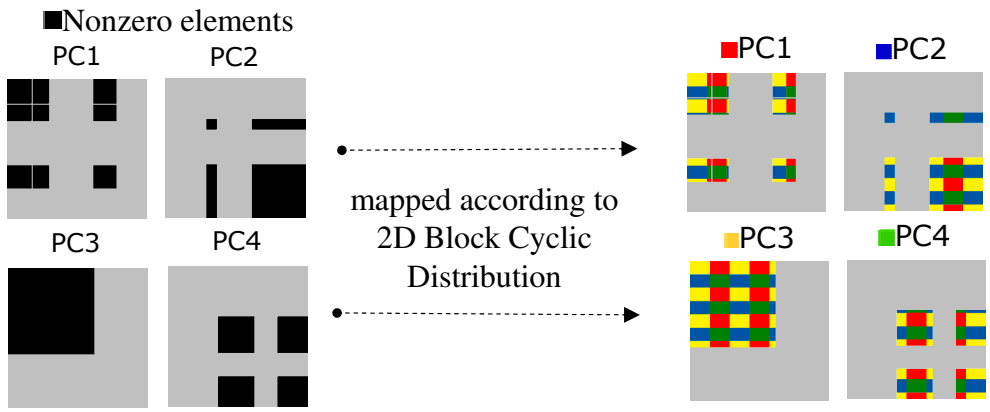


Figure 4.8: 2D Block Cyclic Mapping

After the condensation, the data exist in each processor is not in an order required by the solution algorithm. Thus, by using the 2D block cyclic mapping, each processor investigate all the data exist in its memory and decide the owner of each data (Figure 4.8) and forms a buffer for each neighbour processor (Figure 4.9). If the investigated contribution data from i^{th} substructure belongs to processor P_i according to 2D block cyclic mapping, that data is assembled into the local matrix A_i as soon as possible. Otherwise, it is appended to the corresponding data buffer. Besides, if the value of data is equal to zero, such data is omitted and not transferred to other processor.

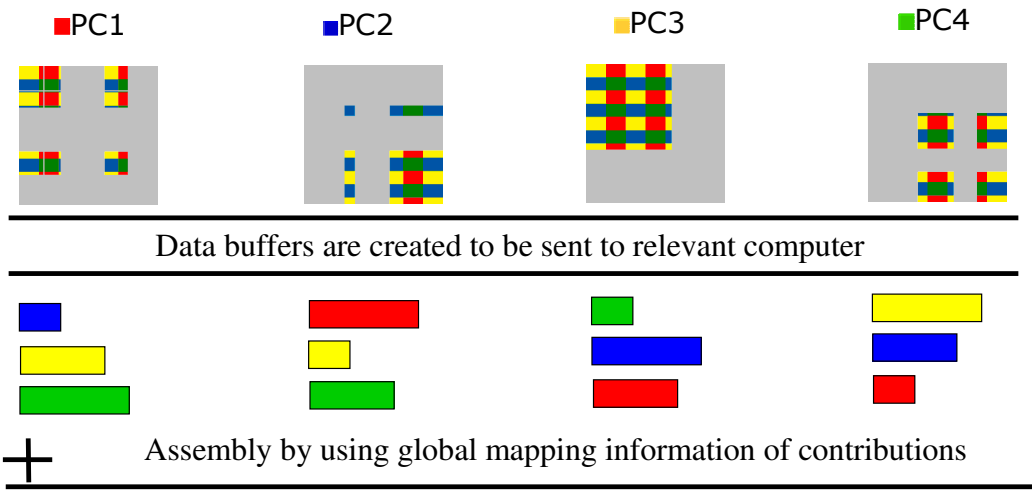


Figure 4.9: Data buffers to be sent to relevant computers

4.4.1.2 Communication Scheme

For *NP* number of processors, each processor forms *(NP-1)* data buffers. At this point, a communication scheme is required to speed-up the data transfers among computers by avoiding processors from staying idle. For example, if processor i tries to sent $C_{i,j}$ (the contribution of substructure i to the matrix territory of processor j) to processor j when processor j communicating with another processor, processor i should wait until processor j becomes idle. This situation increases the time spent during the interface stiffness matrix assembly. Therefore, a communication scheme that organize the order of data communication among computers would increase the speed of assembly step.

Table 4.4: Forming Communication Scheme

Job List	Communication Scheme			
$P_0 \leftrightarrow P_1$ $P_1 \leftrightarrow P_5$	STEP			
$P_0 \leftrightarrow P_2$ $P_2 \leftrightarrow P_3$	1	$P_0 \leftrightarrow P_1$	$P_2 \leftrightarrow P_3$	$P_4 \leftrightarrow P_5$
$P_0 \leftrightarrow P_3$ $P_2 \leftrightarrow P_4$	2	$P_0 \leftrightarrow P_4$	$P_1 \leftrightarrow P_3$	$P_2 \leftrightarrow P_5$
$P_0 \leftrightarrow P_4$ $P_2 \leftrightarrow P_5$	3	$P_0 \leftrightarrow P_2$	$P_1 \leftrightarrow P_5$	$P_3 \leftrightarrow P_4$
$P_0 \leftrightarrow P_5$ $P_3 \leftrightarrow P_4$	4	$P_0 \leftrightarrow P_3$	$P_1 \leftrightarrow P_4$	
$P_1 \leftrightarrow P_2$ $P_3 \leftrightarrow P_5$	5	$P_0 \leftrightarrow P_5$	$P_1 \leftrightarrow P_2$	
$P_1 \leftrightarrow P_3$ $P_4 \leftrightarrow P_5$	6	$P_2 \leftrightarrow P_4$	$P_3 \leftrightarrow P_5$	
$P_1 \leftrightarrow P_4$				

As a first step, the communication requirements from each processor is gathered in a job list and step by step these jobs are grouped by eliminating the busy processors at each step (Table 4.4). For example, suppose that there are six processors in the processor grid and each processor requires communication with all other processors in the grid. This is similar to hand shaking of six individuals in a meeting. At the first step, an arbitrary communication (for example $P_0 \leftrightarrow P_1$) is chosen and then another communication that does not require any communication with P_0 and P_1 is searched from the job list. While adding an entry to the communication scheme, that entry is also removed from the job list. Searching for the independent communications from job list continues until only dependent communications left in job list for that step. Then, algorithm moves to the next step and forms communication steps as many as required to clear up the whole job list.

4.4.1.3 Data Compression

Assembly of the interface system requires considerable amount of data to be transferred within the processor grid. Because of this reason, communication time may govern the overall interface solution time. Ke et al. [73] indicates that for many MPI [54] applications large messages dominate the overall message makeup and they proposed a framework that compresses the data before sent and decompresses it after received to reduce the communication overhead. With this understanding, the procedure presented in Figure 4.10 is implemented in the framework. First of all, to avoid from any memory insufficiency, sender and receiver computers check whether there are enough memory for compressing and decompressing of data to be transferred. If there are enough memory at both computers, sender computer compresses the data and sends it to the receiver computer with a checksum value. Checksum is a fixed-size datum computed from an arbitrary block of digital data for the purpose of detecting accidental errors that may have been introduced during its transmission or storage. The integrity of the data can be checked at any later time by recomputing the checksum and comparing it with the stored one. If the checksums do not match, the data was certainly altered. After receiving the data, receiver computer uncompresses the data and compares its checksum with the received checksum value. If data integrity is satisfied, data transfer with runtime compression is completed successfully. If not, uncompressed data is requested from sender computer.

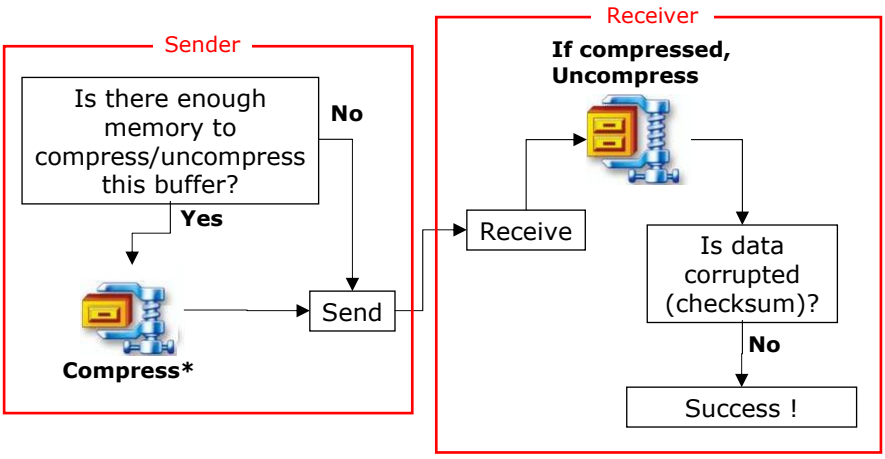


Figure 4.10: Runtime Data Compression

According to Ke et al. [73], the only way of achieving an improvement with this approach is to use a compression algorithm that satisfy the following inequality;

$$\frac{R}{S_{decompress}} + \frac{1}{S_{compress}} < \frac{1-R}{S_{transfer}} \tag{4.13}$$

where R is the data compression rate and $S_{decompress}$ is decompression speed in $MByte/s$, $S_{compress}$ is compression speed in $MByte/s$ and $S_{tranfer}$ is data transfer speed in $MByte/s$. During this computation, it is assumed that runtime data compression is used for large data buffers ($>$ 100 KByte) such that any start-up latency for compression, decompression and data transfer routines can be omitted.

4.4.2 Interface System Solution

Static condensation of each substructure to its interface nodes and assembly of these contributions, a new system of equations that is highly dependent to the contribution of each substructure to the interface system is formed. Depending on the number of substructures and partitioning, interface stiffness matrix can vary from a full matrix to a variable band matrix. Because of the varying nature of interface stiffness matrix two parallel solvers of ScaLAPACK [10] are implemented at this stage. Besides, both of these routines utilizes Cholesky decomposition to solve the system.

4.4.2.1 Implementation

ScaLAPACK [10] requires that coefficient matrix and load vectors must be distributed on the processor grid prior to the invocation of a ScaLAPACK routine. With On-The-Fly assembly, interface system is distributed on the processor grid. In order to notify, ScaLAPACK routines about the distribution layout of the coefficient matrix and load vectors across the processor grid, they must be assigned to an *array descriptor*. This array descriptor is most easily initialized with a call to a ScaLAPACK TOOLS routine called DESCINIT and must be set prior to the invocation of a ScaLAPACK routine. Two calls with the array descriptors DESCA for coefficient matrix and DESCB for load vectors as follows;

DESCINIT (DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, MXLLDA, INFO)

where *M×N* coefficient matrix with a leading dimension of *MXLLDA* is distributed by *MB×NB* subblocks starting from processor row *RSRC* and processor column *CSRC*.

DESCINIT (DESCB, N, NRHS, NB, NBRHS, RSRC, CSRC, ICTXT, MXLLDB, INFO)

where $N \times NRHS$ load vectors with a leading dimension of $MXLLDB$ is distributed by $NB \times NBRHS$ subblocks starting from processor row $RSRC$ and processor column $CSRC$.

After the assignment of array descriptors, algorithm executes one of two parallel solvers that are used to solve the interface system of equations depending on the nature of the coefficient matrix A. It is recommended that banded solver should be used for the systems that has $BW = 0.1N$ to achieve an optimum performance. Interface system solver use this condition to prefer the solver to be used [10]. Main features of the block solvers that is implemented in the framework is as follows;

- **PDPOSV**; computes the solution to a real system of linear equations

$$Ax = B \tag{4.14}$$

  where A is an N×N symmetric distributed positive definite matrix and x and B are N×NRHS distributed matrices. NRHS denotes the number of right hand sides. The Cholesky decomposition is used to factor sub(A) as

$$A = U^T U \quad \text{or} \quad A = LL^T \tag{4.15}$$

  The factored form of A is then used to solve the system of equations. Besides, this routine requires square block decomposition ($MB = NB$).

- **PDPBSV**; solves a system of linear equations

$$Ax = B \tag{4.16}$$

  where A is an N×N real, banded symmetric positive definite distributed matrix with bandwidth BW. Cholesky factorization is used to factor a reordering of the matrix into LL^T. This routine has three restrictions; distribution can not be cyclic manner, block size can not be too small and has an proper alignment between coefficient matrix and load vectors to prevent poor performance of routine.

4.4.2.2 Improvements for Speed-up

As the parallel dense solver library ScaLAPACK [10] is implemented upon various libraries therefore its efficiency mainly depends on the efficiency of the low-level libraries like BLAS etc. Since the BLAS and the BLACS form a low-level interface between ScaLAPACK software and different computer architectures, their performance highly

affects the performance of ScaLAPACK. The efficient implementations of the BLAS and the BLACS are provided by computer vendors (or others) for their own computer hardwares. By using libraries supplied by vendors for existing hardware, efficiency of ScaLAPACK from this aspect can be enhanced.

In addition to the low-level library efficiencies, runtime parameters such as block size for block cyclic distribution and the configuration of the processor grid etc. can be considerably effective on the performance of a ScaLAPACK routine. Suppose that there are P processors available. It is recommended to use one dimensional processor grid ($P_{row} = 1$ and $P_{col} = P$) if $P < 9$ in [10]. Block size for block cyclic distribution is another important parameter in efficiency of a ScaLAPACK routine. ScaLAPACK uses algorithmic block size equal to distribution block size, which means it executes submatrices with the same size which they are distributed over a PC cluster. Smaller block sizes will cause an increase in number of data transfers among processors. Therefore, communication overhead increases due to message start-up latencies (refer to Appendix A). On the otherhand, larger block sizes may require larger memory during submatrix computations like matrix-matrix multiplications and this requirement may exceed the cache memory of CPU. This situation may result in considerable efficiency reduction for submatrix computations. Therefore, recommended block size range for ScaLAPACK routines is between 32 and 128 [1].

4.4.2.3 Execution Time and Communication Volume Estimations

In large dense linear algebra computations, the computation cost generally dominates the communication cost. To notice such observations and any bottlenecks, time spent at every step of a parallel algorithm should be investigated carefully. The time to execute one floating-point operation by one processor is denoted by t_f. The time to communicate a message between two processors is approximated by a linear function of the number of items communicated. The function is the sum of the time to prepare the message for transmission (t_m) and the time taken by the message to traverse the network to its destination, that is, the product of its length by the time to transfer one data item (t_v). Alternatively, t_m is also called the start-up latency, since it is the time to communicate a message of zero length. Detailed information about these parameters and obtained results for them are presented in Appendix A.

The total execution time of a parallel algorithm is sum of the time required for computation, communication and coordination (refer to Appendix A). The total number

of floating-point operations of a ScaLAPACK [10] routine is computed by $C_f N^3$where N is matrix size and C_f is a characteristic constant depending on routine. Thus, time spent for computation can be computed as

$$t_{comp} = \frac{C_f N^3}{P} t_f \tag{4.17}$$

where P is the number of available computers.

Similarly, the total number of data items communicated is expressed as$\frac{C_v N^2}{\sqrt{P}}$ where

$$C_v = 4 + log_2 P \tag{4.18}$$

Therefore, the time spent for the communication of data can be computed as

$$t_{comm} = \frac{C_v N^2}{\sqrt{P}} t_v \tag{4.19}$$

Finally, the total number of messages during the execution of a ScaLAPACK routine is given by $\frac{C_m N}{NB}$ where NB is block size for block cyclic distribution and

$$C_m = 2 + \frac{log_2 P}{2} \tag{4.20}$$

Thus, time required for the coordination of messages is

$$t_{coor} = \frac{C_m N}{NB} t_m \tag{4.21}$$

Thus, total parallel execution time is given as the summation of t_{comp}, t_{comm}, and t_{coor};

$$T_P = \frac{C_f N^3}{P} t_f + \frac{C_v N^2}{\sqrt{P}} t_v + \frac{C_m N}{NB} t_m \tag{4.22}$$

where T_P designates the total parallel execution time of the algorithm by utilizing P processors.

4.4.3 Test Results

Implemented interface system solution algorithm is tested on two different PC Clusters. First one, HPCE, is composed of eight identical computers which have Intel P4 3.2 GHz processors and 1 GB RAMs. On the otherhand, second cluster, HPCE2, is composed of eight identical computers which have Intel Core2Quad Q9300@2.5 GHz processors and 3.23 GB RAMs. Intel Core2Quad family processors involve four processors that are theoretically working at 2.5 GHz and are able to share the memory. However, this

feature of this cluster is not utilized during these tests. Both of these clusters are connected with ordinary 1 GBit network switches and all computers at both clusters are running Windows XP Professional. Besides, communication, coordination and computation parameters measured for them are presented in Appendix A. Structural models analyzed by the implemented algorithms are introduced in Appendix B.

In substructure based methods, after the substructuring, the contributions of each substructure to the interface system are condensed. Then, the condensed substructure stiffness matrices are assembled and equilibrium equations are solved for the displacements at the interface nodes. Therefore, to inspect the performance of the interface system solution timings for assembly and solution are measured separately.

4.4.3.1 Performance of The Interface System Assembly

Table 4.5 presents the interface matrix sizes for three different test models and assembly timings of these interface systems on HPCE2 cluster. As mentioned previously, increasing number of computers generally increase the number of interface nodes so the number of interface system of equations. Table 4.5 verifies this situation.

Table 4.5: Interface System Assembly Timings for HPCE2

# of Computers	Interface System Assembly					
	Cube30		GroM		StRegis	
	Matrix Size	Assembly (s)	Matrix Size	Assembly (s)	Matrix Size	Assembly (s)
2	3738	2.7	2748	1.2	2772	2.5
4	5598	2.2	8472	5.4	5598	2.4
6	7407	3.4	8796	4.3	10716	2.3
8	9726	3.9	11676	4.6	14544	3.0

When the assembly timings are considered, they are fluctuating irregularly, although matrix sizes are increasing regularly (Table 4.5). This may be reasonable if interaction between assembly algorithm and test models are considered in detail. As mentioned before, assembly algorithm is omitting the zero terms because they designate that no contribution exists for a selected dof from this substructure. Therefore, structural formation of test model can affect the efficiency of the algorithm. For example, if a test model is composed of substructures which form more zero contributions after condensation step, assembly of the interface system for this model may not require excessive time for data communication. Therefore, assembly is completed faster.

Another reason for fluctuating assembly timings is that data compression is an unpredictable procedure because it depends on the overall similarity of data compressed. Since, data compression rate and speed is open to fluctuation, most time consuming step, communication, of interface system assembly is completed in an unpredictable time.

4.4.3.1.1 Performance Improvement by Runtime Data Compression

Table 4.6 presents the speed-up obtained by runtime data compression during interface system assembly of Cube30 model on HPCE2 cluster. As it can be seen, runtime data compression reduces the assembly times approximately 30 %.

Table 4.6: Speed-up obtained by Runtime Data Compression

# of Computers	Matrix Size	Assembly Times (s)		Speed-up
		Without Compression	With Compression	
2	3738	3.2	2.7	1.19
4	5598	2.8	2.2	1.28
6	7407	4.8	3.4	1.41
8	9726	5.3	3.9	1.37

4.4.3.2 Performance of The Interface System Solution

This section presents a discussion on results about the interface system solution algorithm (PDPOSV routine of ScaLAPACK [10]) obtained by utilizing test models described in Appendix B. In first subsection, execution time and communication volume estimations of solution algorithm are presented and compared with the actual results. On the otherhand, in second subsection, speed-up and efficiency values obtained as a result of test are presented and discussed in detail.

4.4.3.2.1 Execution Time Estimation and Comparison with Actual Timings

Execution time and communication volume estimation results of PDPOSV routine of ScaLAPACK [10] for three different structural models, GroM, Cube30, StRegis are obtained on HPCE and HPCE2 clusters.

To illustrate the procedure, results on four computers of HPCE is given in Table 4.7. Block size used for block cyclic distribution is 32. Measured parameters like communication speed, time per data item communicated t_v, time per floating-point operation t_f and time per message t_m are presented in the first row of parameters section (refer to

Appendix A). C_f is given as $\frac{1}{3}$ for PDPOSV routine of ScaLAPACK [10]. Rest of the parameters like C_v and C_m are computed by using Equation 4.18 and Equation 4.20 respectively.

Table 4.7: Execution time and communication volume estimations on 4 computers of HPCE

Models	Parameters									
	Comm (Mbit/s)			**tf**		**tv**			**tm (s)**	
	5.556E+08			4E-10		1.15E-07			0.00002	
	Block Size			**Cf**		**Cv**			**Cm**	
	32			0.33		6.00			3	
	Results									
	Size	**Total Operation Count**	**Time for Comp.**	**Comm. Data Count**	**Time for Comm.**	**Message Count**	**Time for Coor.**	**Prediction (s)**	**Actual (s)**	**Error**
Cube30	6528	9.27E+10	9.27	1.3E+08	14.73	612	0.01	24.01	23.38	3%
GroM	9036	2.46E+11	24.59	2.4E+08	28.22	847	0.02	52.83	52.33	1%
StRegis	6276	8.24E+10	8.24	1.2E+08	13.61	588	0.01	21.86	24.07	9%

Similarly, results on four computers of HPCE2 is given in Table 4.8. Note that interface matrix sizes are different because the data preparation step partitions the structure according to cluster properties.

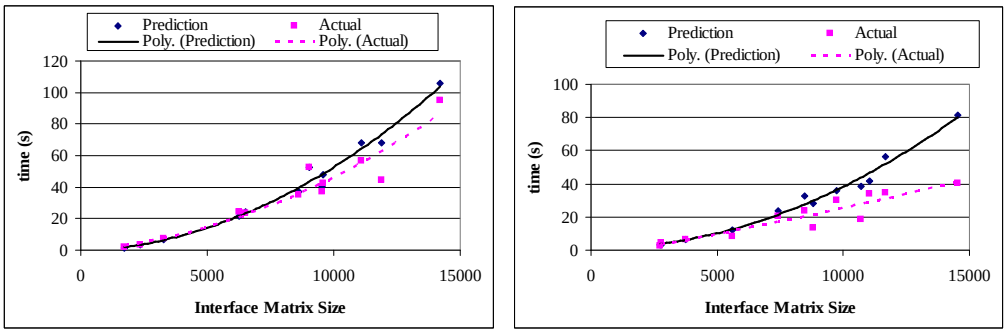
Table 4.8: Execution time and communication volume estimations on 4 computers of HPCE2

Models	Parameters									
	Comm (Mbit/s)			**tf**		**tv**			**tm (s)**	
	8.073E+08			3.125E-10		7.93E-08			0.000006	
	Block Size			**Cf**		**Cv**			**Cm**	
	32			0.33		6.00			3	
	Results									
	Size	**Total Operation Count**	**Time for Comp.**	**Comm. Data Count**	**Time for Comm.**	**Message Count**	**Time for Coor.**	**Prediction (s)**	**Actual (s)**	**Error**
Cube30	7407	1.4E+11	10.58	1.6E+08	13.05	694	0.00	23.64	20.34	16%
GroM	8472	2.0E+11	15.84	2.2E+08	17.07	794	0.00	32.91	23.49	40%
StRegis	5598	5.8E+10	4.57	9.4E+07	7.45	525	0.00	12.02	8.57	40%

Parallel execution timing estimation is sum of the t_{comp}, t_{comm}, and t_{coor}. Therefore, it is calculated by using Equation 4.22. Similarly, execution time and communication volume estimation results are obtained from the solution of structural models on 2, 4, 6, and 8 computers. Then, interface solution prediction and actual timings in Figure 4.11 are obtained for both HPCE and HPCE2 clusters.

For HPCE cluster, interface solution predictions are fairly consistent with the actual

timings (Figure 4.11a). However, for HPCE2 actual timings are considerably less than expected (Figure 4.11b). This situation can be expected when it is considered that HPCE2 cluster is composed of computers with 4 processors. Therefore, communication and computation can be overlapped because of the non-blocking communication structure of ScaLAPACK library [10].



(a) for HPCE (b) for HPCE2

Figure 4.11: Interface Solution Prediction vs Actual Timings

4.4.3.2.2 Speed-up and Efficiency

For interface system solution, block cyclic full matrix solver (PDPOSV) of ScaLAPACK [10] is utilized. To measure the performance of this algorithm *speed-up* and *efficiency* are investigated. With the assumption of sequential execution time equals to the time required for the execution of the total floating-point operations on a single computer, following formula can be derived from Equation 4.17 as

$$T_{seq} = C_f N^3 t_f \tag{4.23}$$

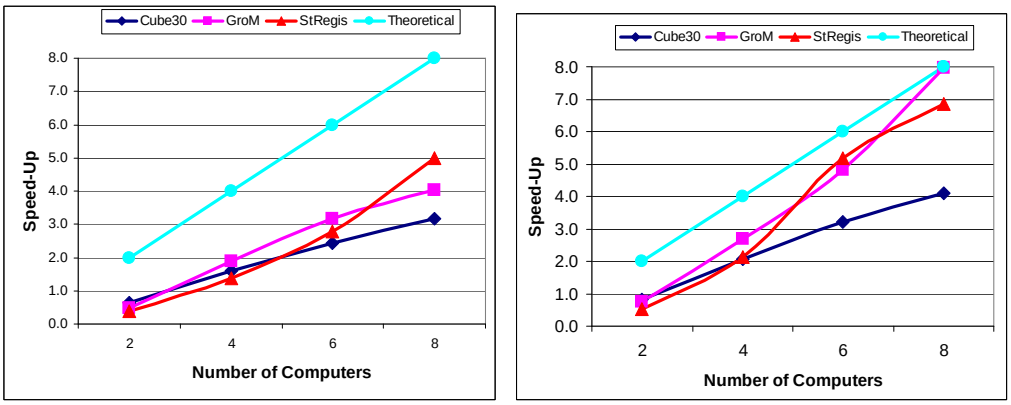
Thus, the speed-up obtained with this parallel algorithm can be estimated as

$$S_P = \frac{T_{seq}}{T_P} \tag{4.24}$$

By utilizing the above expression, speed-up graphs for HPCE and HPCE2 are obtained (Figure 4.12). Both graphs shows that obtained speed-up ratios are increasing with the increasing number of computers. Another important note can be the speed-up ratios obtained for solutions by 2 computers. As it can be noticed, instead of obtaining a speed-up, execution time is increased when 2 computers were utilized. For example, for StRegis model, the interface system has 1704 equations. Since the time required for the coordination and communication of data transfers (refer to Appendix A) were

larger than time required for the solution of whole interface system, parallel solution was slower than solution on a single computer.

As presented in Figure 4.12b, for the solution of interface system of StRegis model on eight computers of HPCE2 cluster, the theoretical speed-up ratio, 8 is almost obtained and similarly, speed-up ratios for HPCE2 cluster is slightly greater than the ones for HPCE cluster. Since HPCE2 cluster is composed of multi-processor computers, communication and computation can be overlapped, high speed-up ratios are observed.



(a) for HPCE (b) for HPCE2

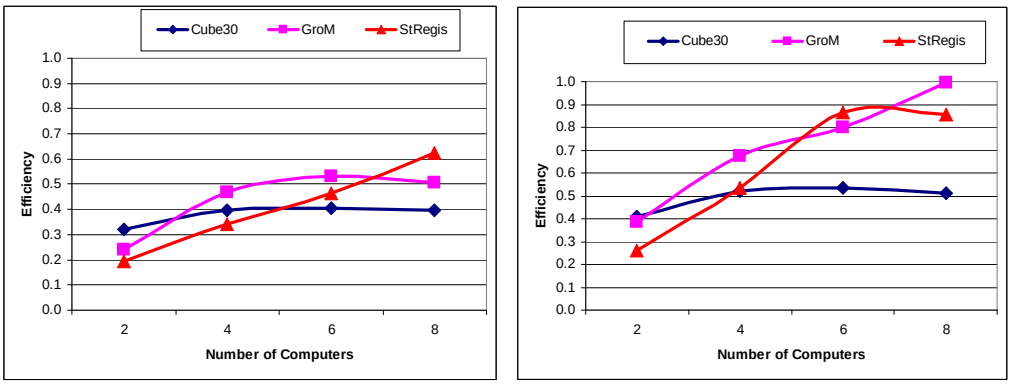
Figure 4.12: Interface Solution Speed-Up Graphs

The efficiency of the interface solution by utilizing PDPOSV routine of ScaLAPACK [10] can be computed by

$$E_P = \frac{1}{P}\frac{T_{seq}}{T_p} \tag{4.25}$$

Efficiency graphs for HPCE and HPCE2 are presented in Figure 4.13. As it can be noticed, generally efficiency of the interface system solution algorithm (PDPOSV routine of ScaLAPACK [10]) was increasing up to the point where 8 computers were utilized. In all cases, the number of equations in interface system increase as the number of computers increase. Because of the increased number of equations, the solution of interface system was dominated by the data communications. Thus, computers stayed idle instead of solving the equations.

Similar to speed up graphs, HPCE2 produced more efficiency over HPCE cluster. As mentioned, HPCE2 cluster is composed of multi-processor computers that overlapped the communication and computation. Thus, efficiency values obtained on HPCE2 cluster is greater than HPCE cluster which is composed of single processor computers.



(a) for HPCE (b) for HPCE2

Figure 4.13: Interface Solution Efficiency Graphs

CHAPTER 5

OVERALL PERFORMANCE OF PARALLEL SOLUTION FRAMEWORK

The performance of the presented solution strategy were investigated by solving different structural models on different numbers of computers of two different homogeneous clusters. First one, HPCE, is composed of eight identical computers which have Intel P4 3.2 GHz processors and 1 GB RAMs. On the otherhand, second cluster, HPCE2, is composed of eight identical computers which have Intel Core2Quad Q9300@2.5 GHz processors and 3.23 GB RAMs. Intel Core2Quad family processors involve four processors that are theoretically working at 2.5 GHz and are able to share the memory. However, this feature of this cluster is not utilized during these tests. Both of these clusters are connected with ordinary 1 GBit network switches and all computers at both clusters are running Windows XP Professional. Besides, communication, coordination and computation parameters measured for them are presented in Appendix A.

5.1 Method

Three different structural models (Cube30, GroM, and StRegis) which are introduced in Appendix B are analyzed by solution framework implemented. Two, four, six, and eight computers of clusters are utilized. All timings for parallel algorithms are measured by using MPI [54] timing tools and all timings for sequential algorithms are measured by using frequency performance tools of WinAPI [74].

Data preparation timings, local solution timings and total solution timings were measured during the test runs. Data preparation timings include the initial partitioning, workload balancing iterations and the preparation of input data for the parallel solution. Local solution time is composed of assembly and condensation timings. Total parallel

solution time involves data preparation or equation ordering, stiffness matrix generation and assembly, equation solution, and nodal displacement calculations. The time spent during reading the structural database from the input file and initial distribution of nodes and elements among computers were excluded in the total solution time. Speed-up and efficiency graphs are based on the estimated serial solution times computed by dividing the total operation count to computational speed of computer.

5.2 Test Results

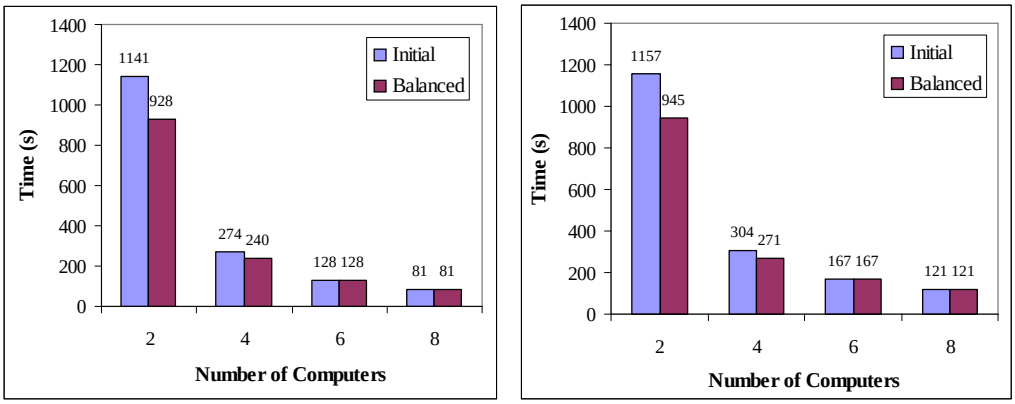
This section presents the overall test results for solution framework on three different models and two different cluster. Each model will be discussed in a separate subsection and finally a simulation test result will be given to discuss the effects of communication on this solution framework.

5.2.1 Cube30

Cube30 is the mathematical model of an imaginary cube (Figure B.1) that has 27,000 8-node brick elements with 29,791 nodes and 89,364 equations. This model represents a symmetric and uniform model composed of same type of elements with significantly large bandwidth.

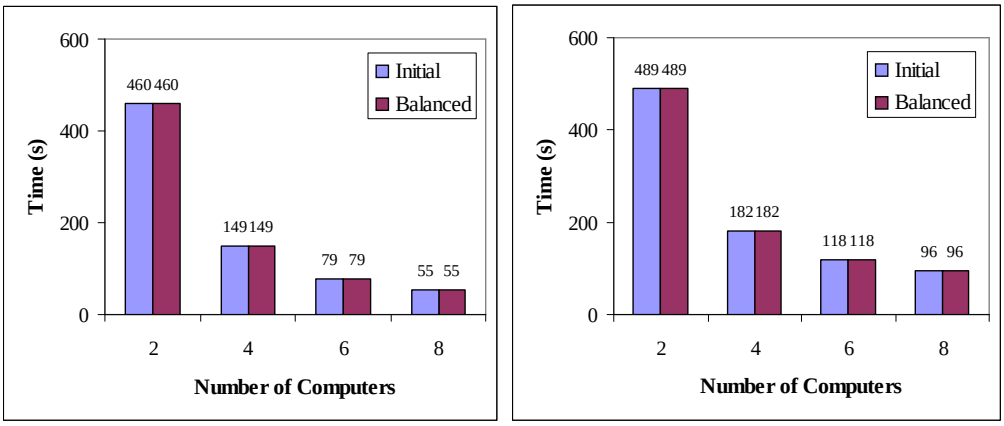
Figure 5.1 presents the local solution (assembly and condensation) timings and total solution times of Cube30 model obtained by HPCE cluster. The parallel solution with balanced substructures was faster than the parallel solution with the initial substructures up to six processors. Improvement obtained by the workload balancing is decreasing when the number of substructures increase. For example, for the solution with two substructures, the workload balancing steps decreased the local solution time by 213 seconds (∼18%) and for the solution with four substructures reduction was equal to 34 seconds (∼12%). As more than four computers utilized for solution, no improvement was obtained by workload balancing. Since this model represents a symmetric and uniform model composed of same type of elements, initial partitioning mostly produced the best partitioning and workload balancing iterations were terminated after a few iterations. Moreover, as the size of the substructures decreases the assembly time starts to govern the local solution time where initial balancing algorithms can successfully balance the workloads for such cases. Because of this reason workload balancing iterations did not improve the timings for HPCE2 cluster at all as demonstrated in Figure 5.2.

During the tests, the number of interface system equations almost remained same for Cube30 model, so, the decrease in local solution time was equal to the decrease in total solution time for solutions for two and four computers.



(a) Local Solution Timings (b) Total Solution Timings

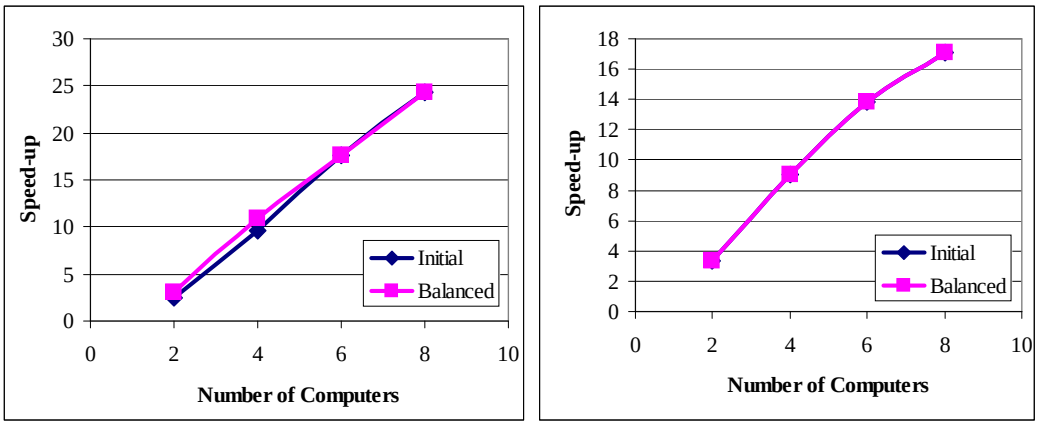
Figure 5.1: Test Results for Cube30 Model on HPCE cluster



(a) Local Solution Timings (b) Total Solution Timings

Figure 5.2: Test Results for Cube30 Model on HPCE2 cluster

The serial solution time for this model was estimated as 2943 and 1639 seconds for HPCE and HPCE2 clusters, respectively. By using the estimated serial solution time of this model speed-up graphs which are presented in Figure 5.3 were obtained. Ideally, speed-up values can not exceed the number of computers utilized. However, for all solutions of this model, super speed-ups were observed. This is mainly because the substructuring not only decreased the size but also the bandwidth of the local problem. Although the interface problem size also increases as the number of substructures increases, the solution time was mostly governed by the local solution.



(a) on HPCE Cluster (b) on HPCE2 Cluster

Figure 5.3: Speed-ups for Cube30 Model

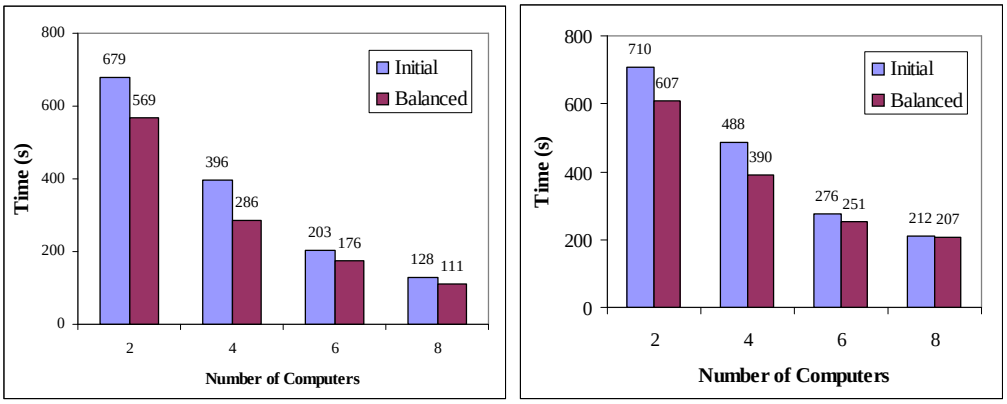
5.2.2 GroM

GroM is an actual structural model generated for modelling a nuclear waste plant (Figure B.2) that has 2,811 frames and 43,776 quadrilateral shell elements with 46,587 nodes and 233,616 equations. This model is composed of both 1D and 2D elements but the connectivity among 1D and 2D elements is limited. The structure is geometrically symmetric in two horizontal directions. The bottom box consists of many small size rooms whose walls are also modelled with finite elements. This situation significantly increases the bandwidth of the structural stiffness matrix of this model.

Figure 5.4 and Figure 5.5 present the local solution (assembly and condensation) timings and total solution timings of GroM model obtained by HPCE and HPCE2 clusters, respectively. For HPCE cluster, the parallel solution with balanced substructures was always faster than parallel solution with the initial substructures. For example, ∼16%, ∼27%, ∼13%, and ∼14% improvements in local solution timings were obtained for the tests with two, four, six, and eight computers. These improvements on local solution timings were reflected to the total solution timings as ∼15%, ∼20%, ∼9%, and ∼9%. On the otherhand, for HPCE2 cluster, workload balancing did not produce a better partitioning result only in one case which is the solution with eight computers. Because, in this test, the number of interface nodes increased considerably (∼15000 nodes) so any improvement for local solution could not improve the total solution timing.

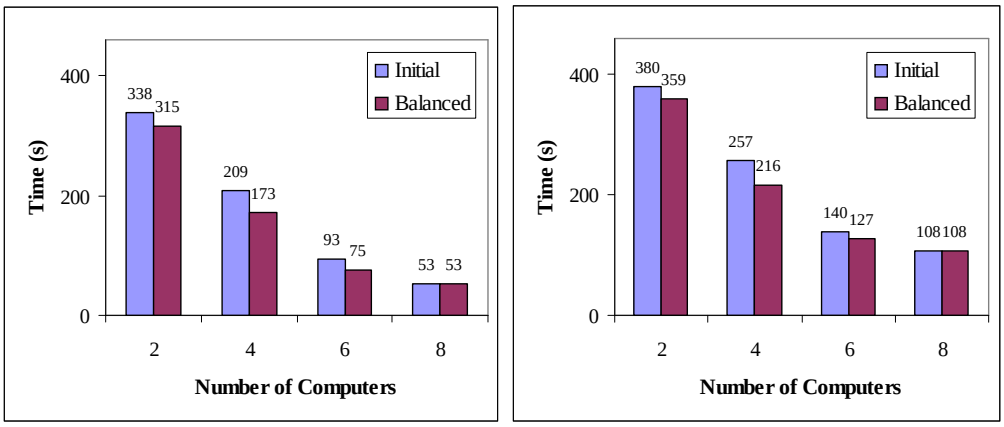
For both clusters improvement obtained by workload balancing is decreasing when the number of substructures increases because of the increase in the number of interface

nodes. Improvement obtained for local solution was reduced because of the increase in interface system solution timings.



(a) Local Solution Timings (b) Total Solution Timings

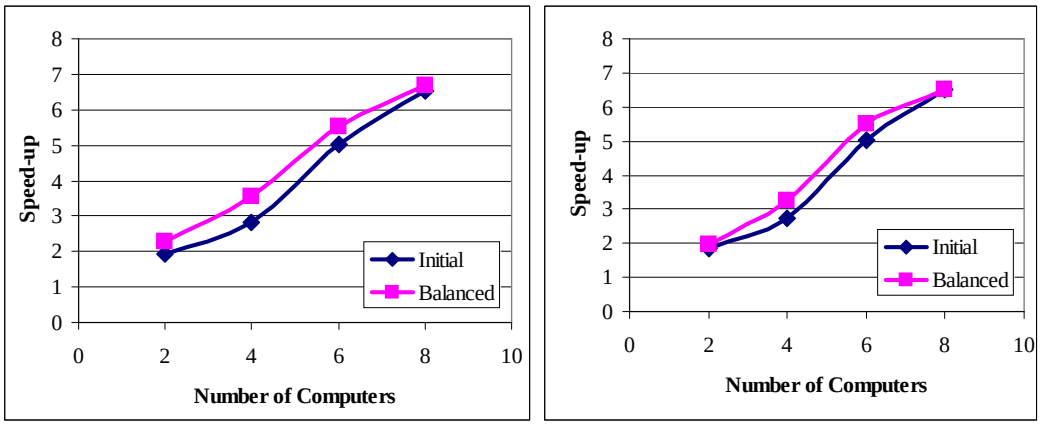
Figure 5.4: Test Results for GroM Model on HPCE cluster



(a) Local Solution Timings (b) Total Solution Timings

Figure 5.5: Test Results for GroM Model on HPCE2 cluster

The serial solution time for this model was estimated as 1383 and 701 seconds for HPCE and HPCE2 cluster computers, respectively. By using the estimated serial solution time of this model, speed-up graphs which are presented in Figure 5.6 were obtained. 1.95, 3.24, 5.51, and 6.51 speed-up values obtained for the solutions by two, four, six, and eight computers of HPCE2. For HPCE cluster, similar speed-up values were obtained. In only one case, which is the solution with two computers, super speed-up of 2.2 was observed. This is mainly because the substructuring not only decreased the size but also the bandwidth of the local problem.



(a) on HPCE Cluster (b) on HPCE2 Cluster

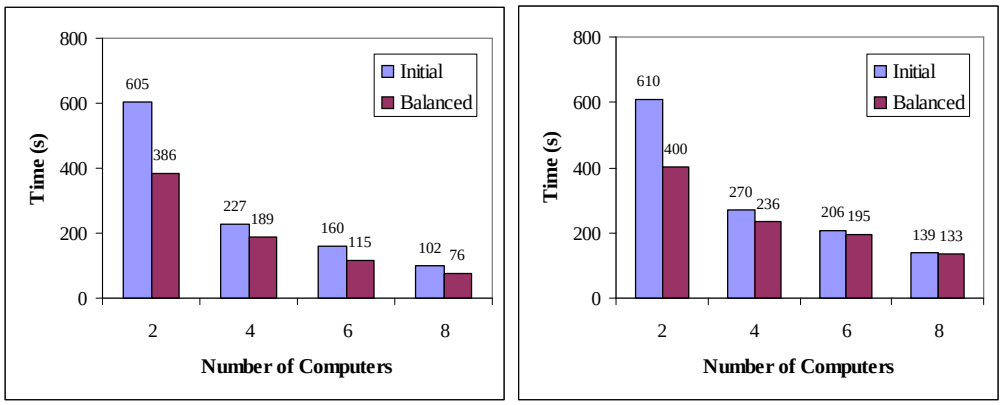
Figure 5.6: Speed-up Graphs for GroM Model

5.2.3 StRegis

StRegis is another actual model of a 30 storey high rise building (Figure B.3) that was prepared by a structural engineer for designing a mixed use hotel and residential building composed of flat plate slab systems with shear walls and columns. The model was composed of 6,641 frames and 86,509 both quadrilateral and triangular shell elements with 87,075 nodes and 518,580 equations. The model is highly irregular, the floor plan of lower levels are different at each level with large openings at various locations. The slabs are modelled with both triangular and quadrilateral elements with no uniformity at storey levels except for 15 middle residential levels. The slabs are connected with 1D elements that represent columns. This model is a very representative model for such reinforced concrete buildings, thus the performance of the parallel solution approaches will be a good indicator for the solution of actual civil engineering models where no symmetry and uniformity exists.

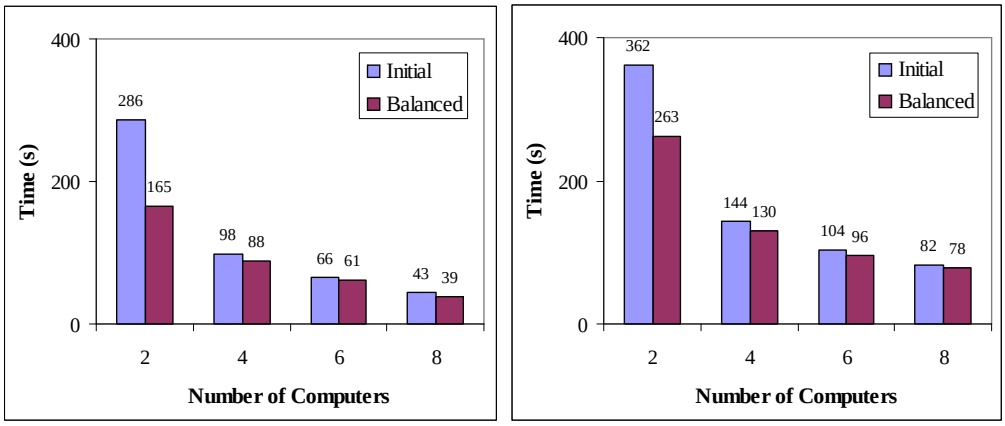
Figure 5.7 and Figure 5.8 present the local solution (assembly and condensation) timings and total solution timings of GroM model obtained by HPCE and HPCE2 clusters, respectively. For both clusters, the parallel solution with the balanced substructures was always faster than parallel solution with the initial substructures.

For HPCE cluster, total solution timings for the tests with two and four computers, are decreased by 210 seconds and 34 seconds, respectively. Similarly, for HPCE2 cluster, the total solution timings for the tests with two and four computers, are decreased by 99 seconds and 14 seconds, respectively. For the rest of the test runs workload balancing did not improve the solution times considerably.



(a) Local Solution Timings (b) Total Solution Timings

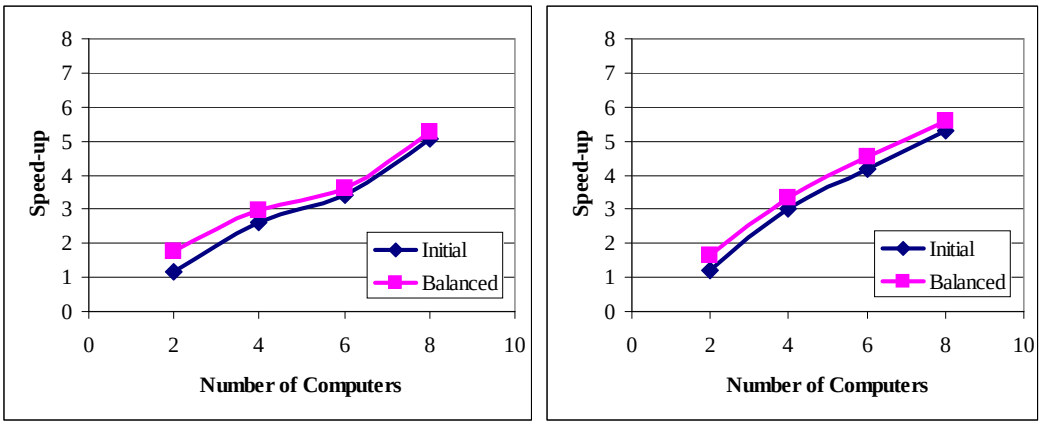
Figure 5.7: Test Results for StRegis Model on HPCE cluster



(a) Local Solution Timings (b) Total Solution Timings

Figure 5.8: Test Results for StRegis Model on HPCE2 cluster

The serial solution time for this model was estimated as 706 and 435 seconds for HPCE and HPCE2 cluster computers, respectively. By using the estimated serial solution time of this model, speed-up graphs which are presented in Figure 5.9 were obtained. 1.16, 2.61, 3.42, and 5.08 speed-up values obtained for the solutions by two, four, six, and eight computers of HPCE. For HPCE2 cluster, similar speed-up values were obtained.



(a) on HPCE cluster (b) on HPCE2 cluster

Figure 5.9: Speed-up Graphs for StRegis Model

5.2.4 Shared Memory Simulation

This subsection presents a shared memory simulation of implemented solution framework. Testing method can be summarized as using all processors of multi-processor computers. HPCE2 cluster is composed of 4 processor computers. Each testing model was partitioned as if they would have been solved in cluster. Then, these substructured models were solved by utilizing multiple processors of eight, four, two, and one computers.

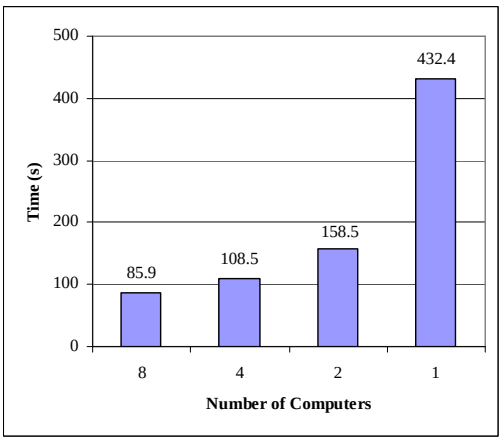
More than one copy of application can be executed in a computer by using MPICH2 [63] for simulation purposes. In previous results, only one application was executed on each computer. Each computer executes the request of this application and application handles the communications and computations. During communication, local area network was used. However, when more than one copy of application executed at the same computer, local area network was not required to transfer data between these copies because all the data required existed in the same computer. Therefore, communication overhead could be ignored between application copies.

During this simulation tests, all test models were substructured as if they would have been solved by eight computers and then eight, four, two, and one computers were utilized. In case of utilization of eight computers, which is standard cluster run, one copy of solution framework is executed on each computer. Although each computer has four processors, one processor is responsible from execution of solution framework. Ideally, the rest of the processors stay idle. However, operation system may use them for communication and coordination purposes.

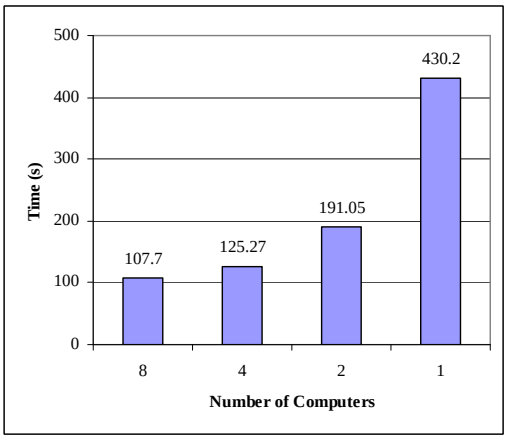
In case of utilization of four computers, two copies of solution framework is executed on each computer. Therefore, rest of the processors on each computer, ideally, stays idle.

Similarly, in case of utilization of two computers, four copies of solution framework is executed on each computer. So, all of the processors of each computer will be responsible from execution of one solution framework application.

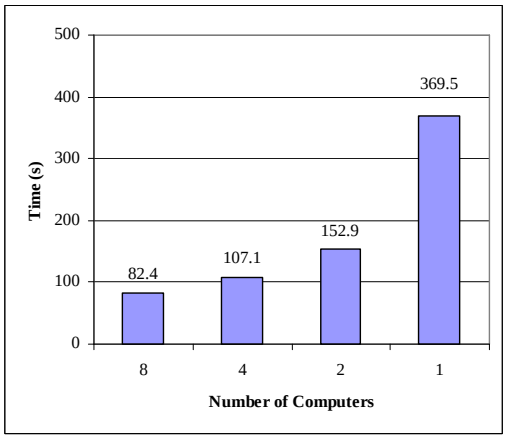
If only one computer is used for the solution of a model that is substructured into eight parts, operating system handles the execution of all requests coming from eight copies of application by utilizing all processors exists. Briefly, it can be assumed that each processor will be responsible from execution of two copies of solution framework.



(a) Cube30



(b) GroM



(c) StRegis

Figure 5.10: Total Solution Times on a Multi-processor Computer in HPCE2 Cluster

Figure 5.10 demonstrates the test results for shared memory simulation mentioned above. Results reveals two behaviors about the solution system. Except from the run

with single computer, during rest of the runs, each application copy was assigned to a processor. Therefore computational power of these systems were same. Since communication overhead would be ignored between the copies of the application executed in the same computers, it was expected to observe a decrease in total solution time as the number of computers utilized decreases. In contrast, absolutely in all tests total solution time increased as the number of computers utilized decreased.

First behaviour can be observed in tests with four and two computers by choosing eight computer utilization case as a benchmark. For these tests, total solution times were slightly larger than the solution by eight computers because network bandwidth was shared by the copies of application that are running on the same computer. In other words, although the communication among copies of application that were running in the same computer became faster, the communication among copies of application that were not running in the same computer was becoming slower.

In addition to the first one, a second behaviour can be observed for the test run with single computers. As Figure 5.10 demonstrates, total solution time for this case was significantly larger than the other test runs. For this special case, communication overhead can be assumed as non-existing. On the other hand, computational capability was lower half of it was before. Because, during rest of the tests, one processor was responsible from one copy of application. In this special case, one processor was responsible from roughly two copies of application. Thus, this fact explains the significant difference mostly. Although, it was not effective as the previous one, there was another effect that caused system to be slowed down. Since all processors used the same memory, efficiency of computation reduced. Either processors waited for available memory space or forced to use virtual memory supplied by operation system which is a virtual memory physically lies on harddisk but behave as Random Access Memory (RAM).

5.3 Discussion of Overall Results

Test runs which illustrated the applicability and efficiency of this solution framework were presented. In these tests, the number of processors was varied from two to eight to demonstrate the performance of the overall solution framework. Balancing the local solution times of substructures decreased the total parallel solution time for most of the testing models. There were some cases where although the local solution times was decreased by the workload balancing step, the interface problem size was increased

so much that there was not any gain in the total solution time. This situation most frequently occurred for solutions with six and eight processors.

The improvement obtained by workload balancing step on total solution time increases when the uniformity and the symmetry of the testing model reduces. Because, for the uniform and symmetric testing model initial substructuring produced the best partitioning.

The initial substructures had the smallest interface problem size for almost every case but the difference between the interface solution time of the initial substructures and the substructures balanced was not high. Since the total solution time was governed by the local solution time, the balanced substructures performed faster.

Data mapping on 2D block cyclic manner, communication scheme and runtime data compression is utilized to increase the efficiency of the interface system assembly and significant improvement obtained against the previous algorithm presented in [6].

Shared memory simulation test results demonstrated that this method can be utilized even on the shared memory computers or on the hybrid systems (distributed memory with shared memory) by supplying sufficient memory to the system.

The total solution time decreased as the number of processors increased. The solution produced larger speed-up as the size of the problem increased. This is mainly because the solution time of the large problems tested in this study was primarily governed by the local solution time. Thus, substructuring not only decreased the number of equations but also the profile of the stiffness matrix of the local problem. As a result, some of the speed-up values were larger than the number of processors.

The presented method is very efficient for the linear solution of large structural models on PC clusters. When the computing systems available in structural engineering offices are considered, overall, this framework is very suitable and can be utilized to solve large linear static problems in parallel without purchasing any additional hardware.

CHAPTER 6

CONCLUSION AND FUTURE PLANS

6.1 Conclusion

This study presented a substructure based parallel solution framework for the linear solution of large structural models on PC clusters. It is based on the framework proposed by Kurç [6] and mainly focuses on the improvements suggested. Every step of this solution framework, from partitioning of the structures into substructures to the computation of the internal displacements in the substructures was performed in parallel.

PC clusters were chosen to be the target parallel environment due to their availability in civil engineering design offices and their cost despite their relatively low communication speed between processors when compared with other parallel architectures. Similarly, among many existing parallel solution approaches, a substructure based solution method was chosen to be the most suitable method for this study since such methods not only decreased the communication cost but also allowed performing the stiffness and force matrix generation, assembly, and computation of element results in parallel.

One of the main challenges of substructure based parallel solution methods is to find a partitioning where the computers have balanced workloads. Otherwise, the parallel solution can be governed by the computer having the highest workload. In order to overcome such problems and thus enhance performance of the parallel solution, a data preparation step was added to the solution framework. Data preparation step is composed of preparing data for the parallel solution which involves partitioning, workload balancing, and equation numbering.

The workload balancing step decreased the local solution times in most cases. More-

over, the time spent during the iterations was insignificant when compared with the improvement in the local solution times. Thus, workload balancing method proposed by Kurç for direct condensation was fast enough to be utilized prior to the actual linear static analysis and robust enough to work with mixed structural models (1D members mixed with 2D shell elements). With this study, the efficiency of this method to decrease the total parallel solution time was increased by taking assembly times into consideration in addition to condensation times during workload balancing iterations.

The solution was transferred to the substructure interfaces after the substructure level stiffness matrices had been condensed. The condensation algorithm is a variant of a sparse solver. Condensation is the most time consuming step of the overall solution. After condensation, the interface equations were assembled. Due to the high data dependency, assembly of interface equations dominated by the data communication. Because of this reason, during this study, three improvements are utilized at this step. First one is mapping the data on 2D block cyclic scheme, so requirement for any assembly on a single computer is avoided and bandwidth is used more efficiently. Second one is forming a communication scheme so that computers do not wait busy computers and directed to idle ones for communication. Third one is the runtime data compression to reduce the communication volume so enhance the speed of the assembly step. After the interface system assembly, interface system is solved in parallel to complete the solution. The interface solution was performed by parallel dense and banded solver routines of ScaLAPACK library [2]. By utilizing the improvements to the interface assembly step and parallel block solver for the interface system solution, the time spent during the overall solution of interface system is significantly improved when the timings obtained in the prior study [6] are considered.

To investigate the performance of the solution framework, various structural models were solved on two different PC clusters. The results demonstrated that balancing the condensation times of substructures also decreased the total solution time in most cases. For most of the problems, the total solution time decreased as the number of processors increased. Moreover, substructuring not only decreased the number of equations but also the profile of the substructures' stiffness matrices resulting in speed-up values greater than the number of processors.

As a conclusion, this study presented the development of a parallel substructure based solution framework designed for the efficient linear static analysis of large structures. The framework can be applied efficiently to models which contain a mixture of

element types and which are having any symmetry.

6.2 Future Plans

There are many extensions and improvements which can be made to the presented framework that would increase its efficiency and functionality. Some of these are discussed below:

- **New condensation algorithm:** Since, existing condensation algorithm is not an optimized algorithm, it can be replaced by a new condensation algorithm that is optimized for efficient memory usage and computation by CPU. In literature, multifrontal condensation algorithms that can utilize dense block solvers exists.

- **Shared Memory Support:** Today, almost all of the computers in the market have multi-processors so shared memory support at any level of the algorithm may increase the performance of the solution framework. For example, stiffness matrix generation and assembly, condensation, interface system assembly and solution and also stress, strain recoveries after solution can be enhanced by utilizing shared memory usage.

- **Solution with multiple right hand sides:** This framework can also be utilized to solve problems having multiple right hand sides. In the structural models of buildings which will be built in earthquake regions, many loading conditions may arise. In this case, the substructure level and the interface stiffness matrices are factorized once, and the load factorizations are performed for each right hand side vector.

- **Support for heterogeneous computing environment:** Civil engineering design offices may have computers that have varying computational speeds. Currently, data preparation step is supporting heterogeneous computing environments. During the workload balancing iterations, a scaling variable added to imbalance factor calculation, to represent the relative computational speeds of computers. This way, the resulting substructures will have the condensation time ratios closer to their processor's computational speed ratios. Additional study is required to support heterogeneous computing environments during interface system assembly and solution. For example, the subblocks of the interface stiff-

ness matrix can be distributed to the processors according to their computational speeds. In other words, faster processors would store more columns and rows.

- **Support for GPU computing:** Graphical cards placed in computers have specialized computing units for linear algebraic computations because rendering and visualization are based on matrix transformations and multiplications. This readily available computational power can be utilized to increase the performance of the solution framework.

REFERENCES

[1] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," *SC Conference* **0**, p. 5, 1996. xii, 10, 11, 12, 58, 60, 68

[2] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. Whaley, "Lapack working note 80: The design and implementation of the scalapack lu, qr, and cholesky factorization routines," tech. rep., Knoxville, TN, USA, 1994. xii, 9, 13, 24, 27, 59, 60, 88

[3] G. Karypis and V. Kumar, *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*, 1998. xii, 16, 19, 35, 36, 37

[4] P. Ivanyi, *Parallel, Distributed and Grid Computing for Engineering*, Saxe-Coburg Publications, City, 2009. 1

[5] H. Sutter, "The concurrency revolution," *C/C++ Users Journal* **23,2**, pp. 54–62, February 2005. 1

[6] Ö. Kurç, *A Substructure Based Parallel Solution Framework For Solving Linear Systems With Multiple Loading Conditions.* PhD thesis, Georgia Institute of Technology, 2005. 2, 3, 18, 19, 22, 23, 24, 86, 87, 88

[7] J. Stoer, *Introduction to Numerical Analysis*, Springer-Verlag, Berlin, 2002. 3

[8] F. L. B. Ribeiro and I. A. Ferreira, "Parallel implementation of the finite element method using compressed data structures," *Computational Mechanics* **41**, pp. 31–48, December 2007. 4

[9] M. Silva, "Sparse matrix storage revisited," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pp. 230–235, ACM, (New York, NY, USA), 2005. 4

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK user's guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. 5, 10, 49, 58, 66, 67, 68, 69, 71, 72, 73, 74, 103

[11] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst, *Numerical Linear Algebra for High Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. 5, 10, 14, 15

[12] N. I. M. Gould, J. A. Scott, and Y. Hu, "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations," *ACM Trans. Math. Softw.* **33**(2), p. 10, 2007. 6, 14

[13] E. Rothberg and A. Gupta, "An evaluation of left-lookikng, right-looking and multifrontal approaches to sparse cholesky factorization on hierarchical memory machines," tech. rep., Stanford, CA, USA, 1991. 9

[14] D. Irony, G. Shklarski, and S. Toledo, "Parallel and fully recursive multifrontal supernodal sparse cholesky," *Future Generation Computer Systems* **20**, pp. 425–440, 2004. 9, 17

[15] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent, and X. S. Li, "Analysis and comparison of two general sparse solvers for distributed memory computers," *ACM Trans. Math. Softw.* **27**(4), pp. 388–421, 2001. 10, 14, 15, 16, 17

[16] O. Schenk and K. Gärtner, "Two-level dynamic scheduling in pardiso: improved scalability on shared memory multiprocessing systems," *Parallel Comput.* **28**(2), pp. 187–197, 2002. 10, 14

[17] O. Bessonov, D. FougÃ¨re, K. D. Quoc, and B. Roux, "Methods for achieving peak computational rates for linear algebra operations on superscalar RISC processors," in *Parallel Computing Technologies*, *Lecture Notes in Computer Science* **1662**, pp. 756–757, Springer Berlin / Heidelberg, 1999. 10

[18] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK's user's guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. 10

[19] J. Dongarra and L. S. Blackford, "Scalapack tutorial," in *PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, pp. 204–215, Springer-Verlag, (London, UK), 1996. 12, 60

[20] P. R. Amestoy, I. S. Du, and J. Koster, "Mumps: A general purpose distributed memory sparse solver," in *In Proc. PARA2000, 5th International Workshop on Applied Parallel Computing*, pp. 122–131, Springer-Verlag, 2000. 14, 16

[21] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.* **31**(3), pp. 302–325, 2005. 16

[22] G. Z. M. Berglund and S. W. de Leeuw, "A study into the feasibility of using two parallel sparse direct solvers for the helmholtz equation on linux clusters: Research articles," *Concurr. Comput. : Pract. Exper.* **18**(7), pp. 749–769, 2006. 16, 17

[23] A. Guermouche, J.-Y. L'Excellent, and G. Utard, "Impact of reordering on the memory of a multifrontal solver," *Parallel Computing* **29**(9), pp. 1191–1218, 2003. 16

[24] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications* **17**(4), pp. 886–905, 1996. 16

[25] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, "Multifrontal parallel distributed symmetric and unsymmetric solvers," *Comput. Methods Appl. Mech. Eng.* **184**, pp. 501–520, 2000. 16

[26] J. Schulze, "Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods," *BIT* **41**, p. 2001, 2001. 16

[27] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *HPCN Europe 1996:*

Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, pp. 493–498, Springer-Verlag, (London, UK), 1996. 16

[28] M. Berger and S. Bokhari, "A partitioning strategy for non-uniform problems on multiprocessors," **C-36**, pp. 570–580, May 1987. ICASE Report No. 85-55, 1985. 18

[29] M. T. Jones and P. E. Plassmann, "Computational results for parallel unstructured mesh computations," *Computing Systems in Engineering* **5**, pp. 297–309, 1994. 18

[30] H. D. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering* **2**, pp. 135–148, 1991. 18

[31] C. Farhat, "A simple and efficient automatic fem domain decomposer," *Computers and Structures* **28**, pp. 579–602, 1988. 18

[32] J. G. Malone, "Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers," *Comput. Methods Appl. Mech. Eng.* **70**(1), pp. 27–58, 1988. 18

[33] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws," *J. Parallel Distrib. Comput* **47**, pp. 139–152, 1997. 18

[34] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal* **23**(98), pp. 298–305, 1973. 18

[35] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal* **49**(1), pp. 291–307, 1970. 18

[36] L. Oliker and R. Biswas, "Plum: Parallel load balancing for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing* **52**, pp. 150–177, 1998. 19

[37] H. D. Simon, A. Sohn, and R. Biswas, "Harp: a dynamic spectral partitioner," *J. Parallel Distrib. Comput.* **50**(1-2), pp. 83–103, 1998. 19

[38] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.* **7**(2), pp. 279–301, 1989. 19

[39] Y. Yang and S. Hsieh, "Iterative mesh partitioning optimization for parallel non-linear dynamic finite element analysis with direct substructuring," *Computational Mechanics* **28**(6), pp. 456–468, 2002. 19

[40] C. Walshaw and M. Cross, "Mesh partitioning: a multilevel balancing and refinement algorithm," *SIAM J. Sci. Comput* **22**, pp. 63–80, 2000. 19

[41] Ö. Kurç and K. Will, "An iterative parallel workload balancing framework for direct condensation of substructures," *Computer Methods in Applied Mechanics and Engineering* **196**(17-20), pp. 2084 – 2096, 2007. 19

[42] S. K. Karypis, George and V. Kumar, *PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1*, 2003. 19, 35, 37, 40

[43] I. S. Duff and J. A. Scott, "The use of multiple fronts," in *in Gaussian Elimination, Proc. Fifth SIAM Conf. on Applied Linear Algebra, SIAM*, pp. 567–571, 1994. 20

[44] C. Farhat, E. Wilson, and G. Powell, "Solution of finite element systems on concurrent processing computers," *Engineering with Computers* **2**(3), pp. 157–165, 1987. 20

[45] P. Bjørstad, J. Brækhus, and A. Hvidsten, "Parallel substructuring algorithms in structural analysis, direct and iterative methods," in *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, R. Glowinski, Y. A. Kuznetsov, G. Meurant, J. Périaux, and O. B. Widlund, eds., pp. 321–340, SIAM, (Philadelphia), 1991. 20

[46] J. W. B. Jr. and S. K. Sharma, "Evaluation of distributed finite element algorithms on a workstation network," *Engineering with Computers* **10**, pp. 45–62, March 1994. 21

[47] R. E. Fulton and P. S. Su, "Parallel substructure approach for massively parallel computers," *Computers in Engineering* **2**, pp. 75–82, 1992. 21

[48] S. Y. Synn and R. E. Fulton, "Practical strategy for concurrent substructure analysis," *Computers & Structures* **54**(5), pp. 939–944, 1995. 21

[49] S. H. Hsieh, S. Modak, and E. D. Sotelino, "Object-oriented parallel programming tools for structural engineering applications," *Computer Systems in Engineering* **6**(6), pp. 533–548, 1995. 22

[50] S. Modak, E. D. Sotelino, and S. H. Hsieh, "A parallel matrix class library in c++ for computational mechanics applications," *Microcomputer in Civil Engineering* **12**, pp. 83–99, 1997. 22

[51] Y. Escaig, G. Touzot, and M.Vayssade, "Parallelization of a multilevel domain decomposition method," *Computer Systems in Engineering* **5**(3), pp. 253–263, 1994. 22

[52] G. P. Nikishkov, A. Makinouchi, G. Yagawa, and S. Yoshimura, "Performance study of the domain decomposition method with direct equation solver for parallel finite element analysis," *Computational Mechanics* **19**, pp. 84–93, Nov. 1996. 22

[53] Ö. Kurç and K. M. Will, "Parallel linear solution of large structures on heterogenous pc clusters," in *5th International Conference on Engineering Computational Technology*, L. P. de Gran Canaria, ed., 2006. 23, 24

[54] "The message passing interface(mpi) standard." WWW page, May 30, 2009. `http://www.mcs.anl.gov/research/projects/mpi/`. 24, 65, 76, 100

[55] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing* **20**, pp. 657–673, 1994. 24

[56] M. Snir, *Mpi–the Complete Reference*, MIT Press, Cambridge, 1998. 24

[57] "Mpich." WWW page, May 30, 2009. `http://www.mcs.anl.gov/mpi/mpich1`. 24

[58] "Lam/mpi." WWW page, May 30, 2009. `http://www.lam-mpi.org/`. 24

[59] "Open mpi: Open source high performance computing." WWW page, May 30, 2009. `http://www.open-mpi.org/`. 24

[60] S. Markus, S. B. Kim, K. Pantazopoulos, A. L. Ocken, E. N. Houstis, P. Wu, S. Weerawarana, and D. Maharry, "Performance evaluation of mpi implementations and mpi-based parallel ellpack solvers," in *MPIDC '96: Proceedings of the Second MPI Developers Conference*, p. 162, IEEE Computer Society, (Washington, DC, USA), 1996. 24

[61] Özgur Kurç and S. Özmen, "An efficient parallel solution framework for the linear solution of large systems on pc clusters," *Tsinghua Science & Technology* **13**(Supplement 1), pp. 65 – 70, 2008. 24

[62] Ö. Kurç and K. M. Will, "A substructure based parallel solution framework for structural systems having multiple loading cases," in *the International Conference on Computing in Civil Engineering, ASCE*, Cancun, ed., 2005. 24

[63] "Mpich2." WWW page, May 30, 2009. `http://www.mcs.anl.gov/research/projects/mpich2/`. 24, 83

[64] W. Mcguire, *Matrix Structural Analysis*, John Wiley, New York, 2000. 28, 29

[65] I. S. Duff, "The impact of high-performance computing in the solution of linear systems: trends and problems," *J. Comput. Appl. Math.* **123**(1-2), pp. 515–530, 2000. 29

[66] F. Zhang, *The Schur Complement and Its Applications*, Springer Science, New York, 2005. 29

[67] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing* **26**, pp. 1519–1534, 1999. 34

[68] B. Hendrickson, "Load balancing fictions, falsehoods and falacies," *Applied Mathematical Modelling* **25**, pp. 99–108, 2000. 34

[69] Ö. Kurç, *Parallel Computing in Structural Engineering*, VDM Verlag, Germany, 2008. 35, 37

[70] T. A. Davis, "Algorithm 849: A concise sparse cholesky factorization package," *ACM Trans. Math. Softw.* **31**(4), pp. 587–591, 2005. 51

[71] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A proposal for a heterogeneous cluster scalapack (dense linear solvers)," *IEEE Transactions on Computers* **50**(10), pp. 1052–1070, 2001. 60

[72] Z. Chen, J. Dongarra, P. Luszczek, and K. R. A, "Self-adapting software for numerical linear algebra and lapack for clusters," *Parallel Computing* **29**, pp. 1723–1743, 2003. 60

[73] J. Ke, M. Burtscher, and E. Speight, "Runtime compression of mpi messanes to improve the performance and scalability of parallel applications," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 59, IEEE Computer Society, (Washington, DC, USA), 2004. 65

[74] "Overview of the windows api." WWW page, May 30, 2009. http://msdn.microsoft.com/en-us/library/aa383723.aspx. 76, 101

[75] J. J. Dongarra and T. Dunigan, "Message-passing performance of various computers," tech. rep., 1995. 100

[76] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, *How to build a Beowulf: a guide to the implementation and application of PC clusters*, MIT Press, Cambridge, MA, USA, 1999. 100, 101

[77] "MSDN "Microsoft Developer Network" Documentation." WWW page, May 30, 2009. http://msdn.microsoft.com/en-us/library/ms632592(VS.85).aspx. 101

[78] S. Succi, F. Papetti, and F. Papetti, *An Introduction to Parallel Computational Fluid Dynamics*, Nova Science Publishers, 1996. 102

APPENDIX A

PERFORMANCE ASSESSMENT OF PARALLEL ALGORITHMS

Performance analysis of parallel algorithms is based on the study of various tests. The aim of the performance analysis is to get tools for predicting and estimating how the parallel algorithm would behave as the number of processors increases. During these tests, elapsed, CPU and communication times can be measured; speedup and efficiency can be computed; and features of parallel processing, like the reduction in processing time for a given problem or the size of a problem to solve on a given time, can be addressed.

A.1 General features of parallel processing

PC Cluster is composed of a group of computers where each computer has at least one processor and one local memory. They are connected through a network. Thus, the natural way of programming is message passing. In this kind of computing environments, programming efficient codes rely on factors such as

- the number of processors and the capacity of their local memories;
- the communication speed among processors;
- the ratio of the computation and communication speeds.

Besides, performance of the distributed memory architectures depends greatly on the network features:

- Topology: how the nodes are connected;

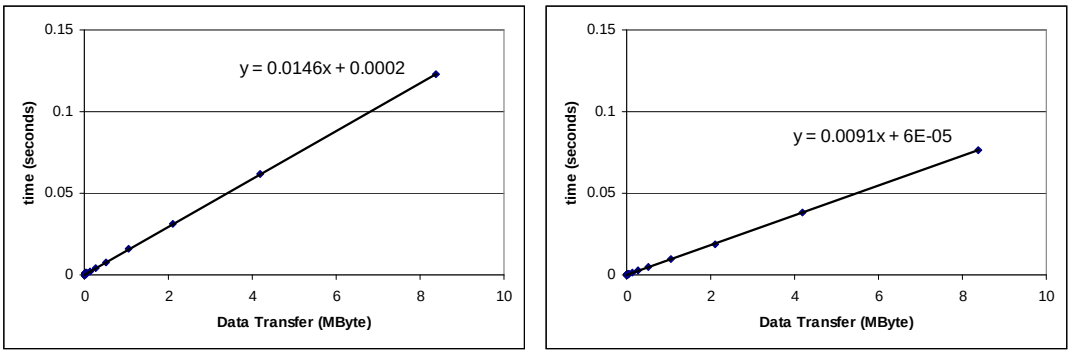
- Latency: time required to initiate the communication;
- Bandwidth: maximum speed for data transfer.

A.2 Time measures

Elapsed times have been measured by means of the MPI [54] function MPI_Wtime(). In general, the time for a message transfer between two processors may be given by [75]

$$t_{comm} = \alpha + \beta n \tag{A.1}$$

α is the latency or start-up time; β is the time needed to transmit 1 byte, and n is the message length (in bytes). $\theta = 1/\beta$ is the bandwidth. Both of the clusters exists in laboratory are tested by transferring different size of data among computers to obtain communication time measures (Figure A.1).



(a) Results from HPCE Cluster (b) Results from HPCE2 Cluster

Figure A.1: Data Transfer vs Time Graphs

By approximating the results by a trendline, communication time measures for;

- HPCE Cluster, $\alpha = 200\,\mu s$ and $\theta = 66.22\,MB/s\ = 556\,Mbit/s$
- HPCE2 Cluster $\alpha = 60\,\mu s$ and $\theta = 96.21\,MB/s\ = 807\,Mbit/s$

are obtained.

The bandwidth measures for clusters are lower than the theoretical values because rest of the bandwidth is used by the communication protocols for headers, routing information etc. [76] The communication time may be estimated for HPCE and HPCE2 clusters as respectively;

$$t_{comm} = 2.00 \times 10^{-4} + 0.0146n \tag{A.2}$$

$$t_{comm} = 6.00 \times 10^{-5} + 0.0091n \tag{A.3}$$

Another measure of interest is

$$n_{1/2} = \frac{\alpha}{\beta} = \alpha\theta \tag{A.4}$$

that is the length of a message for which both terms of the t_{comm} equation are equal. Results for this measure are $n_{1/2} = 13.56\,KBytes$ for HPCE Cluster and $n_{1/2} = 5.91\,KBytes$ for HPCE2 Cluster. Messages of length much shorter than $n_{1/2}$ would be dominated by latency, whereas messages of length much longer than $n_{1/2}$ would be dominated by bandwidth [76].

A.3 Computational speed

The performance of a processor is measured in megaflops: millions of floating point operations (flops) per second. Frequency performance tools of WinAPI [74] is used to estimate the speed of the processors in the cluster. When the size of the problem grows, the speed of the processors decreases, because the matrix can no longer be fully contained in the cache, and parts must be reloaded from main memory.

Frequency performance tools of WinAPI [74] is composed of *QueryPerformance-Frequency* and *QueryPerformanceCounter* functions. *QueryPerformanceFrequency* retrieves the frequency (f) of the high-resolution performance counter in CPU and this frequency can not change while the system is running. On the other hand, *QueryPerformanceCounter* retrieves the current value of the high-resolution performance counter. Therefore the elapse high-resolution performance count (L) can be calculated by subtracting the values of two different calls to this function [77].

Therefore, the elapsed time between two successive calls to high-resolution performance counter function in seconds can be calculated as follows;

$$t_{comp} = \frac{L}{f} \tag{A.5}$$

Various computational speeds for different stages of the solution framework is discussed in related sections.

A.4 Parallel measurement

Performance of parallel programs can be computed by the calculation of the following measures. Let t_1 be the time to execute a given problem with one processor, and t_p the

time needed to execute the same problem with p processors. Then the *speed-up* (S_p) is the relationship among the elapsed times using 1 and p processors:

$$S_p = \frac{t_1}{t_p} \tag{A.6}$$

This measure is a function of the number of processors, although it also turns out to be a function of the problem size. If p processors are utilized, it is expected that the parallel time would be nearly $1/p$ of that corresponding to only one processor. This yields an upper bound equal p for S_p:

The *efficiency* is defined as the speed-up but relative to the number of processors,

$$E_p = \frac{S_p}{p} = \frac{t_1}{p.t_p} \tag{A.7}$$

In an ideal situation, an efficiency equal 1 would be expected. Another way to express the efficiency is the following:

$$E_p = \frac{1}{1+\omega} \tag{A.8}$$

where ω represents the 'generalized' overhead; that is, the communication to computation ratio. The most important sources of parallel overheads are [78]

- *Communications and coordinations.* The parallel execution time t_p with p processors, may be represented in the following manner:

  $$t_p = t_{coor} + t_{comm} + t_{comp} \tag{A.9}$$

  where $t_{comp} = t_1/p$; t_{coor} is the coordination overhead and t_{comm} the communication overhead. Thus, the speed-up and the efficiency may be expressed as follows:

  $$S_p = \frac{1}{\frac{1}{p}+\frac{t_{coor}+t_{comm}}{t_1}} \text{ and } E_p = \frac{1}{1+\frac{t_{coor}+t_{comm}}{t_{comp}}}$$

  therefore, from A.8 it results as

  $$\omega = \frac{t_{coor} + t_{comm}}{t_{comp}} \tag{A.10}$$

- Redundancy. This type of overhead takes place when a parallel algorithm performs the same computations on many processors.

- Load unbalance. This overhead measures the extra time spent by the slowest processor to do the assigned tasks relative to the time needed by the other processors. So, the elapsed time is dictated by the slowest processor.

- Extra work. They are parallel computations that does not take place in a sequential implementation.

In particular, that the communication versus computation performance ratio of a distributed-memory computer significantly affects parallel efficiency. The ratio of the latency to the time per flop ($\frac{t_m}{t_f}$) greatly affects the parallel efficiency of small problems. The ratio of the network throughput to the flop rate ($\frac{t_f}{t_v}$) significantly affects the parallel efficiency of medium-sized problems. For large problems, the node flop rate ($\frac{1}{t_f}$) is the dominant factor contributing to the parallel efficiency of the parallel algorithms for dense systems [10].

APPENDIX B

STRUCTURAL MODELS USED FOR PERFORMANCE TESTS

Three different structural models involving generated and actual models were tested in this study. First one, which is named as Cube30, is the mathematical model of an imaginary cube. Second one, which is named as GroM, is an actual structural model generated for modelling a nuclear waste plant. These two models are uniform and symmetric models. The third model, on the other hand, is another actual model of a 30 storey high rise building. This model is a very representative model for such reinforced concrete buildings, thus the performance of the parallel solution approaches will be a good indicator for the solution of actual civil engineering models where no symmetry and uniformity exists. Structural and computational characteristics of these models will be discussed in detail.

B.1 Cube30

Cube30 is the mathematical model of an imaginary cube (Figure B.1) that has 27,000 (30x30x30) 8-node brick elements with 29,791 nodes and 89,364 equations. This model represents a symmetric and uniform model composed of same type of elements with significantly large bandwidth.

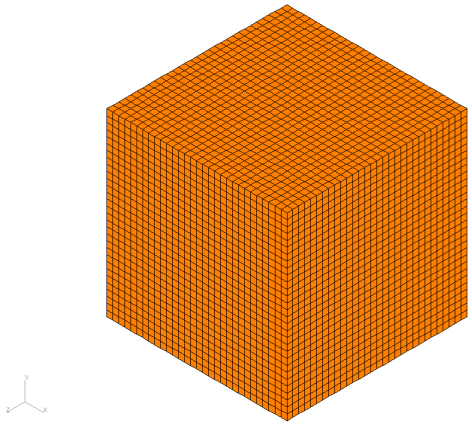


Figure B.1: Cube30 Model

B.2 GroM

GroM is an actual structural model generated for modelling a nuclear waste plant (Figure B.2) that has 2,811 frames and 43,776 quadrilateral shell elements with 46,587 nodes and 233,616 equations. This model is composed of both 1D and 2D elements but the connectivity among 1D and 2D elements is limited. The structure is geometrically symmetric in two horizontal directions. The bottom box consists of many small size rooms whose walls are also modelled with finite elements. This situation significantly increases the bandwidth of the structural stiffness matrix of this model.

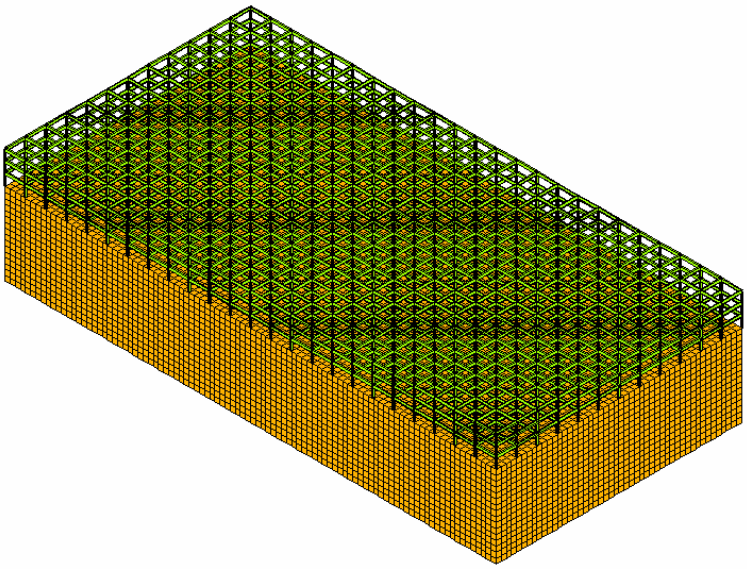


Figure B.2: GroM Model

B.3 StRegis

StRegis is another actual model of a 30 storey high rise building (Figure B.3) that was prepared by a structural engineer for designing a mixed use hotel and residential building composed of flat plate slab systems with shear walls and columns. The model was composed of 6,641 frames and 86,509 both quadrilateral and triangular shell elements with 87,075 nodes and 518,580 equations. The model is highly irregular, the plan of lower levels are different at each level with large openings at various locations. The slabs are modelled with both triangular and quadrilateral elements with no uniformity at levels except than 15 middle residential levels. The slabs are connected with 1D elements that represent columns. This model is a very representative model for such reinforced concrete buildings, thus the performance of the parallel solution approaches will be a good indicator for the solution of actual civil engineering models where no symmetry and uniformity exists.

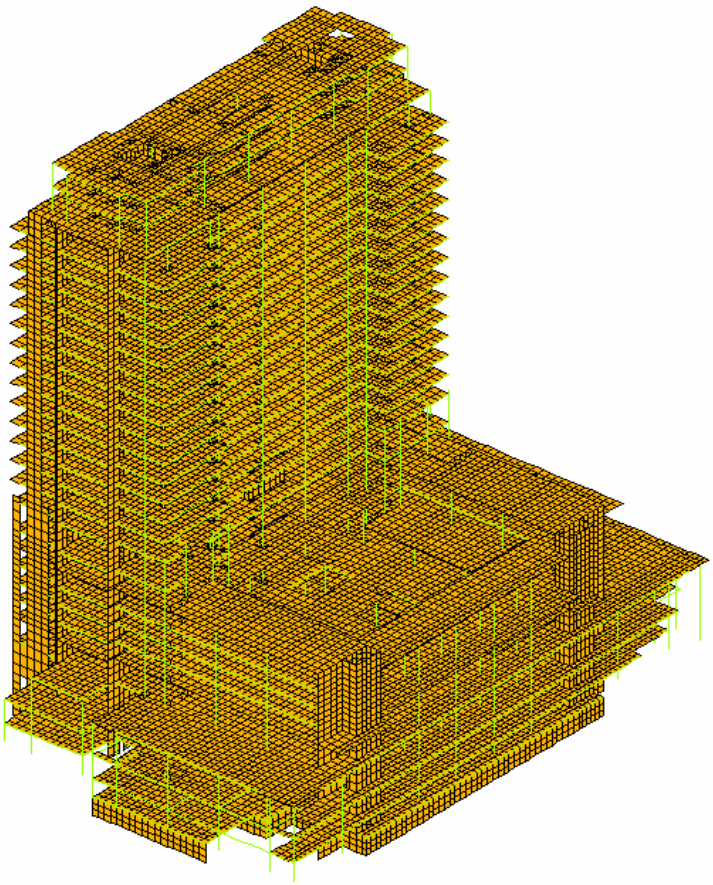


Figure B.3: StRegis Model