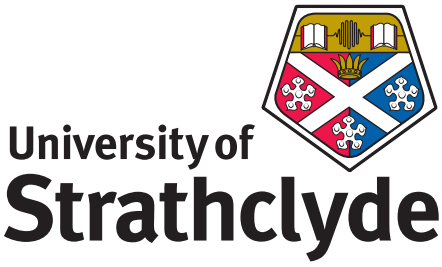



# Parallel finite element methods and software for partial differential equations

Omer Riaz

Department of Mathematics

University of Strathclyde

Glasgow, UK

September 2014

This thesis is submitted to the University of Strathclyde for the degree of Doctor of Philosophy in the Faculty of Science.

# Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors Prof. Mark Ainsworth and Dr. Gabriel R. Barrenechea. For the continuous support of my Ph.D study and research, for their immense knowledge, patience, motivation, and enthusiasm. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisers and mentors for my Ph.D study.

I would like to thanks Numerical Algorithms and Intelligent Software (NAIS) as this Ph.D was not possible without their financial support.

Lastly, I would like to thank my family: my parents and wife for their patience and giving me moral and spiritual support.

# Abstract

In this thesis a Finite Element solver package called FEDomain is developed for C++ finite element software developers. It is focused on solving the Finite Element problem on shared memory as well as distributed memory architectures. The FEDomain package segregates the finite element software into two phases. The first phase includes defining the finite element problem. The user selects the mathematical problem, domain shape, domain dimensions, triangulation of the domain and formulations to compute elements' data. The second phase comprises assembly of system of equations and computing its solution. The FEDomain package concentrates on the second phase. It facilitates the user by providing the efficient implementation of the second stage using parallel algorithms. This design allows the C++ finite element application developers, with no knowledge and experience of parallel computing, to implement parallel finite element application for shared and distributed memory architectures.

More specifically, FEDomain package is focused on introducing a new type of user interface. The interface requires the user to provide the mathematical problem and domain related data in terms of C++ element objects. The FEDomain assembles the system of equations, computes its solution, and provides it back to the user through element objects. The FEDomain package computes the residual vector and solution for the system of equations on shared memory and distributed memory architectures.

# Contents

# Chapter 1

# Introduction

The Finite Element Method (FEM) [9, 50] is widely used for simulating physical problems. The method can be used in design stage as well as to improve a previously existing design. The main advantage of the FEM is its flexibility which allows it to handle arbitrary geometries, different materials and general boundary conditions. One of the key components of the FEM is the solution of the linear system of equations. This process generally involves matrix operations (such as multiplication) which are computationally intensive. Finite element software developers usually use third party linear algebra solver libraries for these routines. The efficient implementations of linear algebra libraries require thorough understanding of the algorithms and coding details. The implementation of these routines is a time consuming task which is prone to error. This thesis presents a novel C++ finite element solver package which alleviates some of these problems by providing efficient solution computation methods for parallel architectures.

This chapter has three section. The first section contains motivation and objectives of this work. The second section covers a literature review of the parallel architectures, parallel libraries, and linear algebra numerical libraries. The third section gives a brief review of the forth coming chapters in the thesis.

# Motivation

In this chapter many commercial and open source packages will be reviewed. Some of these packages are lower level libraries, which have been matured by years of deployment, development and error correction. These have been acting as building blocks for the higher level libraries. The higher level linear algebra libraries have been developed to support numerical applications development. Most of these applications are developed in functional languages like C and FORTRAN. These packages require their users to provide the system matrix and the right hand side. The object oriented finite element developers have to accumulate all the mesh elements data into a global data containers and provide these to the solver. Like wise the solution is provided as a vector to the user. The user has to distribute the solution to all the mesh elements for post processing.

The third category will be reviewed in this chapter is object oriented finite element packages. These are developed as a complete solution which starts from generation of the mesh, creation of the linear systems, solving the linear system and providing an output either as values or as visualisation.

## 1.1 Aims

The aim of this project is to define a generic interface of finite element solver package. This package will lie between the high level linear algebra libraries and object oriented finite element packages. The interface should be simple and easy to adopt by the users, unlike high level linear algebra libraries. It should allow the user to implement any valid finite element method for any model problem. The domain specifications and problem type will be selected by the user. The user will have complete control over the mesh generation and properties of the mesh elements and should select the algorithms to be implemented for the elements.

The concrete way of achieving our main aim, is to develop a finite element solver which assembles the linear system of equations from the element objects as it is implemented in most of the object oriented finite element packages. The system matrix and the right hand side have to be provided to the linear algebra solver. The solver should be responsible for the selection of $\boldsymbol{A}$ and $\vec{f}$ data containers. The FEDomain will select its internal data structures according to the solver selection. The user has to provide the mesh element objects to FEDomain. The FEDomain will collect the system data from these element objects.

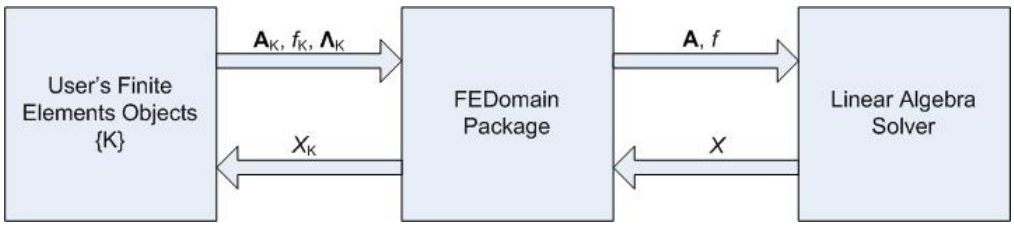



Figure 1.1: Overview of FEDomain package.

The FEDomain package will be an object oriented package which will compute the solution in parallel, both on shared and distributed memory architectures. The package will provide a solution to all the allocated mesh elements as can be seen in Figure 1.1. The FEDomain package will depend on a third party linear algebra package to compute the solution.

## 1.2 Objectives

This project has implemented a finite element solver package. The objectives of this project are as follows.

1. *Simple interface:* The package should provide a simple interface between the user application and the linear algebra solvers. The user should be able to include this package into their C++ finite element application code with minimal efforts.

2. *Generic interface:* The interface should be defined to accommodate as wide a family as possible of problems and finite element spaces. The package should provide support to all finite element shapes and domain dimensions.

3. *Operating system independent:* The package should be compatible to commonly used operating systems like Windows, Linux and MacOS. It should not be dependent on operating system specific resources.

4. *Support multiple solution methods:* The interface should give the user the choice of either direct or iterative methods to solve the linear systems, including domain decomposition method.

5. *Support for shared and distributed architectures:* The package should provide support for shared and distributed memory architectures.

6. *Solution distribution:* The package should provide solution back to the finite elements.

7. *Extendible:* Give to the user the possibility to implement algebraic solvers different to the already implemented ones. It will help the user to implement iterative solvers by calculating residual vector on shared as well as distributed memory architectures.

These objects could be checked using following indicators:

1. The interface of the primary FEDomain library should be concise and easy to read. This makes it easy for the user to include it into their applications.

2. The only requirement from the user is a group of elements and a list of Dirichlet degrees of freedom. These elements should provide

   - a local stiffness matrix,
   - a local right hand side, and
   - a connectivity matrix.

The structure allows any choice of degrees of freedom (point, fluxes, means, moments, etc).

3. A variety of problem types have been implemented and their performance tested. These include scalar equations (such as Laplace and convection-diffusion) and systems (such as elasticity). Also, the interface can deal with symmetric and non symmetric problems, and can accommodate a non-linear solver for the convection-diffusion equations.

4. The user has the choice of algebraic solvers to use. The Jacobi algorithm is implemented to test an iterative solver on shared and distributed memory architectures.

5. The package should provide support to multiple third party linear algebraic solvers.

## 1.3 Parallelism, Core and Threads

Parallelism can be defined as executing many computing calculations simultaneously. The basic idea is to divide the large groups of calculations into smaller groups and solve these at the same time. There are two basic concepts: core and thread. Both have to be explained as these will be used in this thesis.

A core is a physical processing unit in a processor chip. In earlier computers there used to be one core per processor. Physical constraints like power delivering, heat dissipation, and need for higher frequency have made single core processors outdated. The earlier attempt to overcome these constraints involves the introduction of the motherboards with two physical processor sockets which communicate with each other through added circuitry on these motherboards. The communication was considered as much slower to the calculations actually happening inside the processors. To increase processing speed, lower manufacturing cost, and end

user cost, processors are designed to have multiple processing cores in a chip. These cores perform faster as these can communicate swiftly and share common circuitry like caches.

A thread is a set of instructions from a computer process that is executed by a core. A single process is divided into multiple threads which can be simultaneously execute on multiple cores.

## 1.4 Parallel Architectures

Parallel computers have two basic architectures: shared memory and distributed memory.

- In a shared memory computer, multiple processor units share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data. Typically, the number of processors used in shared memory architectures is limited to only a handful of processors. This is because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.

- Distributed memory parallel computers are essentially a collection of computers (nodes) working together to solve a problem. Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network like Ethernet and InfiniBand. Data are exchanged between nodes as messages over the network.

## 1.5 Parallel Libraries

In this section the parallel libraries are reviewed. These libraries are implemented for the shared memory and distributed memory architectures.

### OpenMP

OpenMP [72] is a language extension consisting of pragmas, routines and environment variables implemented for C and Fortran programs. OpenMP is a form of a directed parallelism in which application developers have to add OpenMP pragmas in their application sequential code. These OpenMP pragmas indicate parallel regions to the compiler to generate an equivalent parallel code. The application developer is responsible for correction of the code to avoid parallel software drawbacks like race conditions and deadlocks, etc.

*OpenMP V3.0* [72] provides three scheduling schemes *STATIC*, *DYNAMIC*, and *GUIDED* which represent different algorithms for task scheduling and load balancing. The description of OpenMP scheduling schemes is given below:

- In *STATIC* scheduling, the data is divided into subsets or bins of size *chunk size* only the last subset can be of smaller size. The number of subsets can be more than the total number of threads, and subsets are allocated to threads in a round robin manner.

- In *DYNAMIC* scheduling, the elements are distributed into subsets of size *chunk size*. When a thread finishes one chunk, it is dynamically assigned another until all the subsets are assigned. None of the threads remain in idle state or wait for other thread to finish.

- In *GUIDED* scheduling, the subset of chunk size is proportional to the unassigned data and total threads. Only the last chunk size is less than user

selected *chunk size*. The chunk allocation algorithm is the same as of *DYNAMIC* scheduler. No thread waits for other threads and chunks are allocated as the threads are available.

## Thread Building Blocks

Thread Building Blocks (TBB) [76] is an *Intel*® Corporation library developed to leverage multi-core programming by supporting scalable parallel programming in standard C++. TBB is implemented as template C++ library to take full advantage of Generic programming. It provides performance and scalability while presenting user with higher level task base parallelism. TBB perform synchronization, load balancing and cache optimization to attain increase in speed.

Contrary to OpenMP, TBB library does not need special compiler and can be built for new and older compilers. TBB also provides thread safe data structures like concurrent_vector and concurrent_queue which is TBB advantage over OpenMP. Generic programming enables TBB to be flexible yet efficient while optimizing components to the user's own requirements. TBB can provide better portability, easier programming and more understanding of C++ code due its generic implementation.

## Task Parallel Library

Parallel Library (TPL) [30] provides concurrency support for the Microsoft .NET framework. It provides parallel constructs such as aggregate, do and for to keep different queues balanced. It is based on the concept of a task-based asynchronous operations. The TPL utilizes the threads available to execute these tasks in parallel. Unlike OpenMP and TBB, TPL is supported by MS platform and .NET only.

## Message Passing Interface (MPI)

The MPI library is implemented to make the application executable on the distributed memory computer. It provide the routines to manage the tasks on the group of computational nodes. MPI is intended as a standard implementation of the "message passing" model of parallel computing. A parallel computation consists of a number of processes, each working on some local data. Each process has purely local variables, and there is no mechanism for any process to directly access the memory of another. Sharing of data between processes takes place by message passing, that is,by explicitly sending and receiving data between processes. MPI provides great deal of functionality, including a number of different types of communication, special routines for common "collective" operations, and the ability to handle user-defined data types and topologies. It provides support for heterogeneous parallel architectures.

MPI library can efficiently transfer data organised in consecutive memory in single transfer message it is designed as a functional library. There are few third party object oriented interfaces and implementations of MPI like OOMPI [6]. These interfaces are not accepted as standard in the high performance computing community. The MPI library cannot transfer class objects as mostly an object is set of functionality and data.

## .NET Framework Remoting

.NET Remoting is Microsoft's infrastructure that provides a rich set of classes that allow developers to ignore most of the complexities of deploying and managing remote objects. The calling methods for remote objects are nearly identical to calling local methods. Remoting framework is built into the common language runtime (CLR) that is used to build sophisticated distributed applications and network services. When a client creates an instance of a remote object, it re-

ceives a proxy to the class instance on the server. All methods called on the proxy will automatically be forwarded to the remote class and any results will be returned to the client. From the client's perspective, this process is no different than making a local call. The .NET framework is Windows OS dependent. Other packages like MONO [1] and Portble.NET [2] provide common language infrastructure. MONO supports windows, GNU/Linux and MacOS X. Portable.NET supports GNU/Linux, Solaris, NetBSD and MacOS X. These frameworks are not available on all the development environments.

## 1.6 Basic concepts on object oriented implementation of Finite Element Methods

Object oriented languages provide facilities for balanced support of data abstraction and encapsulation, polymorphism, extensibility and code re usability through inheritance and run-time type support via dynamic binding of operations to objects. Object oriented languages make it possible to construct software using software components directly modelling real world high level entities. There are many object oriented languages like C++, Objective C, C#, Visual Basic .Net, and Java, etc, available for software development. C++ is improved C as it provides methods for data hiding and data encapsulation. C++ retains the efficiency of C by allowing the low level memory access and floating point arithmetic to be carried out without overheads. It also provides object oriented features like inheritance, polymorphism, and real time type support via dynamic binding of operations to objects.

Upto our knowledge, the first finite element object oriented framework is given in [61]. FE++ is an object oriented architecture developed in C++ for finite element programming. This package implements finite element object classes for structural mechanics. This package partitions the common attributes and the concepts in

the finite element analysis into an object oriented architecture using components like element, node, material, assembler, and solver. Later TF++ and PTF++ [70] frameworks are developed using FE++ architectures to compute transient solutions.

A more concrete realisation in terms of finite element solver is given in [32] (1990), where Ford et al presented an object oriented finite element program for linear elasticity analysis with plane, isoparametric elements. The aim was to develop a framework that could be easily expanded to more advanced problems, or incorporated in expert systems. There are 5 FEM classes that have only a few attributes and methods. In addition to the Element, Material, and Node classes, the boundary conditions are handled by a DispBC class and a ForceBC class. To each FEM class belongs a customized version of a List class which handles storage and assembly of the model. The Element is capable of computing the element stiffness and several types of distributed loads. This is done by numerical integrations and requires two new classes, name Gausspoint and Shapefcn. Furthermore, the Element has its own post processing facilities such as stress evaluation and graphical representation of the result. The entire finite element model is represented by a Domain class which stores the customized lists of Nodes, Elements, Materials, and boundary conditions. It is also responsible for the storage of the global matrices and vectors. To perform an analysis the user should provide an application program that controls program evaluation, i.e. definition of a Domain, solution and call of post processing facilities. The expandable framework is simple due to few FEM classes. The program was implemented in a hybrid language using C for the numerical part and ObjectPascal for the object oriented part.

One of the first object oriented finite element application is explained in [78] (1992), where Scholz has developed and implemented a typical finite element analysis program on the basis of the Timoshenko beam using the object oriented pro-

gramming language C++. The merits of object-oriented programming are shown by providing two examples of the design, implementation, and application of the object classes "Vector" and "Matrix". Scholz has again indicated that an object oriented approach to engineering problems leads to easier validation and maintenance of programs than procedural languages, and that the implementation of an object oriented program requires less time and produces smaller programs compared with conventional programming techniques.

The concept of the central data for finite element is used in [82] (1993), where Yu and Adeli define a class library for finite element analysis. The analysis is centred around a GlobalElement object which handles the model assembly. It is a subclass of Element and uses several objects like Node, Material and Shape. The model is stored in a central database from which it is possible for any object to get the data that are needed. A noticeable difference to the systems presented above is the possibility for each object to copy itself, e.g. generate a number of equally spaced Nodes. The class library has been tested on composite laminate problems using a C++ implementation. The authors in [8] (1995) have presented a blackboard software architecture. The blackboard consists of local and global controllers that control the logic of the problem solving. An object oriented database management system (OODBMS) has been created for effective management of input, intermediate and output data. The methods and models developed in this research have been applied to the solution of the inter laminar stress analysis of composite laminates.

One of the earliest examples of finite element programs implemented using Microsoft C++ 7.0 is "Event" given in [56] (1995) by Kong and Chen. The polymorphism and inheritance are used to implement the finite element program. The Event reads the problem data from the file, create elements from the provided geometry and physical classes. The element classes are implemented using multiple

inheritance. From these elements, the stiffness matrix and load vector are created to compute a solution. The focus was to create different element objects by using multiple inheritance. Kong in [55] (1996), studied how a FORTRAN finite element code can be translated into C++ finite element application. The paper shows the way of applying the object oriented design, enhancing data hiding and cohesion, decreasing data coupling, as well as managing finite element objects. It proposes a process of defining private data members and arranging them into C++ FEM class hierarchies to remove the drawbacks of FORTRAN common variables. Static data members are used for implementing common resources that all objects need. The C++ linked list of finite element objects are used to gain high effectiveness and flexibility.

In [62] (1992), Mackie used ObjectPascal to represent the possibility of changing from procedural programming to object oriented programming. A class of Elements for plane stress and plate bending have been defined for static as well as dynamic analysis. The class methods are defined parametric in a style that lies close to the traditional Fortran style. What is accomplished by using object oriented programming is, however, an enhancement of the program structure as well as re-usability due to inheritance. This work was later extended to a C++ implementation in [64]. In this reference a software approach to fully interactive finite element software is presented. The objective of this work was to generate finite element classes with lean interface and the finite element objects are distributed around the graphical model objects. This work distributes the finite element application into two modules, finite element class system and graphical structural model. The graphical model implements the classes key points (TKeyPt), key lines (TKeyLine) and key substructures (TSubStruct). These classes are implemented from the user point of view and data is encapsulated within them. After generating the mesh nodes belong to these structures. The author has used substructuring to compute the solution of these nodes. It uses the TFeNode class for

the nodes, TElement class to represent elements and TProperty class for material properties. The extension to multi-threaded parallel implementations of the finite element software given in [64] was presented in [63, 65, 68].

The main concept described in the previous paragraph was extended to distributed computing in [66] by breaking the calculation into large segments. The motivation is to create an implementation which appears relatively seamless for concurrent and distributed computing. The .NET framework is used to achieve the desired goal. In [66], domain decomposition methods are implemented using direct and iterative linear equation solvers using C# language. The implementation was made using C#, and it showed that using interfaces allows greater flexibility in software design than using inheritance, especially in the case of multiple inheritances. IVectorD and ISubDomain interfaces are used for the data containers classes and sub domains classes, respectively. In [67], Mackie has implemented three different solution scenarios to compute the solution using Conjugate Gradient iterations and showed how the interfaces can be used to add these solutions scenarios in Conjugate Gradient class (CGSolverGen). The work shows that the use of an interface in object oriented application reduces code complexity and increases extensibility. The solution classes implemented ICGMatrix interface and the CGSolverGen class has ICGMatrix object. The solution classes implement concurrent and distributed algorithms. The interfaces allow to hide the local and remote objects from the CGSolverGen object.

The possibility of using multiple computers connected through the internet was advised in [69] by the introduction of remote objects and mobile agents. The focus was on developing flexible software by using interfaces and mobile agents. It showed using delegates for remote objects are easy to use and efficient method for distributed computing. The author has demonstrated design flexibility by the implementation for direct solver ($\boldsymbol{U^T D U}$) and iterative (conjugate gradient)

distributed solvers using mobile agents. It is also shown how to perform distributed assemblies, user interface approach for solver methods and user hosting approach for mobile agents.

A design pattern [35, 10, 34] is the abstraction of a recurring solution to a design problem. It captures the relationship between objects participating in the solution and describes their collaboration. In [35] the detailed introduction and explanation of 23 software design patterns (given in table 1.1). The patterns solve specific design problems and make object oriented code designs more flexible, elegant and, ultimately reusable. These design patterns help software developers to reuse successful designs in their implementations. A software designer familiar with these design patterns can apply them immediately to design problems which will help in fast and right implementation and clean code. These also help in communication among designers and improve documentation. Each design pattern specifically targets recurring design in object oriented systems.

| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioural |
| Scope | Class | Factory Method | Adapter | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facede<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

Table 1.1: Design pattern space given in [35].

Heng and Mackie [44] have presented five basic design scenarios that appear during object oriented finite element software development. The paper explains what design patterns for these scenarios and how these can be applied. Table 1.2

shows these finite element design scenarios and their appropriate design patterns. The detailed explanation, and implementation of scenarios is given in [45].

| Scenarios | Design Patterns |
|---|---|
| Model-Analysis separations | Facade, Mediator |
| Model-UI separations | Model-View-Controller, Observer |
| Modular Elements | Singleton, Template, Strategy |
| Composite Elements | Composite |
| Modular Analysis | Strategy |

Table 1.2: Finite element software scenarios and their appropriate design patterns given in [44].

There are many commercial and open source libraries available for linear algebraic software development. Most of these libraries a continuously evolving to meet new requirements of software developers. Following is an introduction of few commonly used libraries.

## 1.7 Low Level Linear Algebra Systems

*Basic Linear Algebra Subprograms*(BLAS) [57] is a set of low-level kernel subroutines that perform common linear algebra operations such as copying, vector scaling, vector dot products, linear combinations, and matrix multiplication. BLAS is a standard application programming interface API for linear algebra routines, which has to be tuned for specific computer architectures. BLAS functionality is divided into 3 levels. The first level contains vector operations, the second level contains matrix vector operations and the third level contains matrix matrix operations. It is used as the building block for the high level programming language.

*Linear Algebra PACKage*(LAPACK) [12] is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, Schur, gen-

eralized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision. LAPACK routines are written so that as much as possible of the computation is performed by calls to the BLAS. LAPACK is designed at the outset to exploit the Level 3 BLAS. The coarse granularity of the Level 3 BLAS operations, promotes high efficiency on many high-performance computers, particularly if specially coded implementations are provided by the manufacturer.

### 1.7.1 Tuning Paradigms

The lower level linear algebra kernels act as the building blocks for higher level packages. Their performance is critical for the performance of these high level packages. In this section few tuning packages have been reviewed. These packages tune the lower level linear algebra kernels according to the target architecture.

*Automatically Tuned Linear Algebra Software* (ATLAS) [81] is the application of Automated Empirical Optimization of Software (AEOS) paradigm to dense linear algebra software. It produces a BLAS library which is optimized for the target platform. The BLAS building block routines, when tuned well, allow more complicated linear algebra operations such as solving linear equations to run extremely efficiently. ATLAS generates linear algebra kernel for system at installation by static tuning of BLAS library and generated kernel routines are called by the user. ATLAS allows the user to avoid hand tuning his code as it requires detailed knowledge of a complex set of interrelated factors and is a time consuming task.

*Portable High Performance ANSI C* (PhiPAC) [20] is a methodology to achieve high performance linear algebra C libraries for wide range of hardware. PhiPAC strategy consists of three components for code development. In the first phase,

these provide generic model of current C compilers that provides guidelines for producing high performance ANSI C code. In the second phase, these provide parametrized generators which produce optimized routine according to the guidelines. In the third phase, script is provided to automatically tune the code for a particular system by varying the generator parameters and benchmarking the resulting routines. The code guidelines include issues like remove false dependencies, reduce memory bandwidth or use of base + constant offset addressing mode to avoid unnecessary pointer updates.

*The Optimized Sparse Kernel Interface* (OSKI)[80] from the Berkeley Benchmarking and Optimization Group (BeBOP) is a released software package providing automatically tuned sparse computational kernels. OSKI provides statically tuned kernels according to the underlying machine architecture created upon installation. It also provides dynamically tuned routines created at runtime according to the matrix/vector structure. The static kernel becomes the default and called when runtime tuning is not used. OSKI allows the user to select different runtime tuning options such as justify, moderate or aggressive. It can also save tuning transformations for reuse so that tuning overhead can be reduced for future use. OSKI also provides a parameter list input (a way to specify a variable number of inputs) that can be used to indicate the properties of a given problem that OSKI can exploit during the tuning process. The more information a user can provide, the less work it is for OSKI to automatically tune because it narrows down the number of tuning techniques.

## 1.8 High Level Linear Algebra Libraries

This section reviews linear algebra packages which support high level users. These packages provide support to languages such as C, C++, FORTRAN and Python, etc. These packages support multiple solvers on variable OS. Most of these

provide parallel routines for shared memory and distributed memory architectures.

*Unsymmetric Multifrontal Sparse LU Factorization Package*(UMFPACK) [27] is a set of routines for solving unsymmetric sparse linear systems, using the Unsymmetric MultiFrontal method. It includes a MATLAB interface, a C-callable interface, and a Fortran-callable interface. It can be used as a stand alone library but for high performance it uses BLAS routines.

*Portable, Extensive Toolkit for Scientific Computation* (PETSc) [15] is a powerful set of tools for the numerical solution of partial differential equations and related problems on high performance computing. It contains a variety of libraries developed to ease the development of large scale scientific application codes in C, C++, Fortran and Python. Each library manipulates particular sets of objects and the operation one would like to perform on the objects. PETSc provides parallel dense vectors and matrices (dense and several sparse matrices storages). It supports symmetric, block diagonal and sequential matrices. It provides many linear algebra operations like preconditioners (like ILU, LU, Jacobi , block Jacobi, additive Schwartz and ICC), Direct Solvers (like LU and Cholesky), Krylov Subspace methods (like GMRES, Richardson and conjugate gradient), non linear solvers (like Newton-based method, line search and trust region) and parallel time stepping solvers (like Euler). It provides interface to other packages like MUMPS [11] and SuperLU [58]. Packages like OpenFEM, OOFEM and DEALII [17] are using PETSc. PETSc does not provide any sparse vector and matrix matrix multiplication. The user is responsible for creation and population data and will select the solution method.

*MUltifrontal Massively Parallel Solver* (MUMPS) [11] is an other popular package for solving systems of linear system of equations of the form $\boldsymbol{A}x = b$, where $\boldsymbol{A}$ is a square sparse matrix that only need to be invertible. MUMPS implements a direct method based on a multifrontal approach which performs a direct factoriza-

tion. This package includes features like solution of the transposed system, input of the matrix in assembled format or elemental format, iterative refinement and computation of Schurs Complement matrix. This requires other libraries like MPI [73], BLAS [57] and ScaLAPACK. It supports sequential and parallel (shared and distributed memory) applications.

*SuperLU* [58] is the collection of three related subroutine libraries for solving sparse linear systems of equations. It supports multiple right hand sides and arbitrary square matrices. SuperLU has different implementations for sequential processors, SuperLU_MT is implemented for shared memory multiprocessors (SMP) using PThreads, and Super_DIST is implemented for distributed memory multiprocessors and uses MPI library. It provides FORTRAN interface for all three implementations. All the three libraries use BLAS to achieve high performance. In SuperLU_DIST, $\boldsymbol{A}$ can be either stored as replicated if all nodes have enough memory or distributed across all processors where each process is allocated with consecutive rows. The distribution storage is slow as in all stages of solver data has to be re-distribute.

*Parallel Sparse Direct and Multi-Recursive Iterative Linear Solvers* (PARDISO) [7], [77] is a solver package for solving large sparse symmetric and unsymmetic linear systems of equations on shared memory and distributed memory multiprocessors. The commercial version of this package is provided as a part of *Intel*® Math Kernel Library (MKL). It supports general invertible matrices. PARDISO can solve for multiple right hand sides. This package is available in C and FORTRAN languages.

*Watson Sparse Matrix Package* (WSMP) [41, 42] is a collection of algorithms for efficiently solving large systems of linear equations whose coefficient matrices are sparse. This library can be used as a serial package, or in a shared-memory multiprocessor environment, or as a scalable parallel solver in a message-passing

environment, where each node can either be a uniprocessor or a shared-memory multiprocessor. It implements both direct and iterative methods of linear systems. This package is available in C and Fortran languages.

*Matrix Template Library 4* (MTL4) has implemented a linear algebra library in C++ using modern object oriented programming techniques to provide an easy and intuitive user interface, while enabling optimal performance. It provides natural mathematical notations which enables engineers and scientists to implement algorithms in a minimum time. It has used template meta programming like expression templates and meta tuning [39]. It uses BLAS for some operations like dense matrix multiplications.

## 1.9 Object Oriented Finite Element Packages

In this section the numerical packages which are specifically designed for finite element methods are discussed. All the packages are implemented using object oriented paradigm. These packages provide support to C++ developers.

*FemLab.* In [37] (2001) FemLab framework was introduced. The paper has explained in details the implementation of the element objects and construction of the global structures. This toolkit has three main classes FemLab perform analysis of the problem. The Domain class contains problem data. It has DofPlex list which contains the list of degrees of freedom inside finite element mesh. The NodePlex list contains all the nodes in mesh. Each node object has pointers to the corresponding degrees of freedom. The ElementSetPlex list contains all the elements in mesh. The elements have pointer to the corresponding nodes. The MaterialPlex list all the material properties. The Solver class computes the solution and it is independent of the Domain class. The [37] showed the design enable to implement non-linear structural material problem.

*OFELI.* In [79] (2002) has introduced OFELI toolkit and demonstrated how it can be used to implement finite element code by providing a simple elliptic boundary value problem. The package involves reading the mesh, generation of elements and providing solution. It is designed for research or academic purposes where larger packages are not required. OFELI creates the mesh object by reading the mesh file. The mesh file will have the problem description. The mesh object is passed to the matrix and vector objects for stiffness matrix and load vector. The matrix object computes the solution for the system of equations.

*Differential Equations Analysis Library* (DEAL II) [17, 16] is a general purpose finite element library written in C++ for linear and non linear problems. The library uses advance object oriented and data encapsulation to break finite elements into small blocks that can be arranged to fit user requirements. It makes extensive use of templates and STL concepts such as iterators. Deal II focuses on extensibility, simplicity and efficiency [18]. This package is a large group of classes which covers all the aspects of finite element codes as setting up the meshes and finite element spaces, assembling the system of equations, solving this system and post processing. It provides a collection of linear algebra classes for iterative solvers while for direct solver it provides interface to other packages like UMFPACK and HCL. The DEAL II fully supports multi threaded parallelisation for multi-core shared memory machines. It provides interface to PETSc library to add compatibility for distributed memory machines.

*FEniCS* [60] The FEniCS Project is a collection of free, open source, software components with the common goal to enable automated solution of differential equations. The components provide scientific computing tools for working with computational meshes, finite element variational formulations of ordinary and partial differential equations, and numerical linear algebra. It provides C++ and python interface. It provides similar facilities as of DEAL II. FEniCS has defined

user interface as C++ and python library called Dolphin. Dolphin is used for high-level mathematical description of a finite element variational problem.

*Distributed and Unified Numerics Environment*(DUNE) [19] Dune a modular framework for solving partial differential equations with grid-based methods. It is intended to create slim interfaces allowing an efficient use of legacy and/or new libraries. Using C++ techniques Dune allows to use very different implementations of the same concept (e.g., meshes, solvers) using a common interface with a very low overhead. The framework consists of a number of modules which are downloadable as separate packages. DUNE-FEM module defines interfaces for implementing discretisation methods like Finite Element Methods (FEM) and Finite Volume Methods (FV) and Discontinuous Galerkin Methods (DG). It is based on the dune-grid interface library.

*FreeFEM++* [43] is a partial differential equation solver written in C++. It has its own language written in C++ idiom. FreeFem++ has an advanced automatic mesh generator, capable of a posteriori mesh adaptation. It has a general purpose elliptic solver interfaced with fast algorithms such as the multi-frontal method UMFPACK and SuperLU. Hyperbolic and parabolic problems are solved by iterative algorithms prescribed by the user with the high level language of FreeFem++.

*Object Oriented Finite Element Solver* (OOFEM) [74] is a general purpose object oriented FEM code, written in C++. It can solve various linear and nonlinear problems from structural, thermal and fluid mechanics. It particularly includes many material models for nonlinear fracture mechanics of quasibrittle materials, such as concrete. It provides efficient parallel processing support based on domain decomposition and message passing paradigms [75]. The provided direct solvers include symmetric and unsymmetric skyline solver and sparse direct solver and iterative solvers support many sparse storage formats and come with various preconditioners. Interfaces to third party linear solver libraries are available, including

IML, PETSc, SLEPc, and SPOOLES.

## 1.10 Algorithmic Skeletons

The algorithmic skeletons correspond to a high level programming paradigm which hide lower level details from the programmer. Skeletons were introduced by Murray Cole [26]. Skeleton uses common programming patterns and hide lower level complexities of parallel and distributed computing. The skeletons are responsible for achieving high performance by performing optimization of skeleton structure and dynamically adopt to the environment.

The skeletons can be divided into different groups based on their programming paradigm. In *Coordination* approach, the high level language is used to describe the algorithmic behaviour and a host language to handle interaction with the interface. The llc language [29] and the Skeleton Imperative language (Skil) [26] are few examples of augmented languages. They translate skeletal description into the host language and allows the programmer to generate a program by assembling the high level skeletal portion with the host language structure on top of a low level parallel software infrastructure.

The skeletons are introduced in object oriented languages using classes. Based on C++ classes and MPI, the Skeletons in Tokyo (SkeTO) [51], the Munster Skeleton Library (Muesli) [25] and the Malaga-La Laguna-Barcelona (Mallba) library provide data-parallel, task-parallel and resolution skeletons respectively. Other examples such as Calcium, JaSkel, Lithium, Muskel, Quaff and Skandium have focused on distinct skeletons as Java and C++ classes. In object oriented the skeleton libraries rely in the abstraction capabilities of the object oriented host language and they do not require special syntax.

A pattern based C++ library for parallel programming the *Thread Building Blocks*(TBB) [76] has been developed by *Intel*® to take advantage of multi-core architectures. TBB provides parallel patterns including for, reduce, scan, do, sort, and pipeline. It provides abstraction with more control on low level parallelism aspects such as granularity, the possibility to combine with other thread libraries, and direct access to the task scheduler. The *Task Parallel Library* (TPL) [30] provides concurrency support for the Microsoft.NET framework. It provides parallel constructs such as aggregate, do and for to keep different queues balanced.

Skeletons are also deployed as APIs in procedural languages. By procedural calls with in a low level parallel environment they deliver data and task parallel skeletal APIs. The Edinburgh Skeleton Library(eSkel) [21], the Skeleton-based Integrated Environment(SkIE) [14], the Software development System based upon Integrated Skeleton Technology (ASSIST) and the Pisa's Skeleton Library (SKElib) are few of these libraries.

## 1.11 Plan of the thesis

In this thesis the development of the FEDomain package is discussed. The summary of the following chapters are given below:

- The second chapter *Finite Element* discusses the mathematics for the finite element solver algorithms. The implementation of these algorithms are discussed in later chapters.
- The third chapter *FEDomain Interface* discusses the steps taken to develop the user interface. It explains the interfaces for the shared memory and distributed memory architectures, receptively.
- The fourth chapter *FEDomain Shared Memory FE Solvers* discusses the implementation of the FEDomain finite element solvers discussed for the

shared memory architectures.

- The fifth chapter *FEDomain Distributed Memory FE Solvers* discusses the implementation of the FEDomain finite element solvers discussed for the distributed memory architectures.

- The sixth chapter *FEDomain Residual Methods* discusses the implementation of the residual methods in FEDomain package.

- The seventh chapter *FEDomain Shared Memory Residual Methods* discusses the implementation of the residual methods for the shared memory architectures in the FEDomain package.

- The eighth chapter *FEDomain Distributed Memory Residual Methods* discusses the implementation of the residual methods for the distributed memory architectures in the FEDomain package.

- The ninth chapter *Extension to a non-linear solver for the Convection-Diffusion equation* discusses the implementation of the non linear solver in the FEDomain package for Convection-Diffusion problem.

- The tenth chapter *Convection-Diffusion equation Results*, reports the results obtained using the FEDomain package for the convection_diffusion problem using SUPG and SOLD methods.

- The chapter eleven contains the conclusion and future work.

- Lastly, an appendix A (*FEDomain Installation Guide*), explains the user how to use FEDomain package in software. It includes example how to implement Poisson and Elasticity problems.

# Chapter 2

# Finite Elements

This chapter provides an introduction for the finite element methods. It explains how system of equations are developed from the finite elements in mesh. It also includes a summery of methods to compute the solution of these linear system of equations. This topics in this chapter will be referred to in forthcoming chapters.

## 2.1 Finite Element Methods

The Finite Element Method is a numerical procedure that allows one to obtain an approximation to the solution of an ordinary or partial differential equation under appropriate initial and boundary conditions. The finite element method has a solid theoretical foundation (see e.g [22, 31]) and has become one of the most used techniques in the approximation of differential equations. The efficiency of the finite element method relies on two distinct ingredients: the approximation capability of finite elements and the ability of the user to approximate his model in a proper mathematical setting.

Let us consider the following abstract problem.

$$\begin{cases} \text{Find } u \in V \text{ such that} \\ a(u,v) = f(v) \quad \forall v \in V, \end{cases} \tag{2.1}$$

where:

- $V$ is a Hilbert space;
- $a$ is a continuous bilinear form on $V \times V$;
- $f$ is a continuous linear form on $V$.

In many applications the bilinear form $a$ results from the weak formulations of a partial differential equations post on a domain $\Omega \in \mathbb{R}^d$ with boundary conditions on its boundary $\Gamma$.

Three representative examples falling into this framework of the abstract problem (2.1) are the followings:

**The Laplace equation:** Consider the partial differential equation $-\Delta u = f$ in $\Omega$ supplemented with the homogeneous Dirichlet condition $u_{|\Gamma} = 0$. This problem can be reformulated in the form (2.1) by setting

$$\left\{ \begin{array}{c} V = H_0^1(\Omega), \\ a(u,v) = \displaystyle\int_\Omega \nabla u \cdot \nabla v \quad \text{and} \quad f(v) = \int_\Omega fv, \end{array} \right. \tag{2.2}$$

if $f \in L^2(\Omega)$. Where $L^2(\Omega)$, $H^1(\Omega)$, and $H_0^1(\Omega)$ are defined as follows:

$$\begin{aligned} L^2(\Omega) &= \left\{ f : \Omega \longrightarrow \mathbb{R} \mid \int_\Omega |f|^2 < +\infty \right\} \\ H^1(\Omega) &= \left\{ f \in L^2(\Omega) : \bigtriangledown f \in L^2(\Omega)^d \right\} \\ H_0^1(\Omega) &= \left\{ f \in H^1(\Omega) : f \mid_{\partial\Omega} = 0 \right\} \end{aligned}$$

**The Elasticity equation:** Consider the partial differential equation $-\nabla \cdot \sigma(u) = f$ in $\Omega$ supplemented with the homogeneous Dirichlet condition $u_{|\Gamma} = 0$. Where $\sigma(u) = 2\mu \underline{\underline{\varepsilon}}(u) + \lambda tr(\underline{\underline{\varepsilon}}(u))\boldsymbol{I}$ and $\underline{\underline{\varepsilon}} = 0.5(\nabla u + \nabla u^T)$. This problem can be refor-

mulated in the form (2.1) by setting

$$\begin{cases} V = H_0^1(\Omega), \\ a(u,v) = \displaystyle\int_\Omega 2\mu \underline{\underline{\varepsilon}}(u) : \underline{\underline{\varepsilon}}(v) + \lambda tr(\underline{\underline{\varepsilon}}(u)).tr(\underline{\underline{\varepsilon}}(v)) \quad \text{and} \quad f(v) = \displaystyle\int_\Omega fv, \end{cases} \tag{2.3}$$

if $f \in L^2(\Omega)$.

**Advection-Reaction-Diffusion:** Consider the partial differential equation $-\nu\Delta u + \vec{c} \cdot \nabla u + \kappa u = f$ in $\Omega$ supplemented with the homogeneous Dirichlet condition $u_{|\Gamma} = 0$. This problem falls into the above framework by setting

$$\begin{cases} V = H_0^1(\Omega), \\ a(u,v) = \displaystyle\int_\Omega \nu \nabla u \cdot \nabla v + \vec{c} \cdot \nabla u \; v + \kappa uv \quad \text{and} \quad f(v) = \displaystyle\int_\Omega fv, \end{cases} \tag{2.4}$$

if $f \in L^2(\Omega)$ and $\vec{c}$ is a solenoidal function.

The existence and uniqueness of a solution for the weak formulation of the three previous problems follows by classical results from function analysis such as the **Lax-Milgram Lemma** [22] or **Banach-Nečas-Babuška** theorem [38]. Notice that all the previous weak formulations can be modified to consider non homogeneous Dirichlet and Neumann boundary conditions. For more details of the study of weak forms for different problems see Brezzi and Fortin [23], Brenner and Scott [22], Ern and Guermond [31], Girault and Raviart [38].

### 2.1.1 Galerkin Methods

The key idea underlying Galerkin methods is to replace the spaces $V$ by finite-dimensional space $V_h$. The space $V_h$ is termed the solution space as well as *test space*. In its most general form, the Galerkin method constructs an approximation

of $u$ by solving the following approximate problem:

$$\begin{cases} \text{Find } u_h \in V_h \text{ such that} \\ a_h(u_h, v_h) = f_h(v_h) \quad \forall v_h \in V_h. \end{cases} \tag{2.5}$$

Notice that (2.5) involves an approximation $a_h$ to the bilinear form $a$ and an approximation $f_h$ to the linear form $f$.

In order to construct an appropriate approximation space $V_h$, we will use finite element spaces, defined in the sense of Ciarlet as follows: A finite element consists of a triplet $\{K, P, \Sigma\}$ where:

- $K \subseteq \mathbb{R}^d$ is an element domain, for example an edge, triangle or tetrahedron.
- $P$ is a finite-dimensional space of shape functions on $K$, for example polynomials.
- $\Sigma$ are degrees of freedom, for example values at the vertices of $K$.

### 2.1.2 The Linear System

The approximation problem (2.5) is simply a linear system. To see this let:

$$N = dim(V_h) \quad \text{with a basis } \{\phi_1, \phi_2, ....., \phi_N\}. \tag{2.6}$$

In the framework of finite element methods, the functions $\{\phi_1, \phi_2, ....., \phi_N\}$ can be taken to be the *global shape functions* in $V_h$ [31, 22].

Consider the expansion of $u_h$ in the basis of $V_h$

$$u_h = \sum_{i=1}^{N} U_i \phi_i, \tag{2.7}$$

and introduce the coordinate vector of $u_h$, $U = [U_1, U_2, ..., U_N]^T$. In this thesis, the degrees of freedom are considered as the values of the functions on the nodes

of the mesh. Let $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ be the *stiffness matrix* with entries

$$[\boldsymbol{A}]_{ij} = a_h(\phi_j, \varphi_i), \quad 1 \leq i, j \leq N, \tag{2.8}$$

and let $\vec{b} \in \mathbb{R}^N$ be the vector with components

$$[b]_i = f_h(\varphi_i), \quad 1 \leq i \leq N. \tag{2.9}$$

It is readily verified that

$$(u_h \, solves \; (2.5)) \iff (\boldsymbol{A}u = b). \tag{2.10}$$

### 2.1.3 Partitioning the domain

Let $\Omega$ be a bounded domain in $\mathbb{R}^d$, where $d = 1, 2$ or $3$ with boundary $\Gamma$. Let $\mathcal{P} = \{K\}$ be a partitioning of $\Omega$ into elements such that

$$\Omega = \bigcup_{K\in\mathcal{P}} K$$

and the non-empty intersection of a distinct pair of elements is a single common point, edge, or face of both elements. The most commonly used element types are triangle, quadrilaterals when $\Omega$ is $\mathbb{R}^2$ and tetrahedra and hexahedra when $\Omega$ is $\mathbb{R}^3$, for polygon and polyhedral domains

For a fix partition $\mathcal{P}$ [1] let:

- $K$ denote an element of the partition;
- $\#\mathcal{P}$ be the total number of elements in $\mathcal{P}$;
- $\mathcal{N}$ index the set $\{\boldsymbol{x}_n\}_{n\in\mathcal{N}}$ of all global degrees of freedom (DOFs) $\boldsymbol{x}_n$ on $\mathcal{P}$;

[1] represents set of all elements in $\Omega$.

- $\mathcal{N}_K$ index the set $\{\boldsymbol{x}_n\}_{n\in\mathcal{N}_K}$ of local degrees of freedom (DOFs) $\boldsymbol{x}_n$ on $K$;
- $N$ is the total number of DOFs in $\mathcal{P}$;
- $N_K$ is the total number of DOFs in $K$.

The following functions maps $K$ local DOFs numbering to global DOFs numbering respectively:

$$\begin{array}{rccc} \lambda^K & : & \mathcal{N}_K & \longrightarrow & \mathcal{N} \\ & & i & \mapsto & \lambda^K(i) = j. \end{array} \tag{2.11}$$

Each DOF $\boldsymbol{x_i}$ in $\mathcal{P}$ is associated with a basis function $\varphi_i$, defined on $\Omega$ with the property that

$$\varphi_i(\boldsymbol{x}_j) = \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases} \tag{2.12}$$

In order to solve the problem (2.10), first we mention that in general, $a_h$ and $f_h$ take the form

$$a_h(u_h, v_h) = \int_\Omega \mathcal{B}_h(\boldsymbol{x}, u_h, v_h) \quad \text{and} \quad f_h(v_h) = \int_\Omega \mathcal{L}_h(\boldsymbol{x}, f, v_h),$$

where $\mathcal{B}_h$ and $\mathcal{L}_h$ are operators. Notice that, from the three previous problem we have that

$$\begin{array}{rl} \textbf{Poisson:} & \mathcal{B}_h(u_h, v_h) = \nabla u \cdot \nabla v, \\ \textbf{Elasticity:} & \mathcal{B}_h(u_h, v_h) = 2\mu\ \underline{\underline{\varepsilon}}(u) : \underline{\underline{\varepsilon}}(v) + \lambda\ tr(\underline{\underline{\varepsilon}}(u)).tr(\underline{\underline{\varepsilon}}(v)), \\ \textbf{Advection-Reaction-Diffusion:} & \mathcal{B}_h(u_h, v_h) = \nu\nabla u \cdot \nabla v + \vec{c}\cdot \nabla u\ v + \kappa uv. \end{array}$$

Now, due to the properties of the basis function and (2.8), it follows that

$$[\boldsymbol{A}]_{ij} = a_h(\phi_j, \phi_i) = \int_\Omega \mathcal{B}_h(\boldsymbol{x}, \phi_j, \phi_i) = \int_{\Omega_{ij}} \mathcal{B}_h(\boldsymbol{x}, \phi_j, \phi_i), \tag{2.13}$$

where

$$\Omega_{ij} = \Omega_i \cap \Omega_j \subset \mathcal{P}.$$

Then

$$\begin{aligned}
[\boldsymbol{A}]_{ij} &= \int_{\Omega_{ij}} \mathcal{B}_h(\boldsymbol{x}, \phi_j, \phi_i), \\
&= \sum_{K \in \Omega_{ij}} \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_j, \phi_i), \\
&= \sum_{K \in \Omega_{ij}} \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(s)}, \phi_{\lambda^K(t)}), \\
&= \sum_{K \in \Omega_{ij}} \mathcal{B}^K_{\lambda^K(s)\lambda^K(t)}.
\end{aligned}$$

where

$$\lambda^K(s) = j, \ \lambda^K(t) = i \quad \text{and} \quad s, t \in \mathcal{N}_K.$$

For a fixed element $K$, let us define $\boldsymbol{A}_K \in \mathbb{R}^{N_K \times N_K}$, a local stiffness matrix with local DOFs numbering, with entries

$$[\boldsymbol{A}_K]_{st} = \left( \mathcal{B}^K_{\lambda^K(s)\lambda^K(t)} \right)_{s,t \in \mathcal{N}_K}. \tag{2.14}$$

Now, let $\boldsymbol{\Lambda}_E \in \mathbb{R}^{N_K \times N}$ be a connectivity matrix with entries

$$[\boldsymbol{\Lambda}_K]_{ij} = \begin{cases} 1 & \text{if } \lambda^K(i) = j, \\ 0 & \text{elsewhere.} \end{cases} \tag{2.15}$$

$\boldsymbol{\Lambda}_K$ is used to map data related to $K$ stored using $\mathcal{N}_K$ into global numbering $\mathcal{N}$. The relation between the local stiffness matrix $\boldsymbol{A}_K$ and global stiffness matrix $\boldsymbol{A}$ can be represented as:

$$\boldsymbol{A} = \sum_{K \in \Omega} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K. \tag{2.16}$$

The same process can be repeated for the right hand side. In fact,

$$[b]_i = f_h(\varphi_i) = \int_\Omega \mathcal{L}_h(\boldsymbol{x}, f, \phi_i) = \int_{\Omega_i} \mathcal{L}_h(\boldsymbol{x}, f, \phi_i)$$

where

$$\Omega_i = supp\{\phi_i\} \subset \mathcal{P},$$

with $supp\{\varphi_i\}$ is the set of elements where the function $\varphi_i$ is not zero-valued, then

$$\begin{aligned}
[b]_i &= \int_{\Omega_i} \mathcal{L}_h(\boldsymbol{x}, f, \phi_i) \\
&= \sum_{K \in \Omega_i} \int_K \mathcal{L}_h(\boldsymbol{x}, f, \phi_i) \\
&= \sum_{K \in \Omega_i} \int_K \mathcal{L}_h(\boldsymbol{x}, f, \phi_{\lambda^K(t)}) \\
&= \sum_{K \in \Omega_i} \mathcal{L}^K_{\lambda^K(t)},
\end{aligned}$$

For a fixed element $K$, let $b_K \in \mathbb{R}^{N_K \times 1}$ be a local load vector with local DOFs numbering, with entries

$$[b_K]_t = \left(\mathcal{L}^K_{\lambda^K(t)}\right)_{t \in \mathcal{N}_K}. \tag{2.17}$$

Hence, using the definition of $\boldsymbol{\Lambda}_K$, it follows that

$$\vec{b} = \sum_{K \in \Omega} \boldsymbol{\Lambda}_K^T \vec{b}_K. \tag{2.18}$$

In order to clarify all the previous definition let us consider the following simple mesh shown in Figure 2.1.

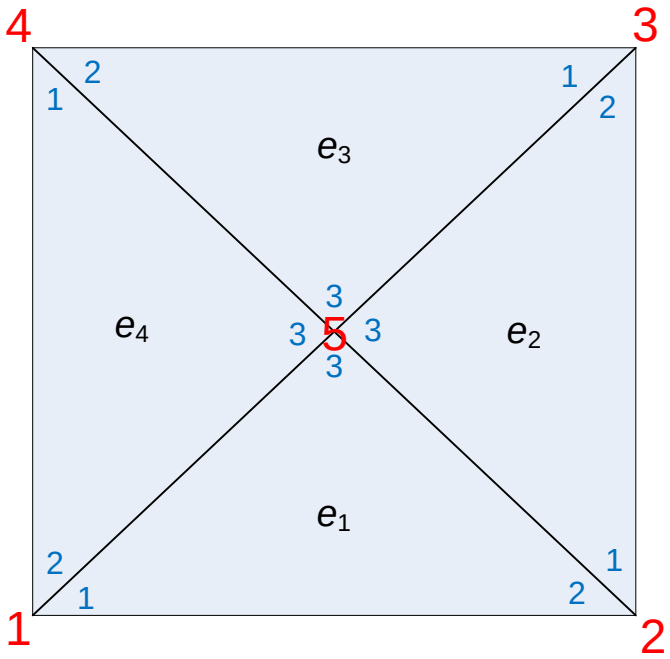



Figure 2.1: Simple mesh with 4 elements and 5 nodes (DOFs), where we mark in red the global numbering of the nodes (DOFs) and in blue the local nodes (DOFs).

Then, for this mesh we have:

$$\boldsymbol{A} = \begin{bmatrix} \int_{\Omega_{11}} \mathcal{B}_h(\boldsymbol{x}, \phi_1, \phi_1) & \cdots & \int_{\Omega_{15}} \mathcal{B}_h(\boldsymbol{x}, \phi_5, \phi_1) \\ \vdots & \ddots & \vdots \\ \int_{\Omega_{51}} \mathcal{B}_h(\boldsymbol{x}, \phi_1, \phi_5) & \cdots & \int_{\Omega_{55}} \mathcal{B}_h(\boldsymbol{x}, \phi_5, \phi_5) \end{bmatrix}, \tag{2.19}$$

and

$$\vec{b} = \begin{bmatrix} \int_{\Omega_1} \mathcal{L}_h(\boldsymbol{x}, f, \phi_1) \\ \vdots \\ \int_{\Omega_5} \mathcal{L}_h(\boldsymbol{x}, f, \phi_5) \end{bmatrix}, \tag{2.20}$$

where

$$\Omega_{11} = e_1 \cup e_4, ..., \ \Omega_{15} = e_1 \cup e_4, ... \text{ and } \Omega_{55} = e_1 \cup e_2 \cup e_3 \cup e_4, \tag{2.21}$$

and

$$\Omega_1 = e_1 \cup e_4, ... \text{ and } \Omega_5 = e_1 \cup e_2 \cup e_3 \cup e_4. \tag{2.22}$$

Now, the local stiffness matrices are

$$\boldsymbol{A}_K = \begin{bmatrix} \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(1)}, \phi_{\lambda^K(1)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(2)}, \phi_{\lambda^K(1)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(3)}, \phi_{\lambda^K(1)}) \\ \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(1)}, \phi_{\lambda^K(2)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(2)}, \phi_{\lambda^K(2)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(3)}, \phi_{\lambda^K(2)}) \\ \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(1)}, \phi_{\lambda^K(3)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(2)}, \phi_{\lambda^K(3)}) & \int_K \mathcal{B}_h(\boldsymbol{x}, \phi_{\lambda^K(3)}, \phi_{\lambda^K(3)}) \end{bmatrix} \tag{2.23}$$

and the local load vector is

$$\vec{b}_K = \begin{bmatrix} \int_K \mathcal{L}_h(\boldsymbol{x}, f, \phi_{\lambda^K(1)}) \\ \int_K \mathcal{L}_h(\boldsymbol{x}, f, \phi_{\lambda^K(2)}) \\ \int_K \mathcal{L}_h(\boldsymbol{x}, f, \phi_{\lambda^K(3)}) \end{bmatrix}, \tag{2.24}$$

where

$$\begin{aligned} &\lambda^{e_1}(1) = 1, \quad \lambda^{e_2}(1) = 2, \quad \lambda^{e_3}(1) = 3, \quad \lambda^{e_4}(1) = 4, \\ &\lambda^{e_1}(2) = 2, \quad \lambda^{e_2}(2) = 3, \quad \lambda^{e_3}(2) = 4, \quad \lambda^{e_4}(2) = 1, \\ &\lambda^{e_1}(3) = 5, \quad \lambda^{e_2}(3) = 5, \quad \lambda^{e_3}(3) = 5, \quad \lambda^{e_4}(3) = 5. \end{aligned} \tag{2.25}$$

Finally, the connectivity matrices are given by

$$\boldsymbol{\Lambda}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, ... \text{ and } \boldsymbol{\Lambda}_4 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

### 2.1.4 Partitioning of partitioning

Let $\mathcal{P}_1, ..., \mathcal{P}_N$ be a partitioning of $\mathcal{P}$ into disjoint subsets such that

$$\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup .... \cup \mathcal{P}_N \quad where \quad \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \quad if \quad i \neq j.$$

Let $\#P_i$ be the number of elements and $N_p$ represents the number of degrees of freedom present in $\mathcal{P}_i$. Let $\mathcal{G}_i$ be the set of dofs on $\mathcal{P}_i$.

$$\mathcal{G}_i = range(\Lambda_K) \quad \forall K \in \mathcal{P}_i \tag{2.26}$$

Let $\mathcal{N}_P$ be the set of degrees of freedom of $\mathcal{P}$ associated to $\mathcal{P}_i$. For $\mathcal{P}_i$, let define $\boldsymbol{A}_P \in \mathbb{R}^{N_p \times N_p}$ be a local stiffness matrix with local DOFs numbering. $\boldsymbol{\Lambda}_P \in \mathbb{R}^{N_p \times N}$ be a connectivity matrix with entries

$$[\boldsymbol{\Lambda}_P]_{ij} = \begin{cases} 1 & \text{if } \lambda^P(i) = j, \\ 0 & \text{elsewhere.} \end{cases} \tag{2.27}$$

The $\boldsymbol{\Lambda}_P$ is used to map $\mathcal{P}_i$ local data stored using $\mathcal{N}_P$ into global numbering $\mathcal{N}$. The relation between the local stiffness matrix $\boldsymbol{A}_P$ and global stiffness matrix $\boldsymbol{A}$ can be represented as

$$\boldsymbol{A} = \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{\Lambda}_P^T \boldsymbol{A}_P \boldsymbol{\Lambda}_P. \tag{2.28}$$

Let $\vec{b}_P \in \mathbb{R}^{N_P \times 1}$ be the local load vector with local DOFs numbering. The relation between $\vec{b}$ and $\vec{b}_P$ can be explained as

$$\vec{b} = \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{\Lambda}_P^T \vec{b}_P. \tag{2.29}$$

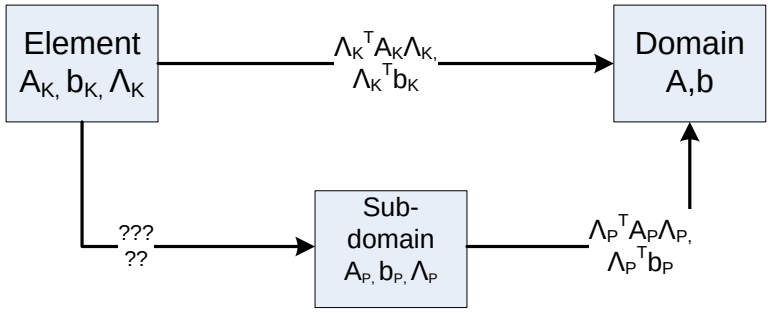



Figure 2.2: Mapping between the elements, sub-partitions and partitions.

The $\mathbf{\Lambda}_K$ and $\mathbf{\Lambda}_P$ are used to map elements and sub-partitions data from $\mathcal{N}_K$ to $\mathcal{N}$ and from $\mathcal{N}_P$ to $\mathcal{N}$, respectively. Figure 2.2 shows there is no $\mathcal{N}_K$ to $\mathcal{N}_P$ mapping information present in $K$. The information is required by $P_i$ to collect it's elements data and store as $\mathcal{N}_P$. Let $\lambda_P^K$ maps $K$ data from $\mathcal{N}_K$ to $\mathcal{N}_P$.

$$\begin{array}{rccc} \lambda_P^K & : & \mathcal{N}_K & \longrightarrow & \mathcal{N}_P \\ & & i & \mapsto & \lambda_P^K(i) = j. \end{array} \tag{2.30}$$

Let $l \in \mathcal{N}_K$ and $m$ can be defined such that

$$\lambda_K^P(l) = m,$$

then

$$\lambda_P(\lambda_K^P(l)) = \lambda_P(m) = \lambda_K(l).$$

Let $\mathbf{\Lambda}_{K_P} \in \mathbb{R}^{\mathcal{N}_K \times \mathcal{N}_P}$ is connectivity matrix which maps element data to sub-partition data. It is defined as

$$[\mathbf{\Lambda}_{K_P}]_{ij} = \begin{cases} 1 & \text{if } \lambda_K^P(i) = j, \\ 0 & \text{elsewhere.} \end{cases} \tag{2.31}$$

Let $\mathcal{P}_i \in \mathcal{P}$ data can be defined as

$$\boldsymbol{A}_P = \sum_{K \in \mathcal{P}_i} \mathbf{\Lambda}_{K_P}^T \boldsymbol{A}_K \mathbf{\Lambda}_{K_P}, \tag{2.32}$$

$$\vec{b}_P = \sum_{K \in \mathcal{P}_i} \mathbf{\Lambda}_{K_P}^T \vec{b}_K. \tag{2.33}$$

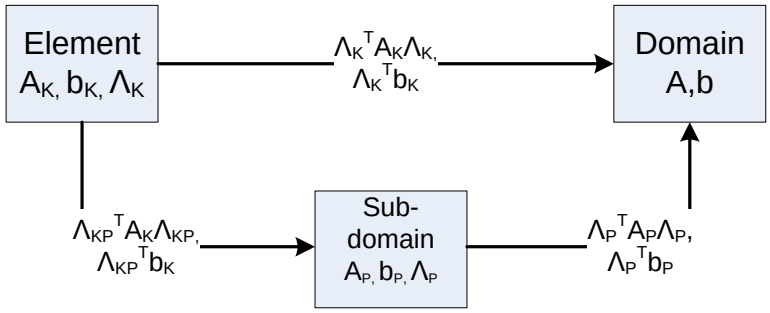



Figure 2.3: Mapping between the elements, sub-partitions and partitions.

## 2.2 Solution of Linear Systems

Finite element methods lead to large linear system. We need to describe the main methods to solve these systems. The are two type of methods direct and iterative. These methods have their advantages and disadvantages according to their computation resources and timing.

### 2.2.1 Direct Solution Methods

The direct methods for solving linear system of equations factorise stiffness matrix and once the factorization is complete, the system of equations can be solved efficiently for multiple right hand side (RHS) vectors by forward elimination and back substitution. The sparsity of the system is used to minimize the arithmetic operations 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.

In general a sparse direct solver is composed of three phases, stated as follows:

- *Analysis and symbolic factorization phase*: determines the pivot ordering required to minimize the time and storage needed by the solution. The non zero structures of the factors, which contains the original non zero elements in

$\boldsymbol{A}$ as well as filled elements, are determined and computed without referring to numerical values of $\boldsymbol{A}$. The minimum degree algorithm [59] and nested disection algorithm [53] are commonly used for symbolic analysis of $\boldsymbol{A}$.

- *Numerical factorisation phase*: the numerical values of factors are computed. The selection of the matrix decomposition algorithm depends on the nature of $\boldsymbol{A}$. The oldest decomposition method is *Gaussian Elimination*, where $\boldsymbol{A}$ is factored into lower and upper triangular matrices. In *LU decomposition or factorization* $\boldsymbol{A}$ is factored into $\boldsymbol{L}$ and $\boldsymbol{U}$. If $\boldsymbol{A}$ is symmetric and positive definite [2], the *Cholesky factorization* is used which decompose it as $\boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is a lower triangle matrix. For structurally symmetric [3] $\boldsymbol{A}$ *free root Cholesky factorization* is used, it decompose into $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$. In case of asymmetric $\boldsymbol{A}$, the *QR factorization* is used to decompose into orthogonal matrix $\boldsymbol{Q}$ and upper triangle matrix $\boldsymbol{R}$.

- *Solution phase*: performs forward elimination and back substitution using the factor matrices found in the Numerical factorization phase. In case of LU factorization following steps are performed:

$$\begin{aligned} \boldsymbol{L}y &= \vec{b} \quad \text{forward elimination for } y \\ \boldsymbol{U}\alpha &= y \quad \text{backward substitution for } \alpha \end{aligned}$$

The computational cost of solving the $2D$ problem is $\mathcal{O}(N^{2/3})$ and $3D$ problem is $\mathcal{O}(N^{5/3})$. The direct methods are preferred if the number of right hand sides in system of equations is so large that the decomposition of $\boldsymbol{A}$ takes relatively small time.

[2] A matrix $\boldsymbol{A}$ is symmetric if $\boldsymbol{A} = \boldsymbol{A}^T$

[3] If the nonzero pattern of $\boldsymbol{A}$ is symmetric, a matrix $\boldsymbol{A}$ is structurally symmetric ; that is, $[\boldsymbol{A}]_{ij} \neq 0$ if and only if $[\boldsymbol{A}]_{ji} \neq 0$ for all $i$ and $j$

### 2.2.2 Iterative Solution Methods

Iterative algorithms solve linear equations while only performing multiplications by $\boldsymbol{A}$, and performing a few vector operations. Unlike the direct methods which are based on elimination, the iterative algorithms do not get exact solutions. The accuracy of the solution $\vec{\alpha}$ is directly proportional to the number of iterations. The advantage of iterative methods is they require less memory for storage by avoiding fill-ins and are often faster than the direct methods. As a disadvantage they do not guarantee to provide an approximate solution ( according to given tolerance ) in a specific number of steps.

The iterative method solution starts by guessing the approximate solution $\alpha_g$ of the system of linear equations, which is used to calculate an error vector $e$ as:

$$\begin{aligned} e &= \alpha - \alpha_g, \\ \boldsymbol{A}e &= \boldsymbol{A}\alpha - \boldsymbol{A}\alpha_g, \\ \boldsymbol{A}e &= \vec{b} - \boldsymbol{A}\alpha_g, \\ \boldsymbol{A}e &= r, \\ e &= \boldsymbol{A}^{-1}r. \end{aligned} \tag{2.34}$$

The pre-conditioner matrix $\boldsymbol{C}$ are used in place of $\boldsymbol{A}^{-1}$ in equation (2.34). Different iterative methods have their own definition of pre-conditioner matrix $\boldsymbol{C}$. The *Jacobi method* has one of the simplest forms of preconditioning matrix, the pre-conditioner is chosen to be the inverse of diagonal of $\boldsymbol{A}$.

$$\begin{aligned} \boldsymbol{D} &= diagonal(\boldsymbol{A}) \\ \boldsymbol{C} &= \boldsymbol{D}^{-1} \\ \boldsymbol{C} &\approx \boldsymbol{A}^{-1} \end{aligned}$$

Alternative iterative method like *Richard method* uses an identity matrix multiplied with suitable constant values as $C$. The $e$ is calculated by multiplying pre-conditioner matrix with $r$.

$$e \approx \boldsymbol{C}r \tag{2.35}$$

At the end of each iteration a new approximation solution is calculated by adding $e$ into $\alpha_g$.

$$\alpha_g += e \tag{2.36}$$

In next iteration the updated $\alpha_g$ is used to calculate $r$. The process repeats itself for every iteration to compute new approximate solution and residual. Mostly but not always in each iteration the approximation solutions converges toward the actual solutions and error reduces. The stopping mechanism is required to check the solution has converged to the required tolerance and stop the iterations. Few example of iterative methods are Jacobi, Gauss-Seidel, Generalized Minimal Residual (GMRES), Quasi-Minimal Residual (QMR) and Conjugate Gradient (CG). Each method has different method of calculating error but every method calculates residual.

The residual vector can be calculated by two methods.

#### 2.2.2.1 Full Assembly

In full assembly (FA) stiffness matrix and load vector of the system are assembled. The assembled data are used to compute residual vector in each iteration using following (2.37)

$$r = \vec{b} - \boldsymbol{A}\alpha_g. \tag{2.37}$$

#### 2.2.2.2 Element By Element

In element by element (EBE) method, the residual vector is calculated at the element level. This process can be implemented in parallel for the distributed architectures. By replacing the $\boldsymbol{A}$ and $\vec{b}$ from (2.16) and (2.18) the following equations are generated.

$$\begin{aligned} r &= \sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T \vec{b}_K - \sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K \alpha_g, \\ &= \sum_{K\in\mathcal{P}} (\boldsymbol{\Lambda}_K^T \vec{b}_K - \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K \alpha_g), \\ &= \sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T (\vec{b}_K - \boldsymbol{A}_K \boldsymbol{\Lambda}_K \alpha_g). \end{aligned} \tag{2.38}$$

Let $r_K$ be the element $K$ residual vector defined as

$$r_K = \vec{b}_K - \boldsymbol{A}_K \boldsymbol{\Lambda}_K \alpha_g, \tag{2.39}$$

then

$$r = \sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T (r_K). \tag{2.40}$$

### 2.2.3 Static Condensation

Static condensation is a method of solving the system of linear equations by decomposing into smaller system of linear equations. The smaller systems of equations are achieved by dividing the domain into smaller domains. ($\Omega = \Omega_1 \cup \Omega_2 \cup ... \cup \Omega_N$) as shown in Figure 2.4. The domain is divided into 4 sub-domains. This subdivision of $\Omega$ generates the following sub-domains of the partition $\mathcal{P}$:

$$\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup ... \cup \mathcal{P}_N. \tag{2.41}$$

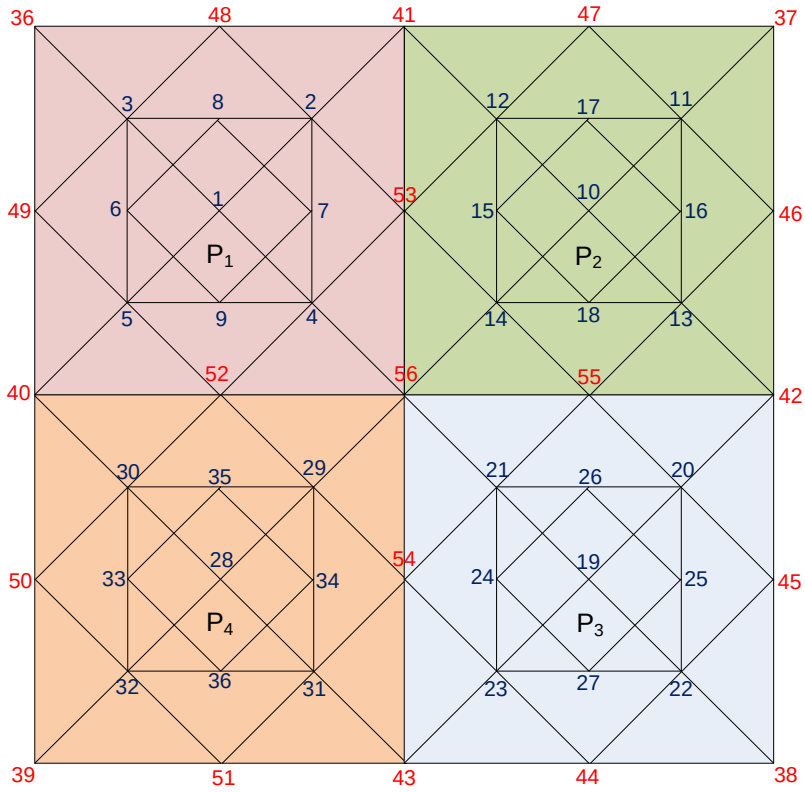



Figure 2.4: The mesh with 4 partitions and each partition has 9 internal nodes and 8 boundary nodes.

Matrix $\boldsymbol{A}$ can be represented in terms of elements as in (2.16) and in terms of sub-partitions as in (2.28). The relation between these equations can be represented as

$$\begin{aligned} \boldsymbol{A} &= \sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K, \\ &= \sum_{i=1}^{N} \sum_{K\in\mathcal{P}_i} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K, \\ &= \sum_{i=1}^{N} \boldsymbol{\Lambda}_{P_i}^T \boldsymbol{A}_{P_i} \boldsymbol{\Lambda}_{P_i}. \end{aligned} \tag{2.42}$$

From (2.42) and (2.28) it is observed

$$\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{A}_{P_j} \boldsymbol{\Lambda}_{P_j} = \sum_{K \in \mathcal{P}_j} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K \tag{2.43}$$

The following theorem state the relation between the connectivity mapping of the elements and the sub partitions.

**Theorm 1.** $\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{A}_{P_j} \boldsymbol{\Lambda}_{P_j} = \sum_{K \in \mathcal{P}_j} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K$

*Proof.* $\boldsymbol{\Lambda}_{P_j}$ maps $\mathcal{P}_j$'s data from $\mathcal{N}_{P_j}$ to $\mathcal{N}$. The $\boldsymbol{\Lambda}_{P_j}^T \in \mathbb{R}^{\mathcal{N} \times \mathcal{N}_{P_j}}$ maps the data from the $\mathcal{N}$ to $\mathcal{N}_{P_j}$. From the definition of the $\boldsymbol{\Lambda}_{P_j}$

$$\boldsymbol{\Lambda}_{P_j} \boldsymbol{\Lambda}_{P_j}^T = \boldsymbol{I}, \tag{2.44}$$

where $\boldsymbol{I}$ is the identity matrix of dimension $N_{P_j} \times N_{P_j}$. The $\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{\Lambda}_{P_j}$ generates a matrix of size $N \times N$. This is a sparse matrix in which most of the rows are empty. The rows which ids are identical to the sub partition DOFs global ids have a single value at diagonal. $\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{\Lambda}_{P_j}$ has only $N_{P_j}$ non zero values all of which are set to 1 and lies on matrix diagonal. Let suppose the DOFs present in $\mathcal{P}_j$ are allocated with 1 to $N_{P_j}$ consecutive global DOF indexes. Than:

$$\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{\Lambda}_{P_j} = \begin{array}{c} \begin{array}{cc} N_{P_j} & N - N_{P_j} \end{array} \\ \left[\begin{array}{c|c} \mathrm{I} & 0 \\ \hline 0 & 0 \end{array}\right] \end{array}. \tag{2.45}$$

Multiplying both sides of (2.43) to the left by $\boldsymbol{\Lambda}_{P_j}$ and to the right by $\boldsymbol{\Lambda}_{P_j}^T$. We

have

$$\boldsymbol{\Lambda}_{P_j}\boldsymbol{\Lambda}_{P_j}^T\boldsymbol{A}_{P_j}\boldsymbol{\Lambda}_{P_j}\boldsymbol{\Lambda}_{P_j}^T = \sum_{K\in\mathcal{P}_j}\boldsymbol{\Lambda}_{P_j}\boldsymbol{\Lambda}_K^T\boldsymbol{A}_K\boldsymbol{\Lambda}_K\boldsymbol{\Lambda}_{P_j}^T, \tag{2.46}$$

and using (2.44) we get

$$\boldsymbol{A}_{P_j} = \sum_{K\in\mathcal{P}_j}\boldsymbol{\Lambda}_{P_j}\boldsymbol{\Lambda}_K^T\boldsymbol{A}_K\boldsymbol{\Lambda}_K\boldsymbol{\Lambda}_{P_j}^T. \tag{2.47}$$

Equation (2.47) maps the $\boldsymbol{A}_K$ (where $K \in \mathcal{P}_j$) to $\boldsymbol{A}_{P_j}$. To prove the converse of the equation is true, (2.47) multiply with (2.45).

$$\boldsymbol{\Lambda}_{P_j}^T\boldsymbol{A}_{P_j}\boldsymbol{\Lambda}_{P_j} = \sum_{K\in\mathcal{P}_j}\boldsymbol{\Lambda}_{P_j}^T\boldsymbol{\Lambda}_{P_j}\boldsymbol{\Lambda}_K^T\boldsymbol{A}_K\boldsymbol{\Lambda}_K\boldsymbol{\Lambda}_{P_j}^T\boldsymbol{\Lambda}_{P_j}. \tag{2.48}$$

Let

$$\tilde{\boldsymbol{A}}_{P_j} = \sum_{K\in\mathcal{P}_j}\boldsymbol{\Lambda}_K^T\boldsymbol{A}_K\boldsymbol{\Lambda}_K, \tag{2.49}$$

$$= \begin{array}{c} \begin{array}{cc} N_{P_j} & N - N_{P_j} \end{array} \\ \left[\begin{array}{c|c} \tilde{\boldsymbol{A}}_{P_j} & 0 \\ \hline 0 & 0 \end{array}\right] \end{array}.$$

By replacing the $\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{\Lambda}_{P_j}$ and $\tilde{\boldsymbol{A}}_{P_j}$ in (2.48) we have

$$\boldsymbol{\Lambda}_{P_j}^T \boldsymbol{A}_{P_j} \boldsymbol{\Lambda}_{P_j} = \left[\begin{array}{c|c} \mathbb{I} & 0 \\ \hline 0 & 0 \end{array}\right]\left[\begin{array}{c|c} \tilde{\boldsymbol{A}}_{P_j} & 0 \\ \hline 0 & 0 \end{array}\right]\left[\begin{array}{c|c} \mathbb{I} & 0 \\ \hline 0 & 0 \end{array}\right],$$

$$= \left[\begin{array}{c|c} \tilde{\boldsymbol{A}}_{P_j} & 0 \\ \hline 0 & 0 \end{array}\right]. \tag{2.50}$$

which finishes the proof. □

For a fix sub-domain $\mathcal{P}_j$, $\vec{b}_{P_j} \in \mathbb{R}^{\mathcal{N}_{P_j} \times 1}$

$$\begin{aligned} \vec{b} &= \sum_{K \in \mathcal{P}} \boldsymbol{\Lambda}_K^T \vec{b}_K, \\ &= \sum_{\mathcal{P}_j \in \mathcal{P}} \sum_{K \in \mathcal{P}_i} \boldsymbol{\Lambda}_K^T \vec{b}_K, \\ &= \sum_{\mathcal{P}_j \in \mathcal{P}} \boldsymbol{\Lambda}_{P_j}^T \vec{b}_{P_j}. \end{aligned} \tag{2.51}$$

From (2.51) following equation is emerged.

$$\tilde{\vec{b}}_{P_j} = \boldsymbol{\Lambda}_{P_j}^T \vec{b}_{P_j} = \sum_{K \in \mathcal{P}_j} \boldsymbol{\Lambda}_K^T \vec{b}_K, \tag{2.52}$$

$$= \begin{bmatrix} \tilde{b}_{P_j} \\ 0 \end{bmatrix} \begin{matrix} N_{P_j} \\ N - N_{P_j} \end{matrix}. \tag{2.53}$$

To prove the relation between the $\vec{b}_P$ and $\vec{b}_K$ we multiply (2.52) by $\mathbf{\Lambda}_{P_j}$ to arrive at

$$\mathbf{\Lambda}_{P_j}\mathbf{\Lambda}_{P_j}^T\vec{b}_{P_j} = \sum_{K\in\mathcal{P}_j}\mathbf{\Lambda}_{P_j}\mathbf{\Lambda}_K^T\vec{b}_K, \tag{2.54}$$

where $\mathbf{\Lambda}_{P_j}\mathbf{\Lambda}_{P_j}^T = I$, so (2.54) becomes

$$\vec{b}_{P_j} = \sum_{K\in\mathcal{P}_j}\mathbf{\Lambda}_{P_j}\mathbf{\Lambda}_K^T\vec{b}_K. \tag{2.55}$$

To prove the converse test (2.55) is multiplied by $\mathbf{\Lambda}_{P_j}^T$

$$\begin{aligned}
\mathbf{\Lambda}_{P_j}^T\vec{b}_{P_j} &= \sum_{K\in\mathcal{P}_j}\mathbf{\Lambda}_{P_j}^T\mathbf{\Lambda}_{P_j}\mathbf{\Lambda}_K^T\vec{b}_K, \\
&= \mathbf{\Lambda}_{P_j}^T\mathbf{\Lambda}_{P_j}\sum_{K\in\mathcal{P}_j}\mathbf{\Lambda}_K^T\vec{b}_K, \\
&= \left[\begin{array}{c|c} \mathrm{I} & 0 \\ \hline 0 & 0 \end{array}\right]\left[\begin{array}{c} \tilde{\vec{b}}_P \\ \\ 0 \end{array}\right].
\end{aligned} \tag{2.56}$$

From equation (2.45) the equation (2.56) becomes

$$\mathbf{\Lambda}_{P_j}^T\vec{b}_{P_j} = \sum_{K\in\mathcal{P}_j}\mathbf{\Lambda}_K^T\vec{b}_K \tag{2.57}$$

Some times the system of linear equations can not be computed on single computer due to memory or computation resource restrictions. Static condensation method is used to compute the solution of a linear system of equations on distributed computer systems. Each partition is allocated to a unique computational node and a separate computational node is used to compute the solution of boundary DOFs. This method allows division of actual systems of equations into smaller

systems which can be solved on each node with some communication between them.

Let us assume that for a partitioned mesh all the internal DOFs of each partition are allocated consecutive global numbers. The interface DOFs are numbered after all the partitions internal DOFs. Each partition stiffness matrix and load vector can be represented in term of internal DOFs and interface DOFs, as follows

$$\boldsymbol{A}_{P_j} = \begin{bmatrix} \boldsymbol{A}_{ii} & \boldsymbol{A}_{ib} \\ \boldsymbol{A}_{bi} & \boldsymbol{A}_{bb}^{P_j} \end{bmatrix} \tag{2.58}$$

$$\vec{b}_{P_j} = \begin{bmatrix} \vec{b}_i \\ \vec{b}_k^{P_j} \end{bmatrix} \tag{2.59}$$

In (2.58), $\boldsymbol{A}_{ii}$ and $\boldsymbol{A}_{bb}^{P_j}$ are the stiffness matrices connected to the internal and boundary DOFs of $\mathcal{P}_j$ respectively. While $\boldsymbol{A}_{ib}$ and $\boldsymbol{A}_{bi}$ matrices represents the internal and boundary DOFs interaction. In (2.59), $\vec{b}_i$ and $\vec{b}_B^{P_j}$ are the load vectors connected to the internal and boundary DOFs of $\mathcal{P}_j$ receptively. The full system of equations can be written as

$$\begin{bmatrix} \boldsymbol{A}_{11} & \dots & \boldsymbol{0} & \boldsymbol{A}_{1b}\boldsymbol{\Lambda}_{bB}^{1} \\ \vdots & \ddots & \vdots & \vdots \\ \boldsymbol{0} & \dots & \boldsymbol{A}_{NN} & \boldsymbol{A}_{Nb}\boldsymbol{\Lambda}_{bB}^{N} \\ (\boldsymbol{\Lambda}_{bB}^{1})^T\boldsymbol{A}_{b1} & \dots & (\boldsymbol{\Lambda}_{bB}^{N})^T\boldsymbol{A}_{bB} & \sum_{k=1}^{N}(\boldsymbol{\Lambda}_{bB}^{k})^T\boldsymbol{A}_{bb}^{k}\boldsymbol{\Lambda}_{bB}^{k} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \\ \alpha_B \end{bmatrix} = \begin{bmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_N \\ \sum_{k=1}^{N}(\boldsymbol{\Lambda}_{bB}^{k})^T\vec{b}_b^{k} \end{bmatrix} \tag{2.60}$$

Lets

$$\begin{aligned}
\boldsymbol{A}_{iB} &= \boldsymbol{A}_{ib}\boldsymbol{\Lambda}^{i}_{bB} && (2.61)\\
\boldsymbol{A}^{i}_{BB} &= (\boldsymbol{\Lambda}^{i}_{bB})^{T}\boldsymbol{A}_{ii}\boldsymbol{\Lambda}^{i}_{bB} && (2.62)\\
\boldsymbol{A}_{BB} &= \sum_{k=1}^{N}\boldsymbol{A}^{k}_{BB} && (2.63)\\
\vec{b}_{B} &= \sum_{k=1}^{N}(\boldsymbol{\Lambda}^{k}_{bB})^{T}\vec{b}^{k}_{b} && (2.64)
\end{aligned}$$

The Eq (2.60) becomes

$$\begin{bmatrix} \boldsymbol{A}_{11} & \dots & \boldsymbol{0} & \boldsymbol{A}_{1B} \\ \vdots & \ddots & \vdots & \vdots \\ \boldsymbol{0} & \dots & \boldsymbol{A}_{NN} & \boldsymbol{A}_{NB} \\ \boldsymbol{A}_{B1} & \dots & \boldsymbol{A}_{BN} & \boldsymbol{A}_{BB} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \\ \alpha_B \end{bmatrix} = \begin{bmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_N \\ \vec{b}_B \end{bmatrix}. \tag{2.65}$$

Equation (2.65) is used to calculate the sub-partition $\mathcal{P}_i$ internal DOFs solution $(\alpha_i)$ using following equation

$$\alpha_i = \boldsymbol{A}^{-1}_{ii}(\vec{b}_i - \boldsymbol{A}_{iB}\alpha_B) \tag{2.66}$$

To avoid the dependence of $\alpha_B$ on the internal DOFs $\alpha_i$, we use (2.66) and arrive the following

$$\boldsymbol{A}_{B1}\alpha_1 + \boldsymbol{A}_{B2}\alpha_2 + \ldots + \boldsymbol{A}_{BN}\alpha_N + \boldsymbol{A}_{BB}\alpha_B = \vec{b}_B \tag{2.67}$$

The Eq (2.67) is dependent on all partitions $\alpha_i$. By replacing all the $\alpha_i$ values from (2.66) where $i = 1...N$.

$$\begin{aligned}
\boldsymbol{A}_{B1}\boldsymbol{A}^{-1}_{11}(\vec{b}_1 - \boldsymbol{A}_{1B}\alpha_B) + \ldots + \boldsymbol{A}_{BN}\boldsymbol{A}^{-1}_{NN}(\vec{b}_N - \boldsymbol{A}_{NB}\alpha_B) + \\
\boldsymbol{A}_{BB}\alpha_B = \vec{b}_B
\end{aligned} \tag{2.68}$$

and then $\alpha_B$ satisfies:

$$\sum_{k=1}^{N}(\boldsymbol{A}_{BB}^{k} - \boldsymbol{A}_{Bk}\boldsymbol{A}_{kk}^{-1}\boldsymbol{A}_{kB})\alpha_B = \sum_{k=1}^{N}(\vec{b}_B^{k}) - \boldsymbol{A}_{Bk}\boldsymbol{A}_{kk}^{-1}\vec{b}_k) \tag{2.69}$$

By replacing the values of $\boldsymbol{A}_{kB}$, $\boldsymbol{A}_{BB}^{k}$ and $\vec{b}_k$ in (2.69) from (2.61), (2.62) and (2.64) respectively, we get the final form of the system $\alpha_B$

$$\begin{aligned}\sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T(\boldsymbol{A}_{bb}^{k} - \boldsymbol{A}_{kb}\boldsymbol{A}_{kk}^{-1}\boldsymbol{A}_{kb})\boldsymbol{\Lambda}_{bB}^{k})\alpha_B = \\ \sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T(\vec{b}_b^{k} - \boldsymbol{A_{ik}}\boldsymbol{A_{kk}^{-1}}b_k))\end{aligned} \tag{2.70}$$

Defining the *Schur Complement* of $\mathcal{P}_j's$ data as

$$\boldsymbol{S}_{bb}^{P_j} = \boldsymbol{A}_{bb}^{P_j} - \boldsymbol{A}_{bi}\boldsymbol{A}_{ii}^{-1}\boldsymbol{A}_{ib} \tag{2.71}$$

$$\vec{v}_b^{P_j} = \vec{b}_b^{P_j} - \boldsymbol{A}_{bi}\boldsymbol{A}_{ii}^{-1}\vec{b}_i \tag{2.72}$$

The (2.70) can be written as

$$\sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T\boldsymbol{S}_{bb}^{k}\boldsymbol{\Lambda}_{bB}^{k})\alpha_B = \sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T\vec{v}_b^{k}) \tag{2.73}$$

By declaring

$$\boldsymbol{S} = \sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T\boldsymbol{S}_{bb}^{k}\boldsymbol{\Lambda}_{bB}^{k}) \tag{2.74}$$

$$\vec{v} = \sum_{k=1}^{N}((\boldsymbol{\Lambda}_{bB}^{k})^T\vec{v}_b^{k}) \tag{2.75}$$

The (2.73) can be written in the compact form

$$\alpha_B = \boldsymbol{S}^{-1}\vec{v}. \tag{2.76}$$

The static condensation method computes the solution of by dividing the system into smaller systems. It is used to solve the system of equation which is not possible to solver on shared memory machine using direct method. This method allows to solve it on distributed memory architectures. This method will be referred to in the distributed direct solver chapter.

### 2.2.4 Parallel Iterative Solver

The residual can be calculated on distributed architectures by computing residual on each partition.

$$\begin{aligned} r &= \vec{b} - \boldsymbol{A}\alpha_g, \\ &= \sum_{K \in \mathcal{P}} (\boldsymbol{\Lambda}_K^T r_K). \end{aligned} \tag{2.77}$$

For a mesh compose of $N$ partitions. The $r$ will be constructed by gathering all the partitions residual vector $r_P$. (2.77) can be written in term of partitions $r_P$ as:

$$r = \sum_{\mathcal{P}_i \in \mathcal{P}} (\boldsymbol{\Lambda}_{P_i}^T r_{P_i}). \tag{2.78}$$

For a partition $\mathcal{P}_i$, its residual vector $r_{P_i}$ can be written as the sum of its elements residual vector $r_K$ using theorm 2.2.3. The relationship is described as.

$$r_{P_i} = \boldsymbol{\Lambda}_{P_i} \sum_{K \in \mathcal{P}_i} (\boldsymbol{\Lambda}_K^T r_K), \tag{2.79}$$

by replacing the $r_{P_i}$ in (2.78)

$$r = \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{\Lambda}_{P_i}^T \boldsymbol{\Lambda}_{P_i} \sum_{K \in \mathcal{P}_i} (\boldsymbol{\Lambda}_K^T r_K). \tag{2.80}$$

In this chapter the basic idea of calculating the residual for partitioned domain is discussed. The method will be discussed in detail in forthcoming Distributed

Iterative Solver chapter.

## 2.3 Conclusion

This chapter has given an introduction of the finite element methods. It has discussed, how to create system of equations from the elements data and multiple methods of computing solution of the system of equations. These methods and algorithms will be referred in forthcoming chapters where their implementations will be discussed.

# Chapter 3

# FEDomain Interface

The FEDomain package interface is designed for beginner and expert finite element software developers. It is aimed for software developers or researchers who have no or little experience of parallel programming. They should be able to convert their existing C++ finite element sequential application code into a parallel application by using this package with minimum modifications. For advanced parallel software developers this interface provides flexibility in terms of software design and implementations. The main objective of the FEDomain interface is that it should be easy to understand and involves minimum user involvement in different finite element solution stages specially assembling of data and computing solution.

The interface of the software package defines the methods of software package interaction with the user. It informs the user about the type of data and its format required by the package. FEDomain package requires finite elements data ($b_K$, $\boldsymbol{A}_K$ and $\boldsymbol{\Lambda}_K$) from all $K$ in $\mathcal{P}$. It treats all elements in the mesh as a C++ object which provide its data through the specified FEDomain interface. The advantage of constructing elements as objects is these encapsulate all the element related information as a single object. It protect the element's private information from outer world. The classes make application code more maintainable and reusable.

The standard interface allows the FEDomain to access data from all the elements irrespective of their internal implementation. This allows developers to reuse their classes with other problems having their elements.

## 3.1 FEDomain Interface Version 1

The FEDoman was initially aimed to efficiently assemble the elements data and calculate residual for iterative solvers. FEDomain will require access to all mesh elements to assemble their system data. The user is responsible for the selecting the problem type, problem data, domain and element geometry. The implements of the element classes are dependent on these parameters. The element class implementation of the Poisson's equations, Elasticity equations and Convection-Diffusion equations will be different. Even in single problem the algorithm can vary as in case of Convection-Diffusion equation which smoothing function is used such as SUPG [24] and SOLD [49]. It depends on the user choice such as for tetrahedron element which quadrature rule is used to approximate load vector etc. From above reasons it is decided to the implementation of the mesh element classes is user domain.

The FEDomain package requires stiffness matrix $\boldsymbol{A}_K$ and load vector $\vec{b}_K$ from the element objects. The FEDomain will require a standard interface in element class to retrieve data. The user should implement the three methods given in Listing 3.1 as an abstract base class. The abstract base class has three interface methods getLoad, getStiffness and getConnectivity as pure virtual functions. The abstract class also define the containers used by interface methods to return $\vec{b}_K$, $\boldsymbol{A}_K$ and $\boldsymbol{\Lambda}_K$.

```
class Element{
public:
      typedef typename dense_vector Vec;
      typedef typename dense_matrix Mtx;
```

```
        typedef typename sparse_matrix SMtx;
        Vec& getLoad() = 0;
        Mtx& getStiffness() = 0;
        SMtx& getConnectivity() = 0;
};
```

Listing 3.1: Element Interface

The FEDomain package requires the list of elements, total number of DOFs and the information about the containers in which the data will return. FEDomain package is implemented as a template class. The first template parameter is the user-provided abstract element class as shown in Listing 3.2. The constructor of the FEDomain package requires a list of element pointers and total number of DOFs.

```
template<typename TEle,typename TVec>
class FEDomain{
public:
    FEDomain(vector<TEle*>* ele_ptrs, size_t total_dofs);
    TVec getResidual(TVec&);
};
```

Listing 3.2: FEDomin Interface Version 1.0

The FEDomain interface and abstract base element class allow FEDomain packages to be used by any set of element classes. There are a few shortcomings of the element interface given in Listing 3.1. First, the elements are required to internally store their data and provide their containers to the FEDomian on request. There are many cases that finite element class developers do not require to store finite elements stiffness and load data inside their objects. Second, all the element classes have to use the same data structures. There can be a possibility that a user can use different data structures in element classes for performance and efficiency.

The users are allowed to choose their data structures but these data structures should support specific signature for get and set functions. The Vec is one dimensional data structure which should support [ ] subscript operator interface. The Mtx should represent the two dimensional data structure to store $\boldsymbol{A}_K$. The most commonly access operator are two consecutive subscript operator [ ][ ]. [ ][ ] are taken as two separate subscript operators in C++, the first subscript operator is used to access the substructure (a row or column depending on the implementation) inside the matrix and the second subscript operator points the data entry in that sub structure. To avoid this problem, round brackets operators ( , ) with two arguments are used to get data from 2 dimensional containers.

In object oriented programming, the data container classes provide accessors methods for their private data. Every data container is designed for specific usage like different storage algorithms will be used to store dense and sparse matrices. If the provided data containers are used in internal calculation then lack of understanding of their internal storage scheme results in an inefficient calculation. On other hand, if elements data is copied in FEDomian solver, internal data container cause wastage of time as well as memory.

## 3.2 FEDomain Interface Version 2

In FEDomain interface version 2, the FEDomain package has provided the data structures to represents $\boldsymbol{A}_K$, $\vec{b}_K$ and $\boldsymbol{\Lambda}_K$ and avoided dependency on third party containers. The FEDomain package will use these data structures to get elements data. The FEDomain package data containers allow to manipulate and store data efficiently. To standardise the element interface the *FEElement* abstract class in Listing 3.3 is added in the FEDomain package. The users have to inherit their element classes from the FEElement class and implement the data access methods in their element classes.

```
class FEElement{
public:
    size_t getDofsCount();
    void getLoad(FEVector&);
    void getStiffness(FEMatrix&);
    void getConnectivity(FESparseMatrix&);
};
```

Listing 3.3: Element Interface

The *FEVector*, *FEMatrix* and *FESparseMatrix* are data containers implemented in the FEDomain package. The FEVector and FEMatrix are a dense vector and a dense matrix, implemented to obtain element's $b_K$ and $\boldsymbol{A}_K$ respectively. The FESparseMatrix is a sparse matrix and used to retrieve the mapping information $\boldsymbol{\Lambda}_K$. The getDofsCount() function is added to obtain the number of degrees of freedom on the element. The FEElement interface does not force element objects to store internal data. It provides the user a freedom for internal implementation by allowing the elements to either store their data into various data structures or do not store any data at all.

The FEDomain interface version 2 is given in the Listing 3.4. The users are provided with the abstract FEElement class which has reduced the FEDomain template count to only one template parameter TVec. It is possible that users are using a third party vector implementation in their algorithm and not wanted to use FEDomain provided container. The FEDomain interface has two functions to get residual *getResidual_FA* and *getResidual_EBE*. These methods implement the *full assembly* and *element by element* residual methods which are explained in section 2.2.2.1. Both of these residual methods require different type of data assembly for calculations. The FA requires $\boldsymbol{A}$ and $\vec{b}$ and EBE requires each elements $\boldsymbol{A}_K$, $\vec{b}_K$ and $\boldsymbol{\Lambda}_K$ as explained in (2.37) and (2.38).

```
template <typename TVec>
```

```
class FEDomain{
public:
    FEDomain(vector<FEElement*>& ele_ptrs, size_t total_dofs);
    TVec getResidual_FA(const TVec&);
    TVec getResidual_EBE(const TVec&);
};
```

Listing 3.4: FEDomin Interface Version 2.0

The FEVector in Listing 3.5 is an one dimensional dense data structure to get load vector from the elements. The subscript operator [ ] is implemented in a FEVector to get and set data. The FEMatrix is a two dimensional dense data structure to get stiffness matrix from the elements. The FESparseMatrix is a two dimensional sparse data structure to store scarcely populated matrix data. The dense and sparse matrices classes have similar interfaces defined in Listing 3.6. The getValue and setValue functions allow to hide the matrix data storage algorithms and provide generic interface to the containers data.

```
class FEVector{
public:
FEVector(size_t i);
size_t size() const;
double& operator[](size_t& i);
double operator[](size_t& i) const;
};
```

Listing 3.5: corrVector Interface

```
class FEMatrix{
public:
FEMatrix(size_t i, size_t j);
size_t rows() const;
size_t cols() const;
void setValue(size_t& i, size_t& j, double& v);
double getValue(size_t& i, size_t& j);
```

```
};
```

Listing 3.6: Matrix Interface

The FESparseMatrix is used to store the element connectivity matrix or system stiffness matrix. The size of connectivity matrix is $N_K$ by $N$. The connectivity data cannot be displayed as a vector because it is possible to have more than one entry in a row as in case of hanging nodes. Suppose we have a triangle element with three DOFs present in a system of 100 DOFs. The $\mathbf{\Lambda}_K$ will have only three non zero entries. The FESparseMatrix object performs efficient data manipulation than dense matrix of same dimensions.

The residual function can be called from multiple instances in the user code. To switch between the residual methods every function call has to be altered. This can be tiresome and error prone process. The residual functions signature return a vector. Each call to the residual function involves a memory coping of $N$ data values.

## 3.3 FEDomain Interface Version 3

Listing 3.7 contains FEDomain interface version 3. The new parameter *residual_method* is added in the FEDomain constructor to set the residual method. The user has to specify the residual method at FEDomain construction time which will not modify during the object lifetime. The residual_method parameter in the FEDomain constructor allows to have a single getResidual method in the FEDomain package.

```
template <typename TVec>
class FEDomain{
public:
   FEDomain(vector<FEElement*>& elements,
            size_t total_system_dofs,
```

```
            map<size_t , double>& dirichlet_constraints ,
            size_t max_dofs_in_elements ,
            res_type& residual_method );
    void getResidual(const TVec& apprx_sol_vec , TVec& res_vec );
};
```

Listing 3.7: FEDomin Interface Version 3.0

The *max_dofs_in_elements* is a tuning parameter in the FEDomain constructor. This parameter represents the maximum number of DOFs an element can have in $\mathcal{P}$ and is used in efficient data structure implementation of the FEDomain package. The *dirichlet_constraints* represents the Dirichlet DOFs ids and values. The user has an option to either apply Dirichlet constraints at the elements data before providing it to the FEDomain or provide Dirichlet data to FEDomain at construction stage. FEDomain will apply Dirichlet constraints during residual computation. The std::map is selected for dirichlet_constraints because of its constant search time.

The residual function signature is also modified. The first parameter is a constant vector and represents an approximate solution. The second argument is a residual vector which will be populated by the residual method. Both the containers should have same size. The residual method new signature is adopted to avoid memory copying occurred in due to previous interface. The residual calculation method is selected at the construction stage and is automatically implemented by residual method for each residual call. Both the residual methods require to handle element data differently. By modifying the residual method a much optimized data management can be done.

## 3.4 FEDomain Interface Version 4

```
template <typename TVec>
```

```
class FEDomain{
public:
   FEDomain(vector<FEElement*>& elements,
            FE_UINT total_system_dofs,
            map<FE_UINT, FE_DATA>& dirichlet_constraints,
            FE_UINT max_dofs_in_elements,
            res_type& residual_method,
            vector<FE_UINT>& partition_id);
   void getResidual(const TVec& apprx_sol_vec, TVec& res_vec);
};
```

Listing 3.8: FEDomin Interface Version 4.0

The FEDomain interface version 4.0 in Listing 3.8 is implemented for a distributed memory parallel iterative solver. The MPI library is used to implement parallel processes and communication among the processes. For parallel iterative solver the elements are distributed among MPI parallel precesses so that each element is processed in only one of the MPI processes. The new parameter *partition_id* is added in FEDomain constructor to represent the mesh partitions allocated to MPI processes. The user can assign one or more partitions to each MPI process. The partition_id parameter is set as a vector type such that it can represent all the allocated partitions ids. For the user's convenience, the user can either provide all the elements in $\mathcal{P}$ to each MPI process or provide specified partitions elements. The value of partition_id should be zero for a non-partitioned mesh. All the mesh partitions should be represented by identifiers of type size_t greater then zero.

```
class FEElement{
public:
    FE_UINT getDofsCount();
    FE_UINT getPartitionID();
    void getLoad(FEVector&);
    void getStiffness(FEMatrix&);
    void getConnectivity(FESparseMatrix&);
```

```
};
```

Listing 3.9: Element Interface 2.0

In FEDomain constructor all the elements are checked to filter group of elements present in targeted partitions. The FEElement class interface in Listing 3.3 does not have any member functions to provide partition id of element. The FEElement class interface is modified by adding a new member function *getPartitionID* to get partition identifier of all mesh elements as in Listing 3.11.

The getResidual function will be called by the user in all the MPI processes but the solution vector should be provided only to MPI process having process_id 0 (MPI_0). The MPI_0 getResidual function will return the residual vector. This design decision made for the FEDomain user having no experience in parallel coding. Such user can run his sequential code as parallel with minimum additional steps. The FEDomain interface allows the user (new to parallel coding) to execute his sequential or shared memory code to run on distributed memory architectures. The user will be required to add FEDomain library and some other MPI related modifications in his current code.

## 3.5 FEDomain Interface Version 5

The FEDomain was initially developed for efficiently calculating the residual for iterative solvers through full assembly and element by element methods. The support to direct solvers is later added into FEDomain package. A new parameter $SOLUTION\_METHOD$ is added to the FEDomain constructor to represent the solution type of FEDomain object. The direct solver will provide the solution vector of the problem. The *getSolution* method is added to the FEDomain interface to obtain a solution vector. The new interface is in Listing 3.10.

```
template <typename TVec>
```

```
class FEDomain{
public:
   FEDomain(vector<FEElement*>& elements,
            map<FE_UINT, FE_DATA>& dirichlet_constraints,
            FE_UINT total_system_dofs,
            FE_UINT max_dofs_in_elements,
            SOLUTION_METHOD& method,
            vector<FE_UINT>& partition_id);
   void getResidual(const TVec& apprx_sol_vec, TVec& res_vec);
   void getSolution(TVec& sol_vec);
};
```

Listing 3.10: FEDomin Interface Version 5.0

The *SOLUTION_METHOD* is FEDomain's internally define enumeration type to define the solution methods. Currently it supports three methods, two for residual (*RESIDUAL_FA* and *RESIDUAL_EBE*) and one for direct solver (*DIRECT_SOLVER*). The user has to select the solution method at the FEDomain object construction stage.

## 3.6 FEDomain Interface Version 6

The FEDomain interface version 5 lacked the provision of solution allocation to the elements. The user had to provide the solution to the elements for post processing. The FEElement class is modified to support the allocation of solution to the elements and a new interface method setSolution(FEVector&) is introduced as shown in Listing 3.11.

```
class FEElement{
public:
    FE_UINT getDofsCount();
    FE_UINT getPartitionID();
    void getLoad(FEVector&);
```

```
    void getStiffness(FEMatrix&);
    void getConnectivity(FESparseMatrix&);
    void setSolution(FEVector&);
};
```

Listing 3.11: Element Interface 3.0

The getSolution(TVec&) function is removed from the FEDomain class interface and the setSolution(const map&) is added as shown in Listing 3.12. The setSolution(const map&) takes Dirichlet data in terms of map as a parameter. In case of direct solver the setSolution(const map&) function will calculate the solution of the system and provide each element with its DOFs solution. In case of residual mode, the user will provide the entire solution to setSolution(const map&) to provide solutions to mesh elements.

```
template <typename TVec>
class FEDomain{
public:
    FEDomain(vector<FEElement*>& elements,
             map<FE_UINT, FE_DATA>& dirichlet_constraints,
             FE_UINT total_system_dofs,
             FE_UINT max_dofs_in_elements,
             SOLUTION_METHOD& method,
             vector<FE_UINT>& partition_id);
    void getResidual(const TVec& apprx_sol_vec, TVec& res_vec);
    void setSolution(const map<FE_UINT, FE_DATA>& dirichlet_data);
};
```

Listing 3.12: FEDomin Interface Version 6.0

## 3.7 FEDomain Interface Version 7

The FEDomain interface is modified to add support for static condensation, defined in section 2.2.3, into the package for the shared memory systems. A new solu-

tion type $STATIC\_CONDENSATION$ is added into $SOLUTION\_METHOD$ enumerated values. The user can switch between different solvers only by changing one parameter in the constructor. The finite element method can lead to symmetric or unsymmetric matrices, which can further be classified as definite or indefinite matrices. This information is required by the linear algebra solver. The user specifies the type of stiffness matrix which will be produced. The new parameter $MATRIX\_TYPE$ is added into the FEDomain constructor and its values are given in Listing 3.13. The user has to specify one of these types at the construction stage.

```
MATRIX_TYPES = { DEFINITE_SYMMETRIC,
                 INDEFINITE_SYMMETRIC,
                 DEFINITE_UNSYMMETRIC,
                 INDEFINITE_UNSYMMETRIC }
```

Listing 3.13: FEDomin Matrix Types

```
template <typename TVec>
class FEDomain{
public:
    FEDomain(vector<FEElement*>& elements,
             map<FE_UINT, FE_DATA>& dirichlet_constraints,
             vector<FE_UINT>& partition_id
             FE_UINT total_system_dofs,
             FE_UINT max_dofs_in_elements,
             MATRIX_TYPE& matrix,
             SOLUTION_METHOD& method);
    void getResidual(const TVec& apprx_sol_vec, TVec& res_vec);
    void setSolution(const map<FE_UINT, FE_DATA>& dirichlet_data);
};
```

Listing 3.14: FEDomin Interface Version 7.0

The FEEquation class given in Listing 3.15 is added into the FEDomain package. The FEEquation encapsulates the implementation details of the static condensa-

tion. In FEEquation constructor has four parameters where the first one represents the total DOFs of the system. The second and third parameters represent the list of the allocated partitions internal DOFs, global ids and interface DOFs, respectively. The FEEquation has two operational modes. The fourth parameter in constructor selects these modes. If it is set to true the FEEquation class set as a Static Condition and compute $\boldsymbol{S}_{jj}$ and $\vec{v}_j$ using (2.71) and (2.72), respectively, and also calculates $\alpha_i$ by implementing (2.66).

The FEDirectSolver and FEDirectSolverMPI classes are composed of the FEEquation object. The FEEquation has to collect elements stiffness matrix as well as load vector. The two getValue functions are added into FEEquaiton class. The $addValue(size\_t, double)$ is used to get the value of the load vector and $addValue(size\_t, size\_t, double)$ is to update the stiffness matrix. The name of the functions is selected keep the same data structure interface. Previously, the elements were required to provide data using their local DOF numbering. This is not possible with FEEquation so the user has to provide elements data using their global DOFs numbering $\mathcal{N}$. The FEEquation user interface is in the Listing 3.15.

```
class FEEquation{
public:
    FEEquation(FE_UINT& total_sys_dofs,
               vector<FE_UINT>& internal_dofs,
               vector<FE_UINT>& interface_dofs,
               MATRIX_TYPE m_type, bool condensation);
    void addValue(FE_UINT& index, FE_DATA& vlaue);
    void addValue(FE_UINT& row_id, FE_UINT& col_id, FE_DATA& calue);
    void getSystem(FEEquation& EQ);
    template<typename V1, typename V2>
    void solve(V1& interface_value, V2& solution);
};
```

Listing 3.15: FEEquation Interface

The FEElement abstract class does not support FEEquation objects. The new $getSystem(FEEquation\&)$ function is introduced in FEElement class to support static condensation. The new function cannot replace already present getStiffness and getLoad functions which were implemented for the residual calculation. The user has to add data in $getSystem(FEEquation\&)$ using global DOF numbering $\mathcal{N}$ through the interface defined in Listing 3.15. For $RESIDUAL\_FA$ or $RESIDUAL\_EBE$ only the first set of get functions are required and for $DIRECT\_SOLVER$ and $STATIC\_CONDENSATION$ the new get function has to be implemented, but the user can implement all the get functions.

```
class FEElement{
public:
    FE_UINT getDofsCount();
    FE_UINT getPartitionID();
    void getLoad(FEVector&);
    void getStiffness(FEMatrix&);
    void getSystem(FEEquation&);
    void getConnectivity(FESparseMatrix&);
    void setSolution(FEVector&);
};
```

Listing 3.16: Element Interface 4.0

## 3.8 FEDomainMPI Interface

The FEDomainMPI interface is implemented in FEDomain package to represent distributed memory architecture implementations. It is MPI version of FEDomain interface. In each MPI process the FEDomainMPI object is constructed. The MPI process with identifier 0 is called master process (MPI_0) and the FEDomainMPI object constructed in it is master object. All the other MPI processes are client processes and their FEDomainMPI objects are called client objects. For distributed solvers, no mesh partition is allocation to the master object. It will be responsi-

ble to calculate the solution of the interface DOFs. The role of master object is discusses in Distributed Direct Solver chapter. The mesh partitions are allocated to the client objects. The master object requires all the Dirichlet DOFs data at construction time. The user should atleast provide the global DOF ids and value of the Dirichlet DOFs present in allocated partitions to the client object.

```
template <typename TVec>
class FEDomainMPI{
public:
    FEDomainMPI(vector<FEElement*>& elements,
                map<FE_UINT,FE_DATA>& dirichlet_constraints,
                vector<FE_UINT>& partition_id,
                FE_UINT total_system_dofs,
                FE_UINT max_dofs_in_elements,
                MATRIX_TYPE matrix,
                DISTRIBUTED_SOLUTION_METHOD method);
    void getResidual(const TVec& apprx_sol_vec, TVec& res_vec);
    void setSolution(const map<FE_UINT,FE_DATA>& dirichlet_data);
};
```

Listing 3.17: FEDomainMPI Interface

The interface of the FEDomainMPI is kept similar to the FEDomain class. The DISTRIBUTED_SOLUTION_METHOD is introduced in place of SOLUTION_METHOD to select distributed solvers available in the FEDomainMPI. Currently there are two distributed solver, DIS_DIRECT_SOLVER and DIS_HYBRID_SOLVER. The FEDomainMPI objects have to be provided unique list of partition ids. During the design stage of the FEDomainMPI class, it was considered that the FEDomainMPI objects should distribute mesh partitions among them. This decision creates uncertainty of which partition will be mapped to which FEDomainMPI object and the user will have to provide all the elements to all the FEDomainMPI objects. This design will result in huge waste of memory in case of large mesh. The drawback can be avoided by allowing the user to perform partition to FEDo-

mainMPI object mapping. The user will allocate partitions to the FEDomainMPI objects and will provide the elements belonging to these partitions. This design allows the user to only allocate the minimum memory for each MPI process.

## 3.9 Summary and Conclusion

The FEDomain package is implemented for the shared memory and distributed memory architectures. There are two interfaces developed, FEDomin for shared memory and FEDomainMPI for distributed memory architectures. The final version of the FEDomain interface is given in Listing 3.12. The interface required following data from the user.

- The reference to the vector containing all the mesh elements pointer. All the mesh elements classes should be inherited from the FEElement class. The element objects will be accessed in FEDomain package through polymorphism.

- The list of partition ids are required for the static condensation solver. FEDomain package will consider the elements for computation which will belong to the partitions identified by the user. The parameter requires a vector of partition ids. If the size of the vector is zero, all the provided elements are considered as a single partition and solved by factorization method.

- Total number of DOFs are present in targeted system. This variable is dependent on the problem type and the number of nodal points in a mesh.

- Maximum number of DOFs a single element can have. It depends on the type of problem and the shape of elements.

- Matrix type. There are four matrix type supported by FEDomain. Its value can be set any of these four value

    1. DEFINITE_SYMMETRIC

2. INDEFINITE_SYMMETRIC
3. DEFINITE_UNSYMMETRIC
4. INDEFINITE_UNSYMMETRIC

- Solution type. There are six solution methods available in FEDomain package. These methods are selected by setting the Solution type to following values

  1. DIRECT_SOLVER (Compute solution of the system using LU factorization).
  2. CONDENSATION_SOLVER (Compute solution of the system using static condensation method).
  3. RESIDUAL_FA (Compute residual vector using full assembly method).
  4. RESIDUAL_EBE (Compute residual vector using element by element method using OpenMP).
  5. RESIDUAL_TBB (Compute residual vector using element by element method using Intel TBB library).
  6. NON_LINEAR_SOLVER (Compute solution of the non linear system).

The FEDomainMPI class in FEDomain package is the interface for all the distributed memory solvers. The final interface of FEDomainMPI is given in Listing 3.17. The FEDomainMPI object will be created in each MPI process. The description of each parameter in the FEDomainMPI class constructor is given below.

- All the FEDomainMPI objects are allocated one or more partitions. The user has to provide the elements belonging to allocated partitions for MPI process. The list of elements as a vector are provided to each object. Each element should be inherited from FEElement class and will be accessed through polymorphism.

- The Dirichlet constraint data is provided to all the FEDomainMPI object. The data contains the DOF global id and value. Each FEDomainMPI processes should be provided its allocated partitions Dirichlet data. In case of static condensation solver the FEDomainMPI object at MPI process 0 has to be provided all the Dirichlet constraints information.
- The vector of partition ids maps which of the mesh partition will be processed on which MPI process. More than one partitions can be allocated to each FEDomainMPI object. In case of static condensation solver no partition is allocated to the FEDomainMPI object at process 0.
- Maximum number of DOFs in the mesh
- Maximum number of DOFs in the elements. It can vary on each MPI process.
- MATRIX_TYPE is same as of in FEDomain interface.
- DISTRIBUTED_SOLUTION_METHOD is used to select the solution method for the distributed memory architecture. The solution methods are given below.

    1. DIS_DIRECT_SOLVER (Compute solution of the system using static condensation. The solution of the interface DOFs is calculated using LU factorization).
    2. DIS_HYBRID_SOLVER (Compute solution of the system using static condensation. The solution of the interface DOFs is calculated using Conjugate Gradient method).
    3. DIS_RESIDUAL_FA (Compute residual vector using full assembly method).
    4. DIS_RESIDUAL_EBE (Compute residual vector using element by element method).
    5. DIS_RESIDUAL_FA_COMP (Compute residual vector using element by element method).

# Chapter 4

# FEDomain Shared Memory FE Solver

Software developers used to depend on the processor upgrade, to increase the performance of their software. The single core conventional processors used to gain their clock speed by increasing the number of transistors. The increase in clock speed is challenged from unavoidable thermal constraints. To use these processors time efficiently, different parallelism techniques like instruction level parallelism and hyper threading, have been developed. Unfortunately, these techniques could not provide significant decrease in execution time. The processors vendors overcome the performance bottleneck by packaging multiple computational cores in single processor package. The multi-core processors can execute multi-threads in parallel on different processing cores with shared memory.

The FEDomain package is implemented for the shared memory and distributed memory architectures. The FEDomain interface class is implemented in FEDomain package to represent shared memory architecture solver. The FEDomain class is implemented as a Facade design pattern which provides unified interface to a set of interfaces in a subsystems. In this chapter two subsystems of shared memory direct linear algebra solvers are discussed. The first solver uses LU fac-

torization to compute while second solver implemented domain decomposition.

## 4.1 Direct Solver

The shared memory direct solver is selected by setting the SOLUTION_METHDO parameter equal to DIRECT_SOLVER in the FEDomain class constructor (in Listing 3.12). In the FEDomain package the direct solver is implemented as FEDirectSolver class. During development cycle the FEDirectSolver class has evolved through different implementations. In this section these implementation stages will be discussed. The FEDirectSolver requires elements data, the matrix type of the system of equations, Dirichlet DOFs data and maximum possible size of the system in $\mathcal{P}$. The first interface of the FEDirectSolver class is given in Listing 4.1.

```
template <typename TVec>
class FEDirectSolver{
public:
    FEDirectSolver(vector<FEElement*>& elements,
                   map<FE_UINT, FE_DATA>& dirichlet_constraints,
                   FE_UINT total_system_dofs,
                   FE_UINT max_dofs_in_elements,
                   MATRIX_TYPE& matrix);
    void getResidual(const TVec& apprx_sol_vec, TVec& res_vec) {};
    void setSolution(const map<FE_UINT, FE_DATA>& dirichlet_constraints);
};
```

Listing 4.1: FEDirectSolver version 1

The FEDirectSolver requires list of finite element objects as FEElement pointers, Dirichle data and Matrix type, etc. The C++ polymorphism is used to extract $\boldsymbol{A}_K, b_K$ and $\boldsymbol{\Lambda}_K$ for each $K$. The first version of the FEDirectSolver (FEDS_V1) obtains elements data using FEElement interface defined in Listing 3.11. The finite elements should have knowledge about the total number of DOFs in system,

how many elements are in it, and what are the global ids for its DOFs. The FEDirectSolver class has following responsibility.

- Gather elements data and assemble $\boldsymbol{A}$ and $\vec{b}$.
- Transform the data into solver format.
- Initialize a third party solver.
- Compute solution.
- Provide solutions to the user provided elements.

In finite element methods the constraints are applied on the boundaries to specify the solution of the problem. The FEDomain package applies the Dirichlet constraints for the user. The Dirichlet boundary DOFs solutions are given in the problem definition as either constant values or functions. The Dirichlet constraints require modification of the data in $\boldsymbol{A}$ and $\vec{b}$. These have to be applied after assembling all the system of equations. The user can provide elements data after applying the Dirichlet constraints. The FEDirectSolver class constructor requires Dirichlet data from the user where Dirichlet data contains ids and values of the DOFs present on Dirichelet boundaries. For the FEDomain package efficient, it is preferred that the user provides elements data after applying Dirichlet boundary constraints.

In the FEDirectSolver class the constraints have to applied to $\boldsymbol{A}$ and $\vec{b}$ before computing the solution. There two method of applying Dirichlet constraints.

- In the first method, the Dirichlet constraints are applied after assembling $\boldsymbol{A}$ and $\vec{b}$. The application of Dirichlet constraints involves removal of data from $\boldsymbol{A}$ and modification of data in $\vec{b}$. After applying the constraints, the rows and columns of $\boldsymbol{A}$ represent the Dirichet DOFs have only single entry at diagonal position set to 1. The data in these columns are used to

modify $\vec{b}$. In this method the $\boldsymbol{A}$ has to initially accommodate the Dirichlet DOFs rows and columns data. These rows and columns have to delete after apply constraints and before computing solutions. The amount of data and cost of managing data is dependent on the number of DOFs lying on Dirichlet boundaries. In matrix data structures the data removal can be performed by two methods. In first method the data can be removed by setting its value to zero. In the matrix container their memory remain allocated. In second method, the matrix has the ability to dynamically expand and reduce memory footprint. The removal of data is performed by releasing the memory allocation. For efficient sparse matrix data structures it is either very expensive or not possible to remove data.

```
   // Applying Dirichlet constraints
 1 for i ← 1 to N do
 2     if IsNotDirichlet(i) then
 3         for j ← 1 to N_D do
 4             f[i] = f[i] − d_val[j] ∗ A[i][d[j]] ;
 5         end
 6     end
 7 end
   // Clearing Dirichlet rows and columns in A
 8 for j ← 1 to N_D do
 9     for i ← 1 to N do
10         A(i, d[j]) = A(d[j], i) = 0;
11     end
       // Setting A and b Dirichlet values
12     A(d[j], d[j]) = 1;
13     f[d[j]] = d_val[j];
14 end
```

**Algorithm 1:** Dirichlet constraints algorithm after full assembly.

- In the second method the Dirichlet constraints are applied on the element data before assembling $\boldsymbol{A}$ and $\vec{b}$. This method allows only the permanent data to be inserted in $\boldsymbol{A}$ and avoid removal of data. The data can be collected and constraints can be applied in parallel on multiple elements data.

All the user implemented classes have to be publicly inherited from the FEElement class. The FEElement class defines element interface functions as pure virtual functions. The mesh $\mathcal{P}$ is provided to the FEDirectSolver object as a vector of pointers of all $K \in \mathcal{P}$. The FEDirctSolver constructor assembles $\boldsymbol{A}$ and $\vec{b}$ using sequential and parallel methods. For a shared memory architecture the assembly of data is performed in parallel and is implemented using OpenMP directives [72]. In data assembling phase the elements are distributed among parallel threads. The elements distribution is performed through an OpenMP provided scheduler method (STATIC, GUIDED and DYNAMIC), which is selected by the user at runtime. These threads gather allocated elements data and apply Dirichlet constraints in parallel before adding into $\boldsymbol{A}$ and $\vec{b}$. The addition of elements data into $\boldsymbol{A}$ and $\vec{b}$ is a critical task as all the threads have to share the same resources. If one thread is adding data into $\boldsymbol{A}$ the other thread has to wait of the resource.

The FEDomain package is not aimed to develop a new linear algebra solver. There are many third party linear algebra packages like PARDISO [77], MUMPS [11], and SuperLU [58], etc available which are optimized for the shared memory architecture. During initial development of FEDomain the PARDISO solver was chosen as it was available in development environment. All these third party solvers require data ($\boldsymbol{A}$ and $\vec{b}$) into a specified data format and PARDISO requires data in Intel MKL compressed sparse row (CSR) format [7]. The CSR format store matrix entries in a set of three consecutive memory arrays as shown in Table 4.1. The first array contains the row pointers which points the starting index of every row in the next two arrays. The second array contains the column indexes

for each matrix data entries in sorted manner. The last array contains the data value for each entry in second array. The consecutive memory allocation allows efficient memory access required for high performance computation.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| rowIndex | = | ( | 1 | 4 | 6 | 9 | 12 | 14 | | | | | | ) |
| columns | = | ( | 1 | 2 | 4 | 1 | 2 | 3 | 4 | 5 | 1 | 3 | 4 | 2 | 5 | ) |
| values | = | ( | 1 | -1 | -3 | -2 | 5 | 4 | 6 | 4 | -4 | 2 | 7 | 8 | -5 | ) |

Table 4.1: Compressed Sparse Row Format

The insertion and deletion in CSR data containers in random order and location of matrix require unnecessary copying. As an example, if a CSR format matrix is populated with some data. A new value has to be added at the first column of its first row ($\boldsymbol{A}[0][0]$). There are two possibilities for inserting data into CSR. In first case the data is already present inside the container and its value is modified. In second case, a new memory location has to be created at the beginning of the second and third vectors. The memory for new location can be created either by coping all the data in second and third vectors to their next memory location if only these vectors have extra memory available. If there is no extra allocated memory available at the end of these vectors than larger consecutive memory has to be allocated for second and third vectors. The new value and data from old vectors have to be copied into new vectors. All the entries in the first vector have to be modified. The copying of data becomes more expensive as the number of non zero entries increases.

### 4.1.1 Sparse Matrix Container Requirements

The pattern of population of $\boldsymbol{A}$ and $\vec{b}$ cannot be predicted. All the elements have unique sets of DOFs where a DOF is most likely to be present in more than one element. The multiple elements with same DOFs will add values to the same row and column. These will update same location in $\boldsymbol{A}$ and $\vec{b}$. Suppose the DOF $i$ is not a Dirichlet DOFs and is present in 8 elements of $\mathcal{P}$. During data assembly

the diagonal position of the $i_{th}$ row in $\boldsymbol{A}$ will be updated 8 times and the same is true for the $i_{th}$ position in $\vec{b}$. The data structure selected to contain $\vec{b}$ is a dense vector composed of consecutive memory locations and accommodates all zero and non zero entries. $\boldsymbol{A}$ is a sparsely populated and to avoid memory wastage requires a data structure to store only non zero values. The required sparse matrix data structure should have characteristics like the data can be added and retrieved in any order, it should maintain data in sorted order, and the entry lookup in container should be efficient and ideally have a constant access time.

For $\boldsymbol{A}$, a data structure is needed which can perform insertions, erasures, and lookups in random order. The STL provides associative data containers like *std::map* and *std::set* and these containers have the hallmark of guaranteed logarithmic-time lookups. The standard associative containers are typically implemented as balanced binary search trees. A balanced binary search tree is a data structure that is optimized for a mixed combination of insertions, erasures, and lookups. These are designed for applications that do some insertions, then some lookups, then maybe some more insertions, then perhaps some erasures, then a few more lookups etc. The key characteristic of this sequence of events is that the insertions, erasures, and lookups are all mixed up. The *map* containers are sorted associative containers that provide fast retrieval of data associated to a unique key and keys are internally maintained in sorted order.

$$std :: map < row\_id, std :: map < column\_id, value >>$$

The *std::map* has all the properties required in a container to store $\boldsymbol{A}$. A new sparse matrix FESparseMatrixMM (SMMM) is implemented in FEDomain using map of maps. The FESparseMatrixMM enables us to populate $\boldsymbol{A}$ by adding data in random order and the data will always be maintained in sorted order. Like SMV the SMMM has drawbacks. On Linux systems the std::map used with

default allocator. std::map does not actually release allocated memory until the application stops. For efficiency, during application execution the std::map object does not released acquired memory to operating system. So that if any std::map object in application require memory, it does have to reallocate memory. An other drawback of SMMM is its memory consumption. As std::map is a balanced binary search tree made up of tree nodes. Each node holding not only a data, but also a pointer to the node's left child, a pointer to it's right child, and a pointer to its parent. The memory overhead of maintaining an associative container is at least three pointers for each node.

The SMMM memory consumption issue is very apparent as the number of non zero entries being stored in $\boldsymbol{A}$ increases. The new sparse matrix FESparseMatrixVV (SMVV) is implemented to remove the drawback of SMMM. The SMVV is implemented using vectors of pairs. Each row is further divided into smaller chunks of vectors. For data insertion, smaller vector is searched, if the data is not present, lesser amount of data copying is involved. This design also involves lesser comparisons for the lookup. The drawback of this design is that removal of data is not expensive. The timing and data comparisons of these structures are given below in table (4.2).

| Mesh (DOFs) | Cube4(7371) | | Cube5(53907) | | Cube6(411939) | | Cube7(3220035) | |
|---|---|---|---|---|---|---|---|---|
| Container | data size | time | data size | time | data size | time | data size | time |
| SMMM | 203069 | 0.282 | 1902189 | 3.004 | 16408799 | 24.490 | 136202251 | 247.762 |
| SMVV | 203069 | 0.153 | 1902189 | 1.246 | 16408799 | 10.084 | 136202251 | 86.178 |

Table 4.2: Data Structure Population Time and Number of Entries

### 4.1.2 Direct Solver Stages

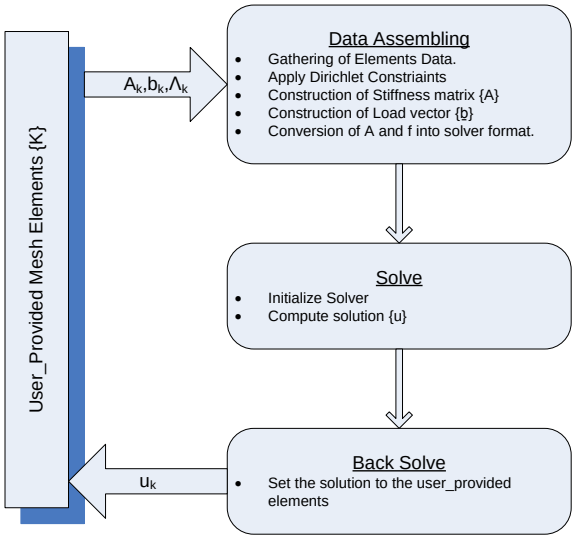



Figure 4.1: FEDirectSolver Model

The FEDomain direct solver can be divided into three stages as shown in Figure 4.1. The first stage objective is to obtain elements data and transform it into the solver acceptable format after applying Dirichlet constraints. The assembling of data is implemented in parallel using OpenMP extensions for shared memory processors. All the OpenMP threads have to share $\boldsymbol{A}$ and $\vec{b}$ and for the validity of data and to avoid race conditions only one of the threads can access $\boldsymbol{A}$ or $\vec{b}$ at any time. The distribution of elements among OpenMP threads depend upon the OpenMP environment variables *OMP_SCHEDULE* and *OMP_NUM_THREADS*. The user has to set these variable before executing the FEDomain application. Each OpenMP threads creates its own copy of a stiffness matrix $\boldsymbol{A}_t$ and and load vector $\vec{b}_t$ or dimensions $N \times N$ and $N$ respectively. Each thread assemble its allocated user elements into its $\boldsymbol{A}_t$ and $\vec{b}_t$. At the end the parallel region all the threads data $\boldsymbol{A}_t$ and $\vec{b}_t$ are added into $\boldsymbol{A}$ and $\vec{b}$. In final version the $\boldsymbol{A}$ is stored in SMVV container.

The FEDomain package does not have its own implementation of linear algebraic solver and it relies on third party PARDISO solver. The PARDISO solver requires $\vec{b}$ as a consecutive memory array of type double and $\boldsymbol{A}$ in CSR format which is again set of three consecutive memory arrays. During data assembly the user_provided elements insert their data $\boldsymbol{A}$ in random order. The $\boldsymbol{A}$ is initially assembled in SMVV container and after the data assembly has finished the $\boldsymbol{A}$ is converted into CSR format for the PARDISO solver. In FEDomain second stage, the solver is initialized and solution is computed. The user has to specify the type of stiffness matrix (SYMMETRIC and NONSYMMETRIC) at the constructor time of the FEDomain class. The PARDISO solver requires $\boldsymbol{A}'s$ all non zero entries for the non_symmetrix stiffness matrix and only upper triangle of $\boldsymbol{A}'s$ non zero entries for symmetric problem. The PARDISO provides minimum degree algorithm[59] and nested dissection algorithm[53] for fill-in reduction. The nested dissection is parallel algorithm implemented for shared memory systems[46]. PARDISO solver solution process is composed of three stages.

- In first stage analysis and symbolic factorization of the $\boldsymbol{A}$ is completed. This stage includes allocation of memory and fill in calculations.
- The second stage is composed of numerical factorization which depends on the matrix type (symmetric and unsymmteric).
- The last stage includes forward and backward solve through iterative refinements.

In the last stage, the computed solution is provided to the user provided finite element objects. This process is implemented in parallel. This will allow the user to use the elements for post processing.

### 4.1.3 Direct Sovler Timing

Table 4.3 shows the timing data for the direct solver. The timings are obtained by solving Elasticity 3D problem using different threads. The mesh used has 811392 elements and 411939 DOFs. Intel Xeon E5560 processor is used which has 4 cores and supports 8 threads. The second column in Table 4.3 contains the FEDomain data assembling timing for various threads. During data assembling stage, $\boldsymbol{A}$ and $\vec{b}$ are constructed by adding contribution from each element. The assembling time reduces with the increase in number of threads. The third, fourth and fifth columns have timing for the PARDISO solver three stages (Symbolic factorization, Numerical factorization and solve stage). It is observed only the numerical factorization varies with the number of threads used. In the back solve stage the FEDomain direct solver provides solution to the element objects. The sixth column shows timing for the back solve stage. The last column is the time consumed by all the stages. The data timing is shown in Figure 4.2.

| OpenMP | Data | Factorization | | | Back | Total |
|---|---|---|---|---|---|---|
| Threads | Assemble | Symbolic | Numerical | Solve | Solve | Time |
| 1 | 192.6 | 6.5 | 292.3 | 2.9 | 1.19 | 495.47 |
| 2 | 101.0 | 5.7 | 166.5 | 2.9 | 0.56 | 276.67 |
| 3 | 69.04 | 4.5 | 118.5 | 2.9 | 0.37 | 195.35 |
| 4 | 53.19 | 4.6 | 92.23 | 2.9 | 0.28 | 153.20 |
| 5 | 43.69 | 4.6 | 82.73 | 2.9 | 0.23 | 134.15 |
| 6 | 37.63 | 4.3 | 72.23 | 2.9 | 0.19 | 117.25 |
| 7 | 33.28 | 4.4 | 65.80 | 2.9 | 0.16 | 106.54 |
| 8 | 32.23 | 4.2 | 63.19 | 2.9 | 0.14 | 100.66 |

Table 4.3: The timing results are obtained using 3D Elasticity mesh. The mesh has 411939 DOFs and 811392 elements. This table timing data of all stages for FEDomain direct solver using 1 to 8 OpenMP threads.

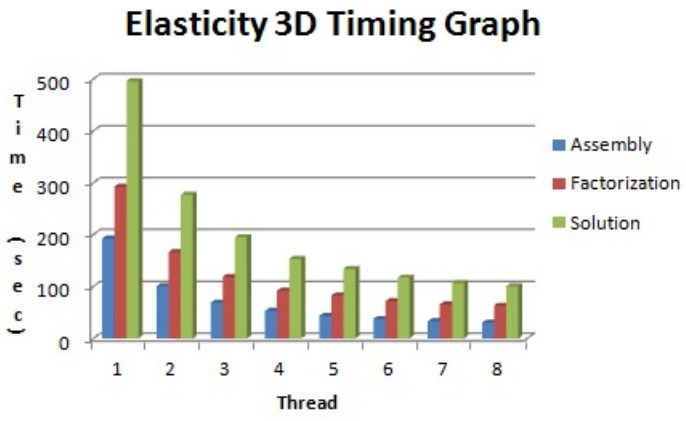



Figure 4.2: The graphs have the timing, speed-up and efficiency data for the FEDirectSolver obtained using Elasticity 3D problem with OpenMP threads. The FEDirectSolver class has two main stages assembling of data and factorization. The solution bar represents the total time used by the FEDirectSolver to calculate the solution. The mesh used contains 411939 DOFs and 811392 elements.

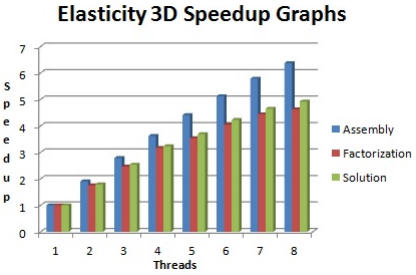
$$\text{Speed up} = T_1/T_n \tag{4.1}$$

$$\text{Efficiency} = \text{Speed up} * 100/n \tag{4.2}$$

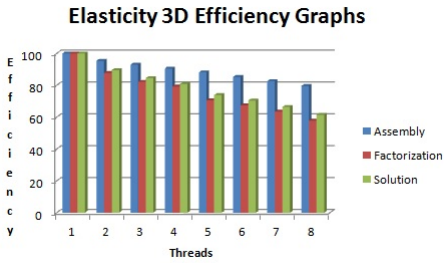
The (4.1) and (4.2) are the speed up and efficiency equations used to compute direct solver speed up and efficiency shown in Table 4.4. The Table 4.3 is as source for Table 4.4. The data assembler is implemented in FEDomain package has obtained better speed up and efficiency as compare to PARDISO solver. The FEDomain solver improves the over all solution time. Figure 4.3a and Figure 4.3b shows the speed up and efficiency graphs of the FEDomain solver. The FEDomain has achieved best speed up of 4.92 for 8 threads. FEDomain data assembling stage has achieved 6.37 speed up and 79.64 efficiency for 8 OpenMP threads. PARDISO solver has obtained 4.63 speed up and 57.82 percent efficiency.

| | Speedup | | | | Efficiency | | | |
|---|---|---|---|---|---|---|---|---|
| OpenMP Threads | Data Assemble | PARDISO Factorization | Back Solve | Total Time | Data Assemble | PARDISO Factorization | Back Solve | Total Time |
| 1 | 1.00 | 1.00 | 1.00 | 1.00 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2 | 1.91 | 1.76 | 2.13 | 1.79 | 95.37 | 87.76 | 106.3 | 89.54 |
| 3 | 2.79 | 2.47 | 3.22 | 2.54 | 92.99 | 82.19 | 107.2 | 84.54 |
| 4 | 3.62 | 3.17 | 4.25 | 3.23 | 90.52 | 79.23 | 106.3 | 80.85 |
| 5 | 4.41 | 3.53 | 5.17 | 3.69 | 88.17 | 70.66 | 103.5 | 73.87 |
| 6 | 5.12 | 4.05 | 6.26 | 4.23 | 85.30 | 67.44 | 104.4 | 70.43 |
| 7 | 5.79 | 4.44 | 7.44 | 4.65 | 82.68 | 63.46 | 106.3 | 66.44 |
| 8 | 6.37 | 4.63 | 8.50 | 4.92 | 79.64 | 57.82 | 106.3 | 61.53 |

Table 4.4: The timing results are obtained using 3D Elasticity mesh. The mesh has 411939 DOFs and 811392 elements. This table data is calculated using data from Table 4.3.




(a) SpeedUp graph




(b) Efficiency graph

Figure 4.3: The graphs have the speed-up and efficiency data for the FEDirectSolver obtained using Elasticity 3D problem with OpenMP threads. The FEDirectSolver class has two main stages assembling of data and factorization. The solutions represents the total time used by the FEDirectSolver to calculate solution. The mesh used contains 411939 DOFs and 811392 elements.

In last stage of the FEDomain direct solver is called *back solve*. In back solve stage the computed solution is allocated to the elements by storing provided solution into internal structures. The solution can be used by elements for post processing. This is implemented in parallel using OpenMP directives. The time consumed by back solve stage reduces with the increase in threads count as shown in Table 4.3. The maximum speedup of 8.5 is achieved by running the FEDirectSolver using 8 threads. The solution is provided to the elements through the *setSolution()* method.

Table 4.4 shows that FEDomain has achieved much better speed up for data assembly and back solve stage than factorization. The FEDomain package actually improves the performance of third party solvers. The FEDomain package facilitates its user to easily implement finite element solver and achieve better performance.

## 4.2 Static Condensation

Static condensation is a second method for calculating the finite element solution and the theory of static condensation is already discussed in section 2.2.3. Static condensation requires a partitioned mesh where a solution $u$ is computed by individually solving smaller systems of linear equations for each mesh partition $\mathcal{P}_i$ (sub-domain). Partitioning the mesh into $N$ sub-domains creates $N$ number of small finite element problems. The dimensions of each small problem will be less than whole mesh. The smaller systems of equations cannot be solved separately because of coupling across the partition's interfaces. Each partition's DOFs classified into two categories:

- The DOFs which are present in a single partition are called *internal dofs* for that partition.
- The DOFs which are present at the boundary of the $P_i$ and are present in more than one partition are called *interface dofs*.

For each partition the solution of its internal DOFs $u_i$ requires the solution for its interface DOFs $u_b$ as shown in Eq (2.66).

The FEDomain interface in Listing 3.14 is designed to support static condensation by adding a new field called partition_ids in the FEDomain constructor. The user has to provide partitioned mesh where each element lies in single partition and should be aware to which partition it belongs. All the mesh partitions should be

assigned a unique positive integer called *partition identification number*(PID). The FEDomain will ask each element for its PID through $getPartitionID()$ present in FEElement superclass in listing 3.16. C++ does not support template virtual functions in abstract classes and FEElement class is an abstract class so the positive integer data type has been assigned to PID. It was possible to avoid adding list of PIDs parameter in the FEDomain constructor interface as these could have been collected from the mesh elements. In FEDomain constructor the SOLVER_TYPE parameter is used to select between direct solver and condensation solver. If the partition_ids vector is empty the direct solver will be used by default.

The FEDirectSolver class is modified to include support for the static condensation solver. The FEDirectSolver class requires the list of partition ids for the condensation solver as well as a condensation flag to select between the two solution modes. The condensation flag is set to TRUE for condensation solver else it is set by default to FALSE at construction time. The new interface is of the FEDirectSolver is in Listing 4.2.

```
class FEDirectSolver: public FEElement{
public:
    FEDirectSolver(std::vector<FEElement*>& elements,
                   std::map<FE_UINT, FE_DATA>& dirichlet_constraints,
                   std::vector<FE_UINT>& partition_ids,
                   FE_UINT& total_system_dofs,
                   FE_UINT& max_dofs_in_elements,
                   MATRIX_TYPE& matrix
                   bool& condensation);
     FE_UINT getDofCount(void);
     FE_UINT getPartitionId(void);
     void getLoad(FE_vector& load);
     void getStiffness(FE_dense_matrix& stiffness);
     void getSystem(FE_equation& equation);
     void getConnectivity(FE_sparse_matrix& dof_ids);
     void getConnectivity(std::vector<FE_UINT>& dof_ids);
```

```
    void setSolution(std::vector<FE_DATA>& Dirichlet_value);
    void getResidual(std::vector<FE_DATA>& approx_solution,
                     std::vector<FE_DATA>& residual);
};
```

Listing 4.2: FEDirectSolver version 2

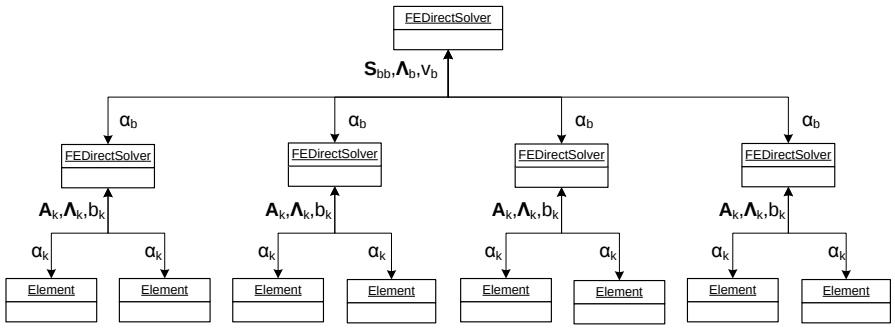



Figure 4.4: FEDomain class diagram. The FEDirectSolver object at top is master object which will be created by the user and provides it with mesh partitioned into 4 sub_domains. The master object is composed of 4 slave FEDirectSolver objects. Each each slave objects is allocated the specified partitions elements.

The design of the static condensation finite element solver is composed of multiple FEDirectSolver objects, one for each provided PID and these objects will be referred to as *client objects*. The client objects are composed inside a FEDirectSolver object will be referred to as *master object* as shown in Figure 4.4. The master object divides the set elements into subsets according to their PIDs. The master object functionalities are given below:

- The master object is provided with sets of elements and PIDs. It divides the set of elements into subsets according to the element's PID. The elements which are provided but do not belong to any specified partitions will be ignored.

- Due to a limited information about allocated DOFs, the client objects cannot differentiate among the internal and interface DOFs. The interface DOFs

information has to calculate the master object. The master object has data for all the allocated partitions. It calculates the interface DOFs which can be defined as a set containing the DOFs present in more than one partition and the Dirichlet DOFs present in $\mathcal{P}$. Let the interface DOFs are denoted as $DOF_B$.

- The master object encapsulates client objects. It is responsible for the construction and deletion of these objects. At the construction of a client object, it is provided with list of elements and interface DOFs along with other parameters.

- Finally master object has to compute the $u_B$ solution for the interface DOFs through (2.76). The computation of the $u_B$ requires $\boldsymbol{S}_{bb}^P$ and $\vec{v}_b^P$ from each client object. Each partition will have a subset of the interface DOFs, and a mapping data $\boldsymbol{\Lambda}_b^P$ will be required to construct $\boldsymbol{S}$ and $\vec{v}$. The solution is provided to all the client objects to compute their internal DOFs solution $u_j$.

The client object functionalities are given below:

- The client object collects allocated elements data.

- A client object has to provide it schur complement stiffness matrix $\boldsymbol{S}_{bb}^P$ (given in (2.74)), schur complement load vector $\vec{v}_b^P$ (given in (2.75)) and connectivity matrix $\boldsymbol{\Lambda}_b^P$.

- It compute solution for internal DOFs.

- The clinet object provides solution to allocated elements.

The client objects behave like an FEElement object which has to provide their data to master object. To keep the client object interface same as of finite element interaface, the FEDirectSolver class is inherited from the FEElement class. In the

condensation mode, the FEDirectSolver object will provide its allocated partition ID for getPartitionID(). Let $DOF_{B_j}$ be the boundary DOFs for the partition $\mathcal{P}_j$ than $DOF_B \supseteq DOF_{B_j}$. The get_dof_count() will provide number of interface DOFs $N_{B_j}$ in client object. The $\boldsymbol{S}_{bb}^P$ is a sparse matrix and in the FEElement class the getStiffness function requires a dense matrix.

For each partition, the $\boldsymbol{S}_{bb}^P$ and $\vec{v}_b^P$ are computed using (2.71) and (2.72), respectively. The Schur complement required $\boldsymbol{A}_P$ and $\vec{b}_P$ to be transformed into smaller matrices and vectors based in the partitions internal and boundary DOFs. The $\boldsymbol{A}_P$ has to be stored as four matrices $\boldsymbol{A}_{ii}$, $\boldsymbol{A}_{ib}$, $\boldsymbol{A}_{bi}$ and $\boldsymbol{A}_{bb}^P$ of smaller dimensions. The load vector $\vec{b}_P$ has to be kept as two smaller load vectors $\vec{b}_i$ and $\vec{b}_b^P$. In the ideal case the partition data should be directly assembled into sub-matrices and vectors. For each element data entry it has to be verified which data structure it will be stored in. The process of checking for each entry of element provided data makes the data gathering process extremely slow. The partition's elements data is stored as $\boldsymbol{A}_P$ and $\vec{b}_P$ using client objects local DOF numbering $\mathcal{N}_P$ which are further transformed into smaller structures.

### 4.2.1 FEEquation Class

The mesh elements provide their stiffness matrices and load vectors in local numbering $\mathcal{N}_K$ and the client objects maintain their allocated elements data in partition local numbering $\mathcal{N}_P$. The allocated elements can provide PID and their DOF's global mapping, but these elements do not have knowledge about their DOF's partition mapping. The mapping $\mathcal{N}_K$ to $\mathcal{N}_P$ is not constant as by repartitioning the mesh the element can be moved to a different partition and so that element's mapping information has to be altered as well. The way to populate the $\boldsymbol{A}_P$ and $b_P$ is to initially map $\boldsymbol{A}_K$ and $b_K$ into $\mathcal{N}$ and finally map global numbered data into $\mathcal{N}_P$ using (2.47) and (2.55).

The FEDirectSolver class was implemented to solve $\boldsymbol{A}u = \vec{b}$ which involves the assembling of the element data, computing the solution and providing the solution to the user elements. The FEEquation class in Listing 4.3 is introduced to encapsulate implementations of the direct solver and the static condensation solver in a class which can be reused for future implementations like distributed direct solver and keep solver classes clean of unnecessary or repeated code. The FEEquation class converts the user provided data into the desired data structures $\boldsymbol{A}_{ii}$, $\boldsymbol{A}_{ib}$, $\boldsymbol{A}_{bi}$, $\boldsymbol{A}_{bb}$, $\vec{b}_i$ and $\vec{b}_b$. For the direct solver the matrix $\boldsymbol{A}_{ii}$ represents mesh partition $\mathcal{P}_i$ internal DOFs mappings and $\boldsymbol{A}_{bb}$ represents mesh boundary DOFs interaction. The $\boldsymbol{A}_{ib}$ and $\boldsymbol{A}_{bi}$ is the internal to boundary DOFs interaction. If the user has provided data after applying constraints the $\boldsymbol{A}_{ib}$ and $\boldsymbol{A}_{bi}$ should be empty and $\boldsymbol{A}_{bb}$ should be a diagonal matrix. For the direct solver, the FEEquation object has to solve (2.66) to compute $u_i$, as $u_b$ are the Dirichlet DOFs and their values are provided by the user.

The FEEquation class constructor takes a list of internal DOFs, interface DOFs, matrix types and solution mode. For direct solver mode the condensation flag has to be set to false and the provided interface DOFs will be Dirichlet DOFs. In the case of static condensation, the condensation flag has to set to true and the interface DOFs should be the partition boundary DOFs and internal DOFs will also be the partition's internal DOFs. The FEEquation class does not interact with the user elements and it depends on its composing object. The FEDirectSolver object has to provide allocated elements data in $\mathcal{N}$ to its FEEquation object using FEEquation's setStiffness() and setLoad() given in Listing 4.3. For FEEquation object in direct mode the getStiffness() and getLoad() functions return $\boldsymbol{A}_{ii}$ and $\vec{b}_i$ while in condensation mode the Schur complement data $\boldsymbol{S}_{bb}^P$ and $\vec{v}_b^P$ is provided. The solve() function requires the interface DOFs solution $\alpha_b$ to compute the solution of internal DOFs $\alpha_i$. Again in direct mode it will be the values for the Dirichlet DOFs on other hand the solution to partition's boundary DOFs.

```
class FEEquation{
public:
   FEEquation(FE_UINT total_system_dofs,
              vector<FE_UINT>& internal_dofs,
              vector<FE_UINT>& interface_dofs,
              MATRIX_TYPE matrix, bool condensation);
   template<typename V> void getload(V& v);
   template<typename V> void setLoad(V& v);
   template<typename M> void getStiffness(M& m);
   template<typename M> void setStiffness(M& m);
   template<typename V1, typename V2>
   void solve(V1& bdr_sol, V2& int_sol);
};
```

Listing 4.3: FEEquation version 1.

The FEEquation object maintains data in $\mathcal{N}$ and cannot differentiate between internal and interface DOFs. The list of internal and interface DOFs are provided to FEEquation which are required for calculating the Schur complement of the partition data. The FEEquation class is aimed at solving the system of linear equations where the user has to provide system data represented as stiffness matrix and load vector. The FEDirectSolver object class has to construct and populate the data containers for $\sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K$ and $\sum_{K\in\mathcal{P}} \boldsymbol{\Lambda}_K^T \vec{b}_K$ for allocated elements and provide these containers to its FEEquation object. The design keeps the FEDomain data acquisition method and internal solution process separate so that if either of these methods changes the other one remains unchanged. The FEEqaution object obtains stiffness matrix and load vector data from the provided containers through the getValue() functions one by one which makes the data coping process very inefficient.

The alternative approach is to provide FEEquation object to the user provided elements. The FEEquation class is considered as a data container as it will keep

data to compute solution. It allow the internal implementation of the FEEquation class independent of its interface. The internal data structures can be selected according to the execution mode. The FEEquation class interface methods are modified to allow elements to insert their data. It should have same interface as of FEvector and FEMatrix classes. From FEEqution class setStiffness() and setLoad() functions have been removed and $addValue(index,\ value)$ and $addValue(row\_id,\ col\_id,\ value)$ are added to populate load vector and stiffness matrix respectively using global DOFs numbering. The new interface of the FEEquation class is shown in Listing 4.4.

```
class FEEquation{
public:
    FEEquation(FE_UINT total_system_dofs,
               vector<FE_UINT>& internal_dofs,
               vector<FE_UINT>& interface_dofs,
               MATRIX_TYPE matrix, bool condensation);
    void addValue(FE_UINT& index, FE_DATA& value);
    void addValue(FE_UINT& row_id, FE_UINT& col_id, FE_DATA& value);
    template<typename V> void getload(V& v);
    template<typename M> void getStiffness(M& m);
    void getSystem(FEEquation& eq);
    template<typename V1, typename V2>
    void solve(V1& bdr_sol, V2& int_sol);
};
```

Listing 4.4: FEEquation version 2, new interface functions addValue are added to insert stiffness matrix and load vector. The getLoad, getStiffness and getSystem functions are added to attain FEEquation class.

In case of static condensation mode the FEEquation class getStiffness() and getLoad() functions provide $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_{b}^{P}$,respectively while in direct solver mode these functions return $A$ and $\vec{b}$, respectively. The FEDirectSolver class is composed of the FEEquation object. The FEDirectSolver class acts as a wrapper

which manage the interaction between group of the FEDirectSolver objects. The $getSystem(FEEquation\&)$ is added into FEElement class interface. This method allows the FEDirectSolver object to pass its FEEquation object's reference to allocated elements. The elements have to populate it with $\mathbf{\Lambda}^T \boldsymbol{A}_K \mathbf{\Lambda}$ and $\mathbf{\Lambda}^T \vec{b}_K$ .

### 4.2.2 Client object as Element object

In static condensation mode, the FEDirectSolver class master object is composed of FEDirectSolver client object, one for each allocated mesh partition. The FEEquation class implements the solution for the system of equations. In master object, its client objects act like the user provided finite element object $K$. In order to assemble schur complement $\boldsymbol{S}$ and $\vec{v}$, the master object has to collect $\boldsymbol{S}^P$ and $\vec{v}^P$ from all its client objects. It is similar to the user provided element class which provides $\boldsymbol{A}_K$ and $\vec{b}_K$ to assemble $\boldsymbol{A}$ and $\vec{b}$, respectively. The collection of system data is similar for the client and master objects. The client objects gather data from element objects through FEElement interface. The FEDirectSolver class is inherited from the FEElement class so that single FEDomainSolver class implementation can be used to create master as well as client objects. The master object uses FEElement interface to gather data from client objects. The master object gather interface DOFs and computes solution for non Dirichlet interface DOFs.

Each client object has FEEquation object in condensation mode to handle internal data storage and manipulation. The FEEquation object is created at the construction stage. The FEDirectSolver does not deallocate its memory space until the destructor, to avoid unnecessary memory allocation, the assembly of the partition data is delayed until setSolution() is called. The client objects assemble their data from the allocated elements and compute the Schur complement of the assemble data. The Schur complement data of the partition is computed by implementing (2.71) and (2.72). This requires assembling the partition data according to internal and interface DOFs into multiple smaller data structures $\boldsymbol{A}_{ii}$, $\boldsymbol{A}_{ib}$, $\boldsymbol{A}_{bi}$,

$\boldsymbol{A}_{bb}$, $\vec{b}_i$ and $\vec{b}_b$. The FEEquation class is using PARDISO solver to compute LU factors of $\boldsymbol{A}_{ii}$. In future, the support for other linear algebraic solvers will be added into FEEquation class. The class diagram of FEDirectSolver is shown in Figure 4.5.

### 4.2.3 Shared Memory Solvers Class Diagrams

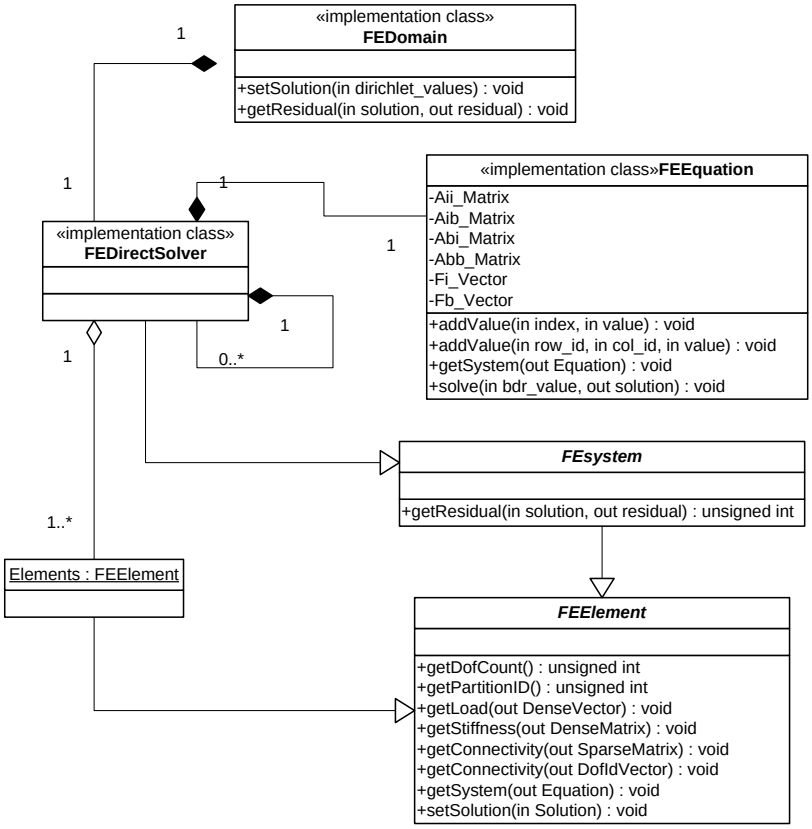



Figure 4.5: FEDomain class diagram

Figure 4.5 represents the FEDomain class diagram for the direct solver and static condensation solver mode for the shared memory architectures. The user will creates a FEDomain object and will define the mode of the solver. The FEDomain object (in both direct and static condensation mode) will be composed of a single FEDirectSolver object which will act as a master object. The master object will always have an FEEquation object and also it will have associative

with the elements. In direct mode, the master object will collects all the provided elements data into its FEEquation object and provide the solution to the elements in back solve. In static condensation mode the user has to provide the partition ids of the mesh and the master object will create client objects, one for each provided partition ids. The FESystem class in the Figure 4.5 is introduced for the FEDomain residual calculation and will be explained at a later chapter.

## Schur Complement Implementation

In the FEDomain package the parallel regions are implemented using OpenMP pragmas and FEDomain does not support nested parallelism. The parallelism is implemented at the root level operations like gathering data or data manipulations. All the composite FEDirectSolver client objects are constructed and treated sequentially and their internal operations like computing $\boldsymbol{S_{bb}}$ are performed in parallel.

| # of RHS | CUBE5 | CUBE6 |
|---|---|---|
| 1 | 316.70 | 6186.1 |
| 2 | 189.97 | 3372.8 |
| 4 | 101.22 | 1893.8 |
| 8 | 61.02 | 1193 |
| 16 | 54.17 | 863 |
| 32 | 35.06 | 673.5 |
| 64 | 20.09 | 348.7 |
| 128 | 25.54 | 381.1 |
| 256 | 19.71 | 207.1 |
| 512 | 24.08 | 291.1 |
| 1024 | 31.54 | 306.8 |

Table 4.5: PARDISO multiple RHS solve timings. Total RHS are 2048 and #RHS are the number of right hand sides provided to solver in a single solve call. The second and thrid column represents the time taken to solve for the CUBE5 and CUBE6 stiffness matrices, respectively. The CUBE5 and CUBE6 matrices dimensions are 53907 and 411939, respectively.

The PARDISO solver can provide sequential and parallel direct solution of symmetric as well as unsymmetric systems of linear equations on shared memory processor. According to the technical report [40], the PARDISO has performed best among other linear symmetric solvers. It is capable to solve the system of linear equations with multiple right hand sides. The user has to provide multiple right hand sides in consecutive memory containers. The size of this containers should be greater than or equal to the number of right hand sides $\times$ size of right hand side. To obtain the solution, the user has to provide the same size consecutive memory container to the PARDISO solver. The experiments have showed, the PARDISO time consumption to compute solution for multiple right hand sides by a single solve call is less than the time it consumes to compute the solution for these right hand sides one by one. The timing comparison for solving the total of 2048 right hand sides for cube5 and cube6 meshes using various number of right hand sides is shown in Table 4.5. The *step size* (#RHS) is the number of right hand sides provided to the solver in each solve call. In both the cases step size 256 has produced best results.

| | QUATER8 (2D) DOFs=229664 | | | | CUBE5 (3D) DOFs=53907 | | | |
|---|---|---|---|---|---|---|---|---|
| | Total RHS = 1024 | | Total RHS = 512 | | #RHS = 1024 | | #RHS = 512 | |
| RHS | Time | Mem (KB) | Time | Mem (KB) | Time | Mem (KB) | Time | Mem (KB) |
| 1 | 112.60 | 9293156 | 55.68 | 4699582 | 159.18 | 2042489 | 78.66 | 1146076 |
| 8 | 51.94 | 9292711 | 25.75 | 4699237 | 29.89 | 2042217 | 15.08 | 1146924 |
| 64 | 19.66 | 9293245 | 9.42 | 4699890 | 10.09 | 2042967 | 5.03 | 1147400 |
| 128 | 18.65 | 9292844 | 8.86 | 4699734 | 10.31 | 2038975 | 5.19 | 1146844 |
| 256 | 16.99 | 9292811 | 9.17 | 4699728 | 9.77 | 2045518 | 4.92 | 1148366 |

Table 4.6: PARDISO multiple RHS solve timings and memory consumption

Table 4.6 contains timing and memory consumption data for QUATER8 (2D Poisson) and CUBE5 (3D Elasticity) linear algebra problems and again all the solver data are obtained from the PARDISO solver. Table 4.6 data suggests that the memory consumed by the solver for the triangular solve depends on the total number of right hand sides. The total time requires to obtain the solution for all

the right hand side depends on #RHS. The $\boldsymbol{S}_{bb}^{P}$ computation algorithm is given in Algorithm 2. To achieve better performance it is necessary to collect consecutive non empty columns from $\boldsymbol{A_{ib}}$. Multiple RHS requires more memory to store data and their results outside PARDISO in dense vector. PARDISO does not support sparse vector so the right hand sides $\vec{c}$ are provided and solution $\vec{t}$ is obtained in dense 1D containers of size internal DOFs counts $\times$ step size. The $\vec{s}$ is a sparse vector which is used to save memory and computations while subtracting it from $\boldsymbol{A}_{bb}$.

```
1 c⃗ ← getConsecutiveNonEmptyCols(A_ib)
2 while !Empty(c⃗) do
3 |   t⃗ = L^P U^P c⃗
4 |   s⃗ = A^P_bi t⃗
5 |   A^P_bb_:ids[i] −= s⃗_i
6 |   c⃗ ← getNextConsecutiveNonEmptyCols(A_ib)
7 end
```

**Algorithm 2:** Psudocode for computing $Sbb^{P}$ for each mesh partition. It will be implemented in each FEDirectSolver client object.

During experiments it is observed that PARDISO solver remain at the solve stage if provided with all zero RHS. Every time a column is obtained from $\boldsymbol{A}_{ib}^{P}$ it has to be checked that column is not empty before providing it to solver.

## Direct Solver Timing

Table 4.7 has the timing information for the FEDirectSolver class in direct mode. These timing are obtained using new implementation having FEEquation object. The CUBE5 and CUBE6 meshes are to solve 3D Elasticity problem .

| Mesh | Cube5 | Cube6 |
|---|---|---|
| Constructor Time (sec) | – | 5.52 |
| Total DOFs | 53907 | 411939 |
| Dirichlet DOFs | 9222 | 36870 |
| # non zero in A | 1889487 | 16355129 |
| # non zero in LU | 26503683 | 5200091653 |
| Solver Mem (kB) | 275025 | 4557979 |
| Solution Time (sec) | 4.96 | 84.01 |

Table 4.7: PARDISO timing and memory data while being used in FEDirectSolver solver (*direct* mode). The cube5 and cube6 are 3D meshes obtained using GMSH software and used to solve an Elasticity 3D problem.

Table 4.8 has the timing of static condensation non partitioned and partitioned meshes where the non partitioned timing are better than partitions meshes. The PARDISO solver memory information is collected from the solver at run time. Table 4.8 displays the memory consumption and computation time for CUBE6 mesh with 5 partitions. The METIS [52] package is used to partition CUBE6 mesh and all the produced partitioned meshes have same number of DOFs and elements. The *Construction_Time* in Table 4.8 represents the time consumed by FEDirectSolver constructor in condensation mode for partitioned meshes which is almost similar. The FEDirectSolver class constructor in static condensation mode perform tasks like filtering the elements according to their PID's, computing the interface DOFs among all the mesh partitions, and construct composite FEDirectSolver objects for each PID. The DOF_B are the mesh interface DOFs which also include the Dirichlet DOFs. The *Sbb_Time* represents the time consumed for populating $\boldsymbol{S}_{bb}$ in master object. It requires computation of $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_{b}^{P}$ for all the client objects and adding each client object Schur complement data into $\boldsymbol{S}_{bb}$ and $\vec{v}_{b}$. To avoid unnecessary memory consumption the elements data is not assembled in the FEDomain solver until $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_{b}^{P}$ are required.

The *Sbb_A_NNZ* represents the amount of data in $\boldsymbol{S}_{bb}$ upper triangle as 3D elasticity is a symmetric problem, The *Sbb_LU_NNZ* represents the number of non zero entries in $\boldsymbol{S}_{bb}$ numerical factorization. The $\boldsymbol{S}_{bb}$ is a densely populated matrix so does its factors are as can be seen in Table 4.8. The *Total_LU_NNZ* represents total amount of factorization data for each FEDirectSovler client object and interface DOFs from master object. The *Solver_Mem* represents the total memory consumed by all the PARDISO solvers. As the mesh partitions increases so does the memory consumption of PARDISO as can be seen in 9th rows of Table 4.8. The memory consumed by each client object deceases and the interface DOFs increases as the number of partitions increases in the provided mesh. All the PARDISO solver memory consumption and total non zero data count are provided by the PARDISO solver.

| Partitions | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Constructor Time | 7.05 | 7.14 | 7.14 | 7.21 | 7.24 |
| Sbb_DOFs | 36870 | 45756 | 51519 | 55137 | 58686 |
| Sbb_IDOF | 0 | 8886 | 14649 | 18267 | 21816 |
| Sbb Time | 0 | 1453.18 | 2148.9 | 2645.83 | 3312.1 |
| Sbb_A_NNZ | 0 | 39484941 | 107303925 | 166850778 | 237979836 |
| Sbb_LU_NNZ | 0 | 39764581 | 107765170 | 167425690 | 238666817 |
| Sbb_Mem (kB) | 0 | 971649 | 2429645 | 3677522 | 5150820 |
| Total_LU_NNZ | 529995920 | 374464081 | 375094627 | 397287579 | 440321571 |
| Solver_Mem (kB) | 4627245 | 3973634 | 4920007 | 5597337 | 7153935 |
| Solution Time (sec) | 94.0114 | 1491.45 | 2218.48 | 2867.65 | 4538.45 |

Table 4.8: PARDISO solver time and memory consumption for the FEDirectSovler in the condensation mode using Cube6 mesh (411939 DOFs and 811392 elements) from 1 to 5 partitions.

## 4.3 Complexity

The complexity of the direct solver depends of the complexity of the PARDISO solver. The complexity of the PARDISO solver for 2D and 3D meshes is shown in Table 4.9 (see [36]).

| | 2D | 3D |
|---|---|---|
| Symbolic Factorization | $\mathcal{O}(nlog(n))$ | $\mathcal{O}(n^{4/3})$ |
| Numerical Factorization | $\mathcal{O}(n^{3/2})$ | $\mathcal{O}(n^2)$ |
| Triangle Solve | $\mathcal{O}(nlog(n))$ | $\mathcal{O}(n^{4/3})$ |

Table 4.9: PARDISO time complexity

In the case of the static condensation solver the calculation of $\boldsymbol{S}_{bb}^P$ requires multiplication of two matrices $\boldsymbol{A}_{ib}^P$ and $\boldsymbol{A}_{ib}^P$ from the LU factors of $\boldsymbol{A}_{ii}^P$. The algorithm of the $\boldsymbol{S}_{bb}^P$ computation is defined in the Algorithm 2.

In line 3 of Algorithm 2 the column $\vec{c}$ is provided to the PARDISO solver to produce the solution as $\vec{t}$. Let $I_P$ be the total number of partition internal DOFs and $B_P$ be the interface DOFs. The complexity of line 3 of Algorithm 2 for 2D mesh is $\mathcal{O}(I_P \log(I_P))$ and for 3D problem is $\mathcal{O}(I_P^{4/3})$. The $\vec{c}$ and $\vec{t}$ can be scarcely populated vectors but PARDISO requires these containers to be dense. The maximum complexity of the matrix vector multiplication in line 4 is $\mathcal{O}(I_P B_P)$. In line 5 a sparse vector is added in a column of $\boldsymbol{A}_{bb}^P$ which can have a maximum complexity of $\mathcal{O}(B_P)$. The whole process repeats itself $B_P$ times. The total complexity for $\boldsymbol{S}_{bb}^P$ calculation for 2D mesh is $\mathcal{O}(max((I_P \log(I_P))B_P, I_P B_P^2, B_P^2))$ and for 3D mesh $\mathcal{O}(max(I_P^{4/3} B_P, I_P B_P^2, B_P^2))$.

Let suppose a $D$ dimension problem has to be solved. The mesh for it has been partitioned in such a way that there are $s$ number of partitions in each dimensions and each partition has $n+1$ DOFs in each dimensions. The total number of partitions $P$ in mesh is

$$P = s^D$$

and total number of DOFs $N$ in the mesh is

$$N = (sn)^D$$

than

$$n = (N/P)^{1/D}$$

partition's interface DOFs $B_P$ are

$$B_P \approx (N/P)^D$$

partition's internal DOFs $I_P$ are

$$I_P \approx N/P$$

By putting the values of $B_P$ and $I_P$ in the Schur complement solver complexity becomes

$$\mathcal{O}((N/P)^{((2^D-(D-2))/D)})$$

so for the 2D mesh the Schur complement complexity becomes

$$\mathcal{O}((N/P)^2)$$

and for the 3D mesh the Schur complement complexity becomes

$$\mathcal{O}((N/P)^{7/3}).$$

The total number of interface DOFs $B$ in D dimension mesh are

$$B = s^D n = P(N/P)^{1/D} = N^{1/D} P^{((D-1)/D)}.$$

The complexity of $\alpha_B$ computation is

$$\mathcal{O}(B^2) = \mathcal{O}(N^{2/D} P^{2(D-1)/D})).$$

Total complexity of the FEDomain static condensation solver is

$$\mathcal{O}(max(t_S, t_B)) = \mathcal{O}(max((N/P)^{((2^{D+1}-D)/D)}, (N/P^{D-1})^{2/D}))$$

## 4.4 Conclusion

In this chapter the FEDomain package is introduced as a C++ finite elements solver package developed for the shared memory architectures. The focus of the package is to provide a user with a finite element solver which any C++ finite elements application developer can add into his application with the minimum modification. The FEDomain package can compute the solution of the symmetric and unsymmetric linear equation systems. The package requires the user to represent the mesh elements as the C++ classes and provides the list of these element objects to the FEDomain objects at construction stage. The FEDomain solvers require element objects to provide their data ($\boldsymbol{A}_k$, $\boldsymbol{\Lambda}_k$ and $\vec{b}_k$) through standard element interface. The standard interface is given as a FEElement class in FEDomain package. The user has to publicly inheriting all element classes from the FEElement abstract class and implements all the interface methods in element classes. The FEDomain package also provides two finite elements direct solvers for the shared memory systems. The FEDirectSolver direct solver is implemented to find the solution of the whole mesh. The FEDirectSolver Static Condensation solver is implemented to find the solution for the partitioned mesh. For the FEDirectSolver direct solver all the three stages (acquisition of data, PARDISO solver and distribution of solution) are implemented using parallel algorithm with OpenMP.

In Static condensation solver, the parallelism is implemented at the root level. All the client objects and the master object of the FEDirectSovler condensation are implemented in parallel. Each FEDirectSolver client objects are implemented

using parallel algorithms. These objects are allocated one by one to the processors. Once all the client objects have computed their $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_{b}^{P}$, the computational resources are provided to the FEDirectSolver master object. The master object uses these resources to attains the client objects data and compute the interface DOFs values. It provides each of its client objects with the interface DOFs values and allows each client objects to solve its internal DOFs values. This method is currently implemented in static condensation as this gives each partition an equal computational opportunity. If all the partitions have similar number of internal and interface DOFs then the Schur complement computation for these partition takes a similar time.

Table 4.10 represents the timing results for the FEDirectSolver Static Condensation solver for the multiple 2D and 3D meshes used to solve Poisson and Elasticity problems, respectively. The second column shows the interface DOFs present in the partitioned mesh. The number of interface DOFs increases with the raise in the mesh partitions. The third column represents the number of entries added into the $\boldsymbol{S}_{bb}$ from all the FEDirectSolver client objects. Its increases with the rise in partitions for each mesh. The FEDirectSolver Static Condensation solver performs Schur complement for each partitions one by one and each client object perform the Schur complement algorithm in parallel. The time consumed by these objects for Schur Complement computation are added to attain the total time the FEDirectSolver spents for calculating all $\boldsymbol{S}_{bb}^{P}$. The Schur Complement time is displayed in fourth column of the Table 4.10. The fifth column represents the time consumed by all the client objects to copy their $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}^{P}$ into the FEDirectSolver master object.

For the 2D meshes the time consumed for $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}^{P}$ calculation and copying increases with the raise in partitions. For 3D meshes these times reduces with the raise in mesh partitions. The similar time pattern continues in the total time

| Mesh Parts | Interface DOFs | NNZ added to $\boldsymbol{S_{bb}}$ | $\boldsymbol{S_{bb}^P}$ timing (sec) for Comput | Copying | Total Time(sec) | PARDISO Mem (kB) | NNZ |
|---|---|---|---|---|---|---|---|
| Quater8 (Poisson 2D) Total DOFs(230465) Total Elements(460931) | | | | | | | |
| 2 | 1328 | 1400514 | 11.901 | 0.053 | 17.6874 | 4826927 | 9807074 |
| 3 | 1592 | 2519349 | 23.169 | 0.103 | 28.7923 | 4829988 | 9596615 |
| 4 | 1841 | 3831927 | 29.235 | 0.146 | 33.9255 | 4763129 | 9441451 |
| 5 | 2185 | 6053256 | 43.044 | 0.207 | 50.7696 | 5057392 | 9411136 |
| 6 | 2316 | 7022921 | 34.564 | 0.215 | 42.5034 | 4628149 | 9390914 |
| 7 | 2472 | 8264713 | 39.841 | 0.232 | 47.3875 | 4372747 | 9219985 |
| 8 | 2771 | 10924099 | 41.422 | 0.305 | 47.1122 | 4506097 | 9367279 |
| Quater9 (Poisson 2D) Total DOFs(920065) Total Elements(1840131) | | | | | | | |
| 2 | 2686 | 5830222 | 89.863 | 0.295 | 104.213 | 39354313 | 46490532 |
| 3 | 3259 | 10808449 | 164.263 | 0.590 | 177.451 | 40067039 | 45710841 |
| 4 | 3821 | 16966849 | 191.343 | 0.954 | 208.447 | 40198292 | 44972000 |
| 5 | 4474 | 25718156 | 211.964 | 1.831 | 228.163 | 41584387 | 45077691 |
| 6 | 4632 | 28090057 | 195.717 | 1.151 | 209.769 | 36631914 | 44886375 |
| 7 | 5380 | 40679792 | 229.262 | 2.008 | 245.553 | 39078202 | 44974507 |
| 8 | 5660 | 45966474 | 223.849 | 1.709 | 245.419 | 36736737 | 44471315 |
| Cb_53907 (Elasticity 3D) Total DOFs(53907) Total Elements(104648) | | | | | | | |
| 2 | 11358 | 150653200 | 123.15 | 51.78 | 178.38 | 11248265 | 18332424 |
| 3 | 12915 | 211915036 | 125.81 | 52.28 | 183.29 | 9011400 | 18676024 |
| 4 | 13413 | 234612867 | 120.17 | 44.43 | 169.89 | 7173906 | 18415385 |
| 5 | 14283 | 276226424 | 121.16 | 38.20 | 165.86 | 6261456 | 20064430 |
| 6 | 14799 | 301968684 | 115.99 | 31.48 | 153.35 | 5480741 | 20884211 |
| 7 | 15216 | 323899173 | 122.23 | 28.34 | 157.63 | 4882505 | 21275395 |
| 8 | 15603 | 345432079 | 102.06 | 29.30 | 150.74 | 4446707 | 22674156 |
| Cb_85103 (Elasticity 3D) Total DOFs(85104) Total Elements(166828) | | | | | | | |
| 2 | 19032 | 404653396 | 525.309 | 179.041 | 712.346 | 27754020 | 28488984 |
| 3 | 20721 | 511124194 | 562.364 | 157.326 | 729.136 | 21034537 | 28979221 |
| 4 | 21561 | 569622775 | 603.584 | 128.084 | 741.592 | 16757094 | 28559086 |
| 5 | 22419 | 633251315 | 528.153 | 110.098 | 649.413 | 14238878 | 31097947 |
| 6 | 22860 | 666969531 | 583.839 | 98.0312 | 692.663 | 12225119 | 30906377 |
| 7 | 23505 | 717169443 | 651.062 | 86.7967 | 750.263 | 10891256 | 31080380 |
| 8 | 23964 | 757270944 | 609.978 | 74.1710 | 695.888 | 9864984 | 33963251 |
| Cb_146781 (Elasticity 3D) Total DOFs(445923) Total Elements(286548) | | | | | | | |
| 2 | 26763 | 790706924 | 1055.511 | 467.629 | 1536.090 | 69858508 | 61509775 |
| 3 | 28872 | 978094224 | 1092.372 | 412.345 | 1521.690 | 52694606 | 61926469 |
| 4 | 29688 | 1055615711 | 1001.947 | 285.810 | 1303.880 | 41357790 | 61923862 |
| 5 | 31554 | 1242757418 | 1041.395 | 308.756 | 1370.000 | 36226749 | 68072665 |
| 6 | 32337 | 1326566357 | 977.949 | 247.854 | 1246.230 | 31295571 | 65960725 |
| 7 | 32544 | 1351168456 | 968.155 | 209.746 | 1197.110 | 27214948 | 66603332 |
| 8 | 32949 | 1397638001 | 902.706 | 156.457 | 1078.090 | 24312238 | 63958023 |

Table 4.10: The table contains timing and memory consumptions of the FEDirectSolver Static Condensation solver. The second column represents the interface DOFs of partitioned meshes. The third column represents the amount of data added from all the client objects. The fourth and fifth columns represent the time consumed for calculating all FEDiectSolver client objects $\boldsymbol{S_{bb}^P}$ and time required to all these $\boldsymbol{S_{bb}^P}$ into $\boldsymbol{S_{bb}}$. The sixth column represents the total time consumed by the FEDiretSolver to compute the solution. The seventh and eighth columns represents all the PARDISO solvers memory consumed and amount of NNZ stored.

required by the FEDirectSovler Static condensation as can be seen in the sixth column. The seventh and eighth column in Table 4.10 represents the total memory consumed and the total non zero data stored by all the PARDISO solvers. For the 2D meshes the PARDISO solver consumed similar amount of memory and stored similar amount of data. For the 3D meshes the PARDISO solver's memory consumption increases with the raise in the mesh partitions while on other hand the data stored reduces as can be seen in Table 4.10. All the PARDISO solver information data is obtained from the solver itself during execution. The PARDISO solver memory consumption represents the memory consumed by all the FEDirectSolver client objects and the FEDirectSolver master object's PARDISO solver and same is true for the PARDISO NNZ. The PARDISO solver memory consumption is shown in Figure 4.6 and FEDirectSolver Static condensation total time consumed is shown in Figure 4.7.

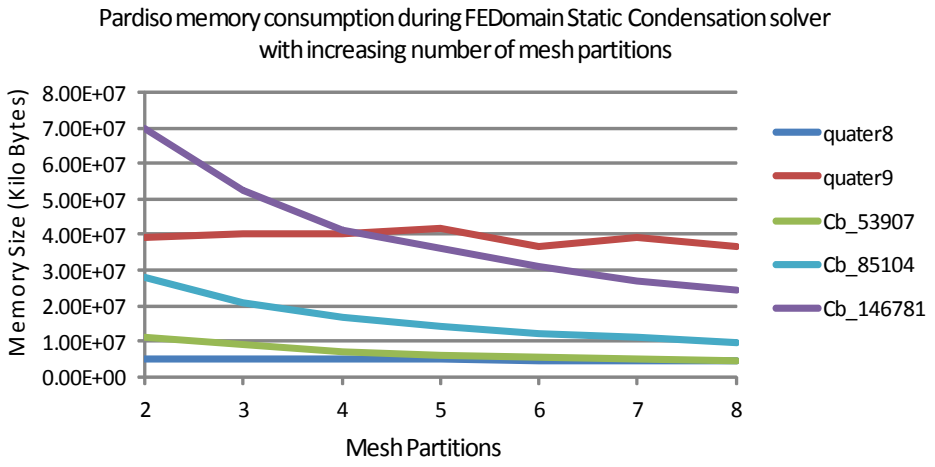



Figure 4.6: The FEDirectSolver Static Condensation solver PARDISO memory consumption of 2D and 3D meshes using 2 to 8 meshes.

| Mesh Name | Total DOFs | Dirichlet DOFs | PARDISO | | Total Time (sec) |
|---|---|---|---|---|---|
| | | | Mem (kB) | NNZ | |
| quater8 | 230465 | 801 | 105480 | 10515549 | 7.16 |
| quater9 | 920065 | 1601 | 487040 | 50400874 | 27.812 |
| Cb_53907 | 53907 | 9222 | 254574 | 26514787 | 5.279 |
| Cb_85104 | 85104 | 16506 | 366473 | 39980939 | 8.63 |
| Cb_146781 | 445923 | 47430 | 3635120 | 433581043 | 71.495 |

Table 4.11: The table contains PARDISO results for the FEDirectSolver direct solver for multiple 2D and 3D meshes for Poisson and Elasticity respectively.

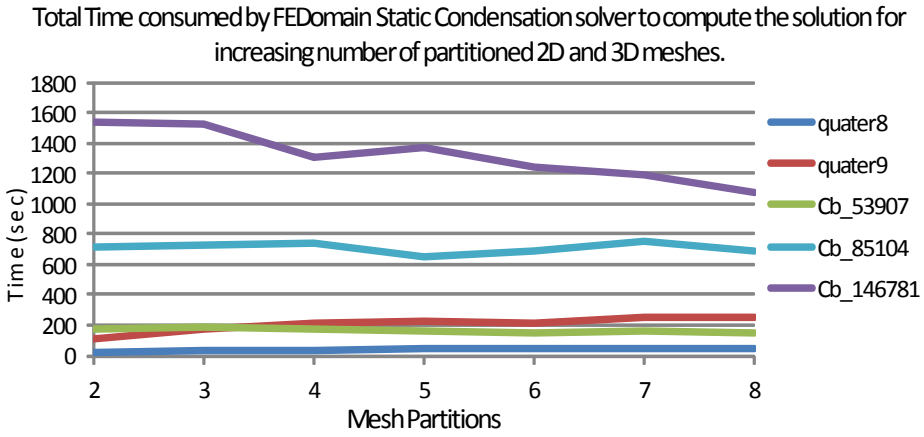



Figure 4.7: The FEDirectSolver Static Condensation solver time consumption for 2D and 3D meshes using 2 to 8 partitions.

Table 4.11 has the PARDISO solver data (computation time and memory acquired) for the meshes given in Table 4.10 for the FEDirectSolver direct solver. All the solutions are computed using eight OpenMP threads as was for the FEDirectSolver Static condensation. The comparison between the Tables 4.10 and 4.11 shows that the Direct solution method has performed better than the Static condensation solver. The FEDirectSolver direct solver has performed 2.47 for quater8, 3.474 for quater9, 28.555 for Cb_53907, 60.636 for Cb_85104 and 15.079 for Cb_146781 speedup than the FEDirectSolver condensation method. The PARDISO solver overall memory consumption is much better in FEDitectSovler direct method than for the Static condensation method. The PARDISO solver for

quater8 mesh consumed 45.76, for quater9 mesh consumed 80.8, for Cb_53907 mesh consumed 17.47, for Cb_85104 mesh consumed 26.92 and for Cb_146781 mesh consumed 6.69 less memory than the FEDirectSolver static condensation solver. For the shared memory architecture the FEDirectSolver direct method has performed much better then the FEDirectSolver static condensation method. The static condensation method will allow to implement the FEDomin package on distributed memory machine and will be discussed in next chapter.

# Chapter 5

# FEDomain Distributed Memory FE Solvers

This chapter is aimed to develop a finite element solver for the distributed memory architectures. This work is based on the developments illustrated in Chapter 4. It implements the domain decomposition algorithms which divides the large system of equations into multiple small systems of equations. These small systems of equations can be independently solved if their Dirichlet and boundary DOFs values are provided to them. This has already discussed in Section 4.2 as a static condensation solver. In this section the algorithm was implemented for the shared memory solver.

The Distributed Finite Element Solvers (DS) will be developed for the finite elements application developers using object oriented paradigm and C++ language for development. The users should be familiar with the MPI library as they will have to initialize and finalize MPI execution environment. The user application has to fulfil the prerequisites like inheriting all the finite element classes from the abstract FEElement class given in FEDomain package defined in Listing 3.16. DS require the partitioned mesh $\mathcal{P}$ where sub-partitions $\mathcal{P}_i$ should not overlap. There are many third party software available for partitioning $\mathcal{P}$ such as libMesh [54],

METIS and ParMETIS [52] developed for the shared memory and distributed memory systems.

## 5.1 Distributed Solver Interface

In FEDomain package the FEDomainMPI class defined in Section 3.8 represents the distributed architecture implementation. The FEDomainMPI class is implemented using Facade design pattern and provides interface to and encapsulates the implementation of subsystems . The FEDomainMPI class interface is designed similar to the FEDomain class. The user implemented finite element classes should be inherited from the FEElement class. Unlike shared memory applications the distributed memory processes do not share memory. The data has to be distributed among the MPI processes.

### 5.1.1 Distribution of the mesh

In FEDomain Schur complement solver discussed in Section 4.2 the partitioned mesh $\mathcal{P}$ was allocated to the master object. The master object on bases of the partition ids, provided by the user, creates client objects one for each partition. During distributed direct solvers development it was aimed to keep the same initialization routine. The user should have no knowledge of the client objects and the master object should be responsible for the initialisation, management and deletion of the client objects. The master object should allocate partition ids to the client objects.

The MPI library is selected to implement parallel process because it is easily available and not operating system dependent. The MPI library has its execution environment which initiates the parallel processes. In MPI environment the processes are created by a mpirun command on the multiple computational nodes. Each MPI process executes the user program as an individual applica-

tion which can communicate with the other applications running in other MPI processes. An MPI process can spawn child processes through *MPI_Comm_spawn* and *MPI_Comm_spawn_multiple* routines. These routines require a command as a parameter which will execute an application in the child or client process. These routines cannot be used in the FEDomain package because of the following reasons:

- The MPI spawn routines require the command to execute a program in child process. The name and arguments of command are selected by the user. To use these MPI routines, the use have to implement the applications to run in client processes.

- The element objects are provided to the master object. MPI does not allow transferring of C++ objects. The allocation of the elements to client object is not possible.

Due to the above reasons, the spawning of client objects from master object is not possible. It is possible to start master and client MPI process from the start and the mesh is read in all these processes. The master object should allocate mesh partitions among the client objects. The drawback of this design is the wastage of the memory by providing unwanted element objects to all the FEDomainMPI objects. This scenario cannot be avoided if the master object is responsible for the allocation of partition ids. The user has no information about the mapping of the mesh partitions to the FEDomainMPI client objects.

The user selects the mesh and has to construct and provide the element objects to the FEDomain package. In the final design of the DS, it is decided that it will be the user's responsibility to map the mesh partitions among the FEDomainMPI client objects. It will allow the user to create objects for the elements present in the allocated mesh partitions for each MPI process. This design will be memory efficient as the subset of the mesh elements objects is required for each MPI

process. The FEDomainMPI class interface is given in section 3.8. The FEDomainMPI interface allows the user to allocate the partitions to the FEDomainMPI client objects. The user can allocate a subset of partitions to a FEDomainMPI client object where no partition should be allocated to multiple FEDomainMPI objects. The user has given a choice of either provide all the elements to all the FEDomainMPI client objects, or just the elements of the allocated partitions to the FEDomainMPI client objects.

In the final version of DS solvers, the MPI library script is responsible for the initialization, communication and termination of parallel MPI processes. These processes are constructed and assigned at the start of the parallel application. Each one of the FEDomainMPI master and client objects are allocated to unique MPI process. All the FEDomainMPI client objects are allocated unique mesh partitions. The master object is not allocated any mesh partition. The FEDomainMPI client objects filter provided elements to separate the allocated partitions elements. It is advised that the user provides only the allocated partition elements. It is an optimal option as it will reduce filtering and allows the client objects to consume less memory per MPI process. The FEDomainMPI constructor requires the list of elements of the allocated partition, the Dirichlet data, and the allocated partitions ids.

## 5.2 DOFs notations

- $TDOF_j$ is the set of DOFs present in elements allocated to FEDomainMPI client object $\mathcal{C}_j$.
- $IDOF_j$ is the subset of $TDOF_j$ which are only present in elements allocated to $\mathcal{C}_j$. These are called internal DOFs of $\mathcal{C}_j$.
- $BDOF_j$ is the subset of $TDOF_j$ which also exist in other FEDomainMPI client objects. It is called interface DOFs of $\mathcal{C}_j$.

- $TDOF_0$ is a set of master object's DOFs. Its is created by collecting all the partitions $BDOF$s.

$$TDOF_0 = BDOF_1 \cup BDOF_2 \cup \cdots \cup BDOF_N.$$

- $BDOF_0$ is a set of Dirichlet DOFs ($DDOF$) which are present in $TDOF_0$. The solution for these DOFs are provided by the user.

$$BDOF_0 = TDOF_0 \cap DDOF.$$

- $IDOF_0$ is a set of non Dirichlet DOFs in $TDOF_0$. These are master object internal DOFs.

$$IDOF_0 = TDOF_0 \setminus BDOF_0.$$

## 5.3 FEDomainMPI

The FEDomainMPI class behaves differently for the master and client modes. In MPI process 0 the FEDomainMPI object is in master mode. The master object calculates the solution of the $IDOF_0$. The client objects have to calculate the solution of their internal DOFs. These also calculate the $\boldsymbol{S}_{bb}^P$ and $\vec{v}_b^P$ by implementing (2.71) and (2.72), respectively. Table 5.1 represents the $TDOF_0$ size and number of non zero entries in $\boldsymbol{S}_{bb}$ upper triangle. The table has comparison between Elasticity 3D and Poisson 2D problems. It can be seen that an increase in the mesh partitions results in a rise in the interface DOFs count. The increase in interface DOFs results in an increase of $\boldsymbol{S}_{bb}$'s non zero entries. The $\boldsymbol{S}_{bb}$'s non zero entries are directly proportional to data container and PARDISO memory consumptions.

| | Elasticity 3D | | | | Poisson 2D | |
|---|---|---|---|---|---|---|
| | Cb1 (85103) | | Cb2 (146781) | | Q9 (920065) | |
| $N_P$ | $TDOF_0$ | $\boldsymbol{S}_{bb}$ NNZ | $TDOF_0$ | $\boldsymbol{S}_{bb}$ NNZ | $TDOF_0$ | $\boldsymbol{S}_{bb}$ NNZ |
| 2 | 2508 | 2282316 | 9636 | 39484725 | 1086 | 589155 |
| 3 | 4284 | 6776475 | 15783 | 106973376 | 1659 | 1375311 |
| 4 | 4887 | 6939847 | 19737 | 136116856 | 2222 | 2041646 |
| 5 | 5877 | 8746089 | 23595 | 160683485 | 2876 | 2803559 |
| 6 | 6480 | 9114492 | 25896 | 170170866 | 3033 | 2571827 |
| 7 | 6978 | 9037714 | 28080 | 171712080 | 3781 | 3728467 |
| 8 | 7467 | 9170828 | 29742 | 166339683 | 4063 | 3615403 |

Table 5.1: The table represents the $TDOF_0$ size and number of non zero entries in $\boldsymbol{S}_{bb}$ upper triangle for three meshes.

## 5.4 Distributed Direct Solver

The FEDomainMPI solver interface given in Listing 3.17 requires a user to provide the Dirichlet constraints data to all the FEDomainMPI objects. The user has to provide all the Dirichlet constraints data for the whole mesh into the FEDomainMPI master object. The client objects should ideally be provided by the Dirichlet constraints data for the DOFs present in the allocated partitions. The user can provide all the Dirichlet constraints data to all the client objects but that will inversely affect the FEDomain performance. The interface has modified in FEDomainMPI class from the FEDomain class. In FEDomain class the user has to provide a set of partitions to the master object, which will generate client objects. In DS solvers the user has to specify the mapping between the FEDomainMPI client objects to the mesh partitions.

A mesh can be partitioned using different algorithms and a mesh can be partitioned into different number of sub-domains. In both cases the number of interface DOFs among the partitions can vary. The FEDomain package does not require the user to calculate the interface DOFs every time the mesh is modified or partitioned.

The FEDomainMPI solver calculates the interface DOFs in the construction stage. The FEDomainMPI client objects cannot individually differentiate between their internal and boundary DOFs. It collects $TDOF_0$ by gathering all the partitions $TDOF_j$. The master object's $TDOF_0$ is further classified into $IDOF_0$ and $BDOF_0$ through the user provided Dirichlet DOFs. The user has to provide all the Dirichlet data to the master object. The master object has to compute the solution for the $IDOF_0$ where the $BDOF_0$ solutions are provided by the user. The master object provides $TDOF_0$ to all the client objects so that these can compute their $IDOF_j$ and $BDOF_j$.

In FEDomain direct solver the master object was composed of the client objects and it used to treat all the FEDomain client objects as user provided elements. In DS solver the FEDomainMPI master object is not composed of the client objects but the client objects execute in parallel MPI processes. The FEDomainMPI master object has no contribution in the creation and deletion of the client objects. In FEDomain solver in condensation mode, the master object collects it's client objects Schur complement data through *getSystem()* method declared in the abstract FEElement class. In FEDomainMPI distributed solver the data can only be accessed through the MPI communication protocols.

```
class FEDirectSolverMPI : public FESystem
{
    FEDirectSolverMPI(std::vector<FEElement*>& element_ptr,
                      FE_UINT& sys_total_DOFs,
                      FE_UINT& max_element_DOFs,
                      std::vector<FE_UINT>& part_ids,
                      std::vector<FE_UINT>& dirichlet_IDs,
                      MATRIX_TYPE matrix_type);
    ~FEDirectSolverMPI(void);
    FE_UINT getDofCount(void);
    FE_UINT getPartitionId(void);
    void getLoad(FE_vector& load);
```

```
    void getStiffness(FE_dense_matrix& stiffness);
    void getSystem(FE_equation& equation);
    void getConnectivity(std::vector<FE_UINT>& dof_ids);
    void setSolution(std::vector<FE_DATA>& dirichlet_value);
    void getResidual(std::vector<FE_DATA>& approx_solutions,
                     std::vector<FE_DATA>& residual);
};
```

Listing 5.1: FEDirectSolverMPI Interface

In Chapter 4 the FEDirectSolver and the FEEquation classes are introduced which are the building blocks of the FEDomainMPI solver. The FEDomainMPI solver client objects have the same functionalities as of FEDomain condensation solver client objects. In FEDomainMPI the client objects are implemented using FEDirectSolver and FEEquation classes. The FEDirectSolverMPI class is added as a communication wrapper around the FEDirectSolver class to add capability for MPI based communication. The FEDirectSolverMPI class enables communication among the master and client objects. The interface of the FEDirectSolverMPI class in given in the Listing 5.1.

The FEDirectSolverMPI interface is similar to the FEDirectSolver interface define in the Listing 4.2. The FEDirectSolverMPI class has to behave differently for master object and client objects. In client mode the FEDirectSolverMPI object has to gather all the allocated elements data and compute the Schur complement. This has already been implemented in the FEDirectSovler class in condensation mode. In FEDirectSolverMPI class is composed of the FEDirectSolver object to gather elements data, compute $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ and finally compute the solution of $\{IDOF_j\}$. The FEDirectSolverMPI class in the client mode has to collect $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ from its FEDirectSolver object and transfer it to the master object. The FEDirectSolver interface in the Listing 4.2 provides the *getSystem()*, *getStiffness()* and *getLoad()* to access Schur complement data. The *getStiffness()* takes the dense

matrix container to access $\boldsymbol{S}_{bb}^{P_j}$ which will be wastage of memory. The *getSystem()* is a suitable option to access partitions Schur complement data. It will require the FEDirectSolverMPI object to have a FEEquation object to collect $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ from the FEDirectSovler object.

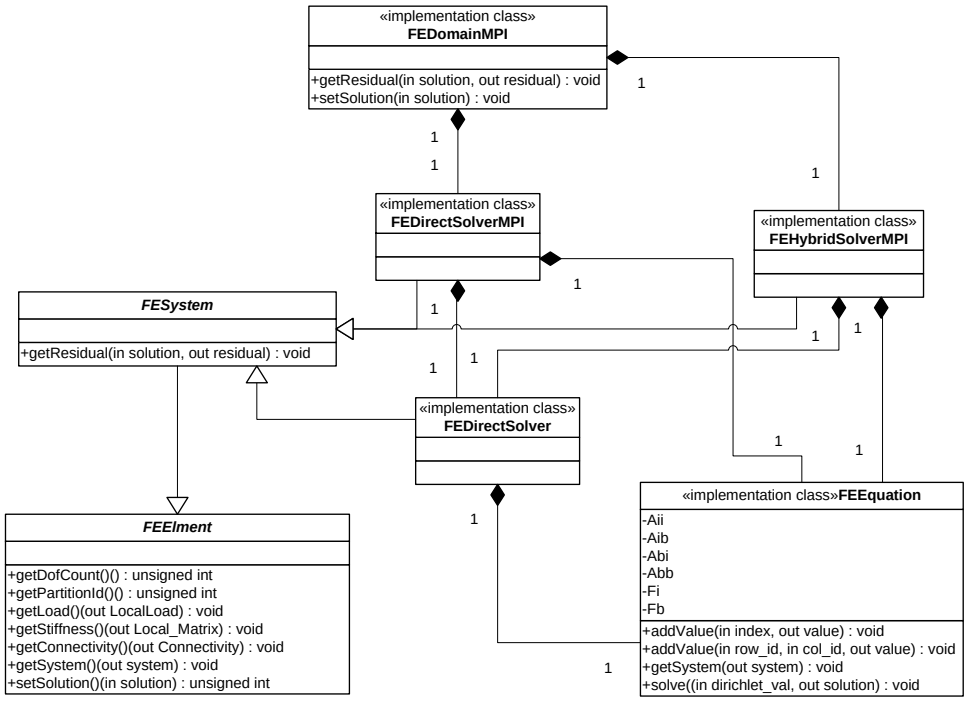



Figure 5.1: FEDomainMPI class diagram.

The FEDirectSolverMPI master object is responsible to compute the solution of the $\{IDOF_0\}$. It collects all its client objects Schur complement data ($\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$) through MPI library. All the client objects keep their Schur complement data into their FEEquation objects. These objects cannot be directly transferred to the mater object due to limitation in MPI library. All the FEDirectSolverMPI objects have to transform their Schur complement data into MPI supported format. The MPI library allows consecutive memory containers to be transferred as a single message. The client's $\vec{v}_b^{P}$ is copied into std::vector and $\boldsymbol{S}_{bb}^{P}$ is transformed into compressed sparse row format (CSR) which is composed of three consecutive arrays

and defined in chapter 4. All the FEDirectSolverMPI client objects access their FEDirectSolver object's $\boldsymbol{S}_{bb}^{P_j}$ and $v_b^{P_j}$ through *FESystem(FEEquation)* by passing their FEEquation object to their FEDirectSolver object. The FEDirectSovlerMPI client objects maintain a extra copy of their allocated elements Schur complement data into their FEEquation objects which further convert its data into the CSR format. The FEDirectSolverMPI master object uses its FEEquation object to collect all the clients Schur complement data. The FEDirectSolverMPI master object does not have to gather data from the user elements. It does not require FEDirectSolver object. The master object's FEEquation object compute the solution of $\{IDOF_0\}$.

### 5.4.1 CSREquation Container

The FEDirectSolverMPI class is composed of FEDirectSolver object, which is responsible for assembling data for systems of equation and compute Schur complement of it. The FEDirectSolver object is further composed of FEEquation object and Schur complement data is persent in it. In shared memory condensation solver implementation given in Section 4.2 the FEEuation object is provide by the master object to collect $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ for all partitions.

In DS the master object is not composed of the client objects so it is not possible to access data directly from the client objects. In the current implementation the FEDirectSolverMPI object has to keep a FEEquation object which has to be passed to its FEDirectSolver's FEEquation object to retrieve $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$. The outer FEEquation object which lies in FEDirectSolverMPI object has to convert the data into MPI compatible CSR format. The CSR format has already discussed in Section 4.1. This design creates three instances of same $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ on single MPI process. The Schur complement data is originally calculated and store in the FEDirectSolver's FEEquation object, the first copy is created by copying data into FEDirectSolverMPI's FEEquation object and the second copy is created when the

data is converted into CSR before transmitting to master object.

In above mentioned implementation the $\boldsymbol{S}_{bb}^{P_j}$ has to be stored three times before transferring to master object. Originally data is saved in the FEDirectSolver's FEEquation object and it does not store $\boldsymbol{S}_{bb}^{P_j}$ in CSR format and has no interface method to provide it in CSR format. The FEDirectSolverMPI object has to get $\boldsymbol{S}_{bb}^{P_j}$ in FEEquation from the FEDirectSolver object and transform it into CSR format. An extra copying of $\boldsymbol{S}_{bb}^{P_j}$ can be avoided if the FEDirectSovler class and FEEquation class have the capability to provide $\boldsymbol{S}_{bb}^{P_j}$ in CSR format. The CSREquation container class is added into FEDomain package as shown in Figure 5.2. A new getSystem(CSREquation&) method is introduced in FEEquation and FEDirectSolver class so that FEDirectSolverMPI object can retrieve $\boldsymbol{S}_{bb}^{P_j}$ and $\vec{v}_b^{P_j}$ in CSR format.

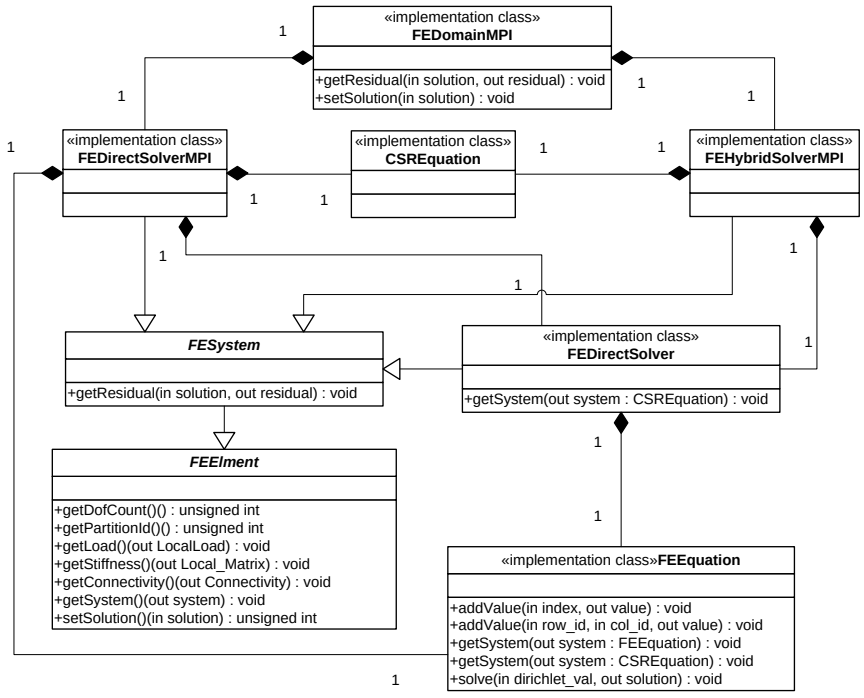



Figure 5.2: FEDomainMPI class diagram with CSREquation object.

Figure 5.2 is the class diagram which shows the static view of the FEDomainMPI solver. The FEDomainMPI is the facade class which hides subsystems Distributed Direct Solver (FEDirectSovlverMPI) and Distributed Hybrid Solver (FEHybrid-SolverMPI). The FEHybridSolverMPI will be discussed in Section 5.5. Both of these classes is composed of FEDirectSolver object, CSREquation object and FEE-quation object. In distribute direct solver the FEEquation object is used in master object to gather all partitions Schur complement data. FEEquation object is not used in client objects. On other had CSREquation object is used in client object while not in master object.

### 5.4.2 Distributed Direct Solver Mathematical Model

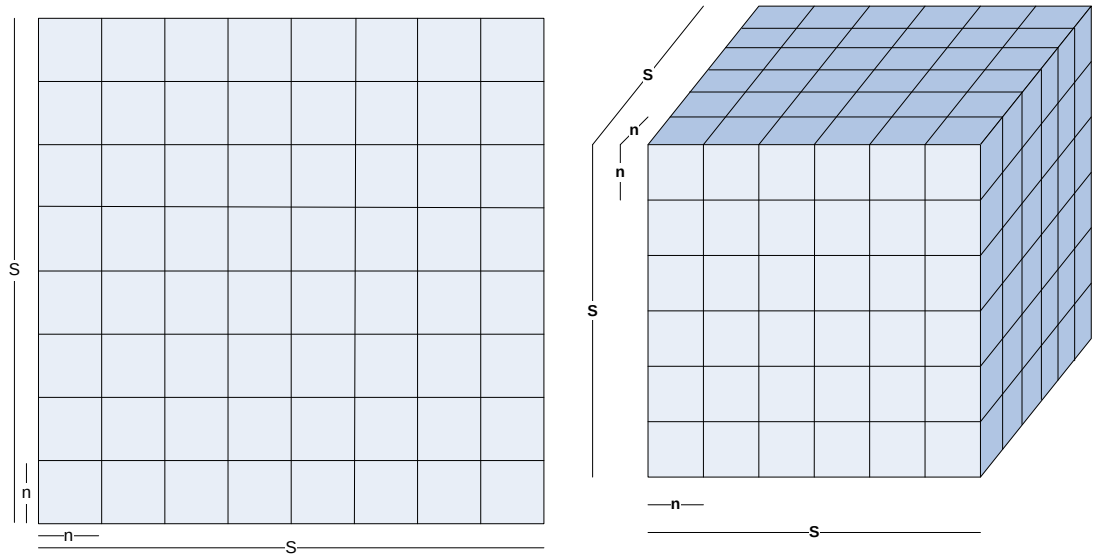
The complexity of the direct solver depends on the complexity of the PARDISO solver. The complexities of PARDISO solver for 2D and 3D meshes are shown in Table 4.9 which are taken from [36]. The $\boldsymbol{S}_{bb}^{P}$ which is defined in (2.71) is the most computationally intensive task which involves multiplication of two sparse matrices. It involves the part of a linear system with matrix $\boldsymbol{A}_{ii}^{P}$, and multiple

right hand side (see chapter 4 for a description).

```
1  C ← getConsecutiveNonEmptyCols(A_ib)
2  ids ← getConsecutiveNonEmptyColsIDs(A_ib)
3  while !Empty(C) do
4  |  T_1 ← L^P U^P C
5  |  T_2 ← A^P_bi T_1
6  |  foreach column i in T_2 do
7  |  |  A^P_bb,:ids[i] −= T_2i
8  |  end
9  |  C ← getNextConsecutiveNonEmptyCols(A_ib)
10 |  ids ← getNextConsecutiveNonEmptyColsIDs(A_ib)
11 end
```

**Algorithm 3:** Sbb algorithm

In Algorithm 3 line 4 calculates $(\boldsymbol{A}_{ii}^{P})^{-1}\boldsymbol{A}_{ib}^{P}$, the consecutive non empty columns are copied into $\boldsymbol{C}$ from $\boldsymbol{A}_{ib}^{P}$ and are provided to the PARDISO solver which return solutions as $\boldsymbol{T}$. Let $I_P$ be the number of internal DOFs and $B_P$ be the number of interface DOFs of the partition $P$. The PARDISO solver takes multiple right hand sides as a consecutive memory arrays $\boldsymbol{C}$ of size $I_P \times \#ids$ and provides the solution in a dense matrix $\boldsymbol{T}_1$. The $\boldsymbol{T}_2$ is a sparse matrix which stores the product of $\boldsymbol{A}_{bi}^{P}\boldsymbol{T}_1$. To control the memory consumption of the solver the size of maximum non empty consecutive columns are limited to *step_size*. The *step_size* has to be tuned on different hardware architectures. The memory consumption in $\boldsymbol{S}_{bb}^{P}$ is directly proportional to the *step_size* value as it affects the size of $\boldsymbol{C}$, $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$. The comparison of values of *step_size* and the time to compute the solution of 2048 right hand sides is given in Table 4.5. The best time is achieved by providing 256 columns to PARDISO solver in single solve call.




(a) 2D Square Mesh. (b) 3D Cube Mesh.

Figure 5.3: Pictures of meshes

#### 5.4.2.1 2-Dimensional Mesh

Let a 2D square mesh in Figure 5.3a is partitioned into $S$ number of partitions in each dimensions. Each partition has $n$ DOFs in each dimensions. The total partitions $P$ are

$$P = S^2$$

and the total DOFs, $N$, in the mesh are

$$N = (nS)^2$$

and

$$n = (N/P)^{1/2}$$

each partition's interface DOFs $B_P$ are

$$B_P = 4n \approx n \approx (N/P)^{1/2}$$

each partition's internal DOFs $I_P$ are

$$I_P = n^2 - 4n \approx n^2 \approx N/P$$

The complexity of PARDISO solver symbolic factorization is $\mathcal{O}(N/P)$, numerical factorization is $\mathcal{O}(N/P)^{3/2}$ and triangular solve complexity $\mathcal{O}(N/P)$. For the 2 dimension mesh the complexity of $\boldsymbol{S}_{bb}^P$ Algorithm 3 has to be calculated for every line. The line 4 complexity for $B_P$ right hand sides is $\mathcal{O}(I_P B_P) = \mathcal{O}((N/P)^{3/2})$, the line 5 complexity is $\mathcal{O}(I_P B_P^2) = \mathcal{O}((N/P)^2)$, and finally for line 6-8 complexity is $\mathcal{O}(B_P^2) = \mathcal{O}(N/P)$. So the total complexity of calculating $\boldsymbol{S}_{bb}^P$ for 2D is

$$\begin{aligned} &= \mathcal{O}((N/P)^{3/2} + (I_P \log I_P)B_P + I_P B_P^2 + B_P^2), \\ &= \mathcal{O}((N/P)^{3/2} + (N/P)^{3/2} + (N/P)^2 + N/P), \\ &= \mathcal{O}((N/P)^2). \end{aligned}$$

Now, for the total complexity, the total interface DOFs $B$ in mesh is:

$$B = S^2 n = P(N/P)^{1/2} = (NP)^{1/2}.$$

The complexity of the computation of $\alpha_B$ is:

$$\mathcal{O}(B^{3/2}) = \mathcal{O}((NP)^{3/4}). \tag{5.1}$$

Hence, complexity of Distributed Direct Solver for 2D mesh is:

$$= \mathcal{O}((N/P)^2) + \mathcal{O}((NP)^{3/4}). \tag{5.2}$$

#### 5.4.2.2 3-Dimensional Mesh

Let a 3D cube mesh shown in Figure 5.3b is partitioned into $S$ partitions as in each dimension. The total number of partitions $P$ in cube is:

$$P = S^3,$$

and the total number of DOFs $N$ in the cube is equal to

$$N = (nS)^3 = n^3 P,$$

and

$$n = \left(\frac{N}{P}\right)^{1/3}.$$

The cube partitions share faces with each other and the DOFs which lies on any of the partition face will be interface DOFs. For any partition in a Figure 5.3b the number of interface DOFs are denoted as $B_P$.

$$B_P = 6((n+1)^2) - 12N - 8 \approx (n)^2 \approx (N/P)^{2/3},$$

each partition's internal DOFs $I_P$ are:

$$I_P = n^3 - n^2 \approx n^3 \approx N/P.$$

For 3D meshes complexity of PARDISO solver symbolic factorization is $\mathcal{O}((N/P)^{4/3})$, numerical factorization is $\mathcal{O}((N/P)^2)$, and triangular solve complexity is $\mathcal{O}((N/P)^{4/3})$. The complexity of $\boldsymbol{S}_{bb}^P$ Algorithm 3 has to be calculated for every line. The line 4 complexity for $B_P$ right hand sides is $\mathcal{O}({I_P}^{4/3} B_P) = \mathcal{O}((N/P)^2)$, the line 5 complexity is $\mathcal{O}(I_P B_P^2) = \mathcal{O}((N/P)^{7/3})$, and finally for line 6-8 complexity is $\mathcal{O}(B_P^2) = \mathcal{O}(N/P)^{4/3}$. So the total complexity of calculating the $\boldsymbol{S}_{bb}^P$ in 3D mesh

is

$$\begin{aligned} &= \mathcal{O}(I_P^2 + I_P^{4/3} B_P + I_P B_P^2 + B_P^2), \\ &= \mathcal{O}((N/P)^{2/3} + (N/P)^2 + (N/P)^{7/3} + (N/P)^{4/3}), \\ &\approx \mathcal{O}((N/P)^{7/3}). \end{aligned}$$

Now in 3D cube mesh, the total interface DOFs $B$ is:

$$B = s^3 n = P(N/P)^{1/3} = N^{1/3} P^{2/3},$$

The complexity of computation of $\alpha_B$ is:

$$\mathcal{O}(B^2) = \mathcal{O}(N^{2/3} P^{4/3}).$$

Hence, the total complexity of the Distributed Direct Solver for 3D mesh is:

$$= \mathcal{O}((N/P)^{7/3}) + \mathcal{O}(N^{2/3} P^{4/3}).$$

#### 5.4.2.3 D-Dimensional Mesh

The total number of partitions in mesh is

$$P = S^D.$$

Total number of DOFs $N$ in cube is equal to

$$N = (nS)^D = n^D P$$

so $n$ becomes

$$n = (N/P)^{1/D}$$

The number of boundary DOFs in a partition of D dimensional mesh is

$$B_P \approx (n)^{D-1} \approx (N/P)^{(D-1)/D},$$

than number of internal DOFs $I_P$ for a partition is equal to

$$I_P = n^D - n^{D-1} \approx n^D \approx N/P.$$

The complexity of computing $\boldsymbol{S}_{bb}^P$ is

$$\mathcal{O}((P/N)^{(3D-2)/D}). \tag{5.3}$$

Now the total interface DOFs $B$ in the mesh is

$$B = s^D n = P(N/P)^{1/D} = N^{1/D} P^{(D-1)/D}.$$

than the time complexity of $\alpha_B$ computation is

$$\mathcal{O}(B^2) \quad = \quad \mathcal{O}((N^{1/D} P^{(D-1)/D})^2). \tag{5.4}$$

Hence, the total time complexity of the Distributed Direct Solver for D dimensional mesh is

$$= \mathcal{O}((P/N)^{(3D-2)/D}) + \mathcal{O}(N^{2/D} P^{2(D-1)/D}). \tag{5.5}$$

## 5.5 Distributive Hybrid Solver

The DS solver allowed user to solver large systems of linear equations on distributed memory systems. The $\boldsymbol{S}_{bb}$ is a dense matrix, its dimension is smaller than the dimension of $\boldsymbol{A}$ but the amount of data stored in $\boldsymbol{S}_{bb}$ is more than $\boldsymbol{A}$. The total number of non zero entries in $\boldsymbol{A}$ for CUBE6 mesh is 16371949 and the

| CUBE6 (3D Elasticity) A_NNZ=16371929 | | | | |
|---|---|---|---|---|
| Partitions | CON_I_DOF | $\boldsymbol{S}_{bb_{ii}}$ NNZ Data | SMMM | SMVV |
| 2 | 8886 | 39484725 | 14.45 | 162.50 |
| 3 | 14649 | 106973378 | 29.39 | 443.13 |
| 4 | 18267 | 136116856 | 35.93 | 548.88 |
| 5 | 21816 | 160683485 | 41.24 | 625.40 |
| 6 | 23982 | 170170868 | 42.93 | 613.95 |
| 7 | 26055 | 171712080 | 43.13 | 580.08 |

Table 5.2: The table represents timing to collect CUBE6 $\boldsymbol{S}_{bb}$ into SMMM and SMVV continers. Interface DOFs increase with the increase in partition count. The amount of non zero data in $\boldsymbol{S}_{bb}$ also increases with the raise in partition count.

number of non zero entries in $\boldsymbol{S}_{bb}$ for CUBE6 meshes divided into multiple partitions are shown in Table 5.2. With the increase in number of partitions, the number of interface DOFs increases and so does the amount of non-zero entries in $\boldsymbol{S}_{bb}$. In FEDirectSolverMPI the $\boldsymbol{S}_{bb}$ is contained in SMMM container which is composed of binary tree. On Linux OS the STL std::map container does not release acquired memory after the map object is deleted. This method is used to maintain efficiency and avoid reallocation of map memory until it is absolutely necessary. The fourth and fifth columns in Table 5.2 represent the time consumed to add the number of non zero entries into $\boldsymbol{S}_{bb}$ for SMMM and SMVV containers, respectively. The SMVV container is composed of std::vector and as the SMVV object is destroyed it returns all its allocated memory to OS. The SMVV is a memory efficient sparse matrix container which stores only column id and value for each entry. This container is not suitable for the densely populated matrix with random order data entry. The SMMM is very efficient data container which has data insert and lookup complexity is $\mathcal{O}(logn)$[71]. The map is a binary tree which atleast store 3 pointers (left child, right and parent nodes) for each data entry (index and value). It consumes at least three times more data than SMVV and also does not release all the memory when the container is destroyed.

A new distributed solver is introduced into the FEDomain package called Distributed Hybrid Solver (DHS). In DHS solver the "Conjugate Gradient" iterative method is used to calculate $\alpha_B$. The iterative solver calculates the solution by the computing residual vector in each iteration. The residual vector is calculated in parallel as shown in (2.78). To keep the distributed direct and iterative solver is alike, the $\alpha_B$ are calculated in the master process. In iterative solver $\alpha_B$ is computed by multiplying each partition's $\boldsymbol{S}_{bb}^P$ by the approximate global solution $d$ to calculate the residual vector $\vec{r}^P$. These are added together to calculate $\vec{r}$. The Conjugate Gradient (CG) solver is implemented in FEDomainMPI and its algorithms is a modified version of the algorithm given in [13] on p.23. The algorithm is modified to be implemented on distributed memory system. The detailed of the implemented algorithm is given in Algorithm 4.

```
1  g = Σ_{i=0}^{i<N} b_i
2  δ_0 = g^T g
3  β = 0
4  while NotConverged(δ_0) do
5  |   d = g + βd
6  |   MPI_Bcast(d)
7  |   h = Σ_{P=0}^{P<N} h^P = Σ_{P=0}^{P<N} ((Λ_b^P)^T S_bb^P) d
8  |   MPI_Reduce(h)
9  |   τ = δ_0/(d^T h)
10 |   x = x + τd
11 |   g = g + τh
12 |   δ_1 = g^T g
13 |   β = δ_1/δ_0
14 |   δ_0 = δ_1
15 |   MPI_Bcast(δ_0)
16 end
```

**Algorithm 4:** Conjugate Gradient Parallel Algorithm.6

The lines 1, 5 and 7 in Algorithm 4 have to be performed by all the client processes. In line 1 all the non Dirichlet interface DOFs load values are added together for all the processes and stored in the master process. In line 4 the value

of $d$ is computed and spread to all the clients. The $d$ is an approximate solution vector which is distributed to all the client process to calculate their residual vector $h^P$. For the first iteration $d$ is equal to $g$. All the client's residual vector $h^P$ are summed on master object to obtain $h$. The master object performs the rest of the steps in the algorithm while the other process wait for $d$. At the end of each iteration all the client objects receive the convergence information $\delta_0$ from the master object.

In FEDomain package DHS is implemented as FEHybridSolverMPI class. The DHS is selected by setting the DISTRIBUTED_SOLUTION_METHOD equal to DIS_HYBRID_SOLVER. The FEDomainMPI interface is the same for both direct and hybrid solvers, and the user can switch between these solvers by only changing a single parameter in FEDomainMPI constructor. In DHS, each partition is allocated to a single client MPI process or vice versa. The master process is fixed for iterator solver and no partition is allocated to it. The DHS processes communicate with each other through MPI and perform internal calculations in parallel using OpenMP. The matrix vector multiplication in line 7 of Algorithm 4 is performed by client processes FEHybridSolverMPI objects. The DHS objects store $\boldsymbol{S}_{bb}^{P_j}$ for allocated partition $\mathcal{P}_j$ in a memory efficient CSR format which is a set of std::vectors which allows fast memory access as well as requires minimum amount of memory to store data.

| CB3_2205745 (3D Elasticity) DOFs=6617235 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| MPI | OMP | Mesh | Interface DOFs | | | Conjugate Gradient | | Solution |
| Process | Thread | Partitions | Internal | Dirichlet | $\boldsymbol{S}_{bb}^{P}$ | Iterations | Time | Time |
| 192 | 1 | 191 | 633963 | 30234 | 375.014 | 2052 | 1057.18 | 1331.97 |
| 96 | 2 | 95 | 486843 | 22317 | 672.141 | 2353 | 1857.68 | 2278.23 |
| 48 | 4 | 47 | 356334 | 16233 | 1562.80 | 1382 | 1545.59 | 2856.83 |
| 24 | 8 | 23 | 255321 | 11799 | 3673.94 | 1160 | 1931.52 | 5688.76 |
| 23 | 8 | 22 | 249747 | 11601 | 3794.24 | 911 | 1672.06 | 5549.42 |
| 22 | 8 | 21 | 237885 | 11409 | 4310.35 | 1068 | 1763.82 | 6163.64 |
| 21 | 8 | 20 | 236718 | 10731 | 4297.23 | 1131 | 2173.39 | 6570.60 |
| 20 | 8 | 19 | 230391 | 10257 | 5504.83 | 1881 | 3396.74 | 9001.51 |
| 19 | 8 | 18 | 214725 | 10320 | 5294.10 | 1827 | 3401.25 | 8813.33 |
| 18 | 8 | 17 | 218469 | 10212 | 6516.16 | 943 | 2707.64 | 9513.44 |
| 17 | 8 | 16 | 215121 | 9825 | 6370.11 | 1250 | 2940.29 | 9554.00 |

Table 5.3: DHS solver timing solution timing for CB3_2205745 mesh for partitions 16 to 23,47,95 and 191.

| CB3_2205745 (3D Elasticity) DOFs=6617235 | | | | | |
|---|---|---|---|---|---|
| MPI | OMP | Mesh | | Conjugate | FEDomain |
| Process | Thread | Partitions | $\boldsymbol{S}_{bb}^{P}$ | Gradient | Solution |
| 192 | 1 | 191 | 16.99 | 2.78 | 7.17 |
| 96 | 2 | 95 | 9.48 | 1.58 | 4.19 |
| 48 | 4 | 47 | 4.08 | 1.90 | 3.34 |
| 24 | 8 | 23 | 1.73 | 1.52 | 1.68 |
| 23 | 8 | 22 | 1.68 | 1.76 | 1.72 |
| 22 | 8 | 21 | 1.48 | 1.67 | 1.55 |
| 21 | 8 | 20 | 1.48 | 1.35 | 1.45 |
| 20 | 8 | 19 | 1.16 | 0.87 | 1.06 |
| 19 | 8 | 18 | 1.20 | 0.86 | 1.08 |
| 18 | 8 | 17 | 0.98 | 1.09 | 1.00 |
| 17 | 8 | 16 | 1.00 | 1.00 | 1.00 |

Table 5.4: This is a speedup graph of the DHS solver for CB3_2205745 mesh. This graph shows the relative speedup graph as the solution was not possible for the 15 partition mesh. The speed up is calculated relative to 16 partitions execution timing.

| CB3_2205745 (3D Elasticity) DOFs=6617235 | | | |
|---|---|---|---|
| MPI | OMP | Internal | Interface |
| 192 | 1 | 29004 | 11313 |
| 96 | 2 | 61566 | 16722 |
| 48 | 4 | 126192 | 28416 |
| 24 | 8 | 261438 | 44163 |
| CB2_1157354 (3D Elasticity) DOFs=3472131 | | | |
| 192 | 1 | 14970 | 6882 |
| 96 | 2 | 31365 | 10956 |
| 48 | 4 | 65532 | 17649 |
| 24 | 8 | 135993 | 28890 |

Table 5.5: Mesh partitions internal and interface DOFs for the biggest partition.

FEDomain DHS is a parallel implementation of Conjugate Gradient method shown in Algorithm 4 which does not require assembling of $\boldsymbol{S}_{bb}$ on the master node. We now show the performance in the case of a 3D elasticity problem in the mesh CB3_2205745. The CB3_2205745 mesh has 6,617,235 DOFs. The FEDomain DS was not able to compute the solution of CB3_2205745 due to the shortage of memory required to accommodate $\boldsymbol{S}_{bb}$ on the master node. The DHS enabled to solve the CB3_2205745 mesh on the same machine, but while using more than 15 computational nodes with 8 OpenMP threads. Meanwhile, for CB3_2205745 mesh with less than 15 partitions due to lack of the available memory required by PARDISO to compute $\boldsymbol{S}_{bb}^{P}$, the applications were aborted abnormally. Table 5.3 represents the timing of the CB3_2205745 mesh using different number of MPI process and OpenMP threads. In each case each MPI process is allocated a partition except for the master process. Each MPI process internal computations are executed in parallel using OpenMP threads. Table 5.3 provides the timing for different stages of the DHS solver. The $\boldsymbol{S}_{bb}^{P}$ column represents the time taken for computing the $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_{b}^{P_j}$ of each partition. The timing value in the column is the longest time taken among the client processes. The conjugate gradient columns represent the number of iterations and time taken by the iteration solver to com-

pute the solution of $IDOF_0$. The last column represents the solution time taken by the DHS solver. It includes construction time, Schur's Complement time, and Conjugate Gradient time. The first row represents the MPI processes where each has sequential code in it. The second row represents the combination of MPI processes and OpenMP threads as in each MPI process computation is distributed between two OpenMP threads. The total partitions are reduced to half as compare to first row. As the number of partitions reduces so does the mesh total interface DOFs but on other hand the number of interface and internal DOFs on each partition increases. The increase in the interface DOFs count increases the computational time for each partition's Schur Complement as shown in Table 5.5. Table 5.4 is the speed up table. The DHS has achieved maximum speed up of 17 for the $\boldsymbol{S}_{bb}^{P}$ calculation and 7.17 speed up for the total execution time. The total speed up has reduced due to the Conjugate Gradient solver methods. The number of TDOFs increase with the number of partitions.

| CB2_1157354 (3D Elasticity) DOFs=3472131 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| MPI | OMP | Mesh | Interface DOFs | | | Conjugate Gradient | | Solution |
| Process | Thread | Partitions | Internal | Dirichlet | $\boldsymbol{S}_{bb}^{P}$ | Iterations | Time | Time |
| 192 | 1 | 191 | 434760 | 20049 | 133.364 | 1067 | 295.326 | 370.179 |
| 96 | 2 | 95 | 335061 | 15129 | 381.786 | 1002 | 421.115 | 690.428 |
| 48 | 4 | 47 | 247515 | 11157 | 577.497 | 856 | 609.054 | 1305.27 |
| 24 | 8 | 23 | 186030 | 8799 | 1311.99 | 744 | 827.42 | 2171.85 |
| 23 | 8 | 22 | 176631 | 8490 | 1244.39 | 757 | 701.50 | 1977.21 |
| 22 | 8 | 21 | 176175 | 8325 | 1524.53 | 724 | 746.22 | 2313.68 |
| 21 | 8 | 20 | 169110 | 7962 | 1652.87 | 779 | 848.97 | 2544.15 |
| 20 | 8 | 19 | 163437 | 7743 | 2321.81 | 710 | 883.42 | 3261.63 |
| 19 | 8 | 18 | 158199 | 7395 | 1894.88 | 774 | 1037.08 | 2968.63 |
| 18 | 8 | 17 | 153309 | 7059 | 2195.67 | 779 | 997.33 | 3242.75 |
| 17 | 8 | 16 | 146265 | 6741 | 2096.10 | 707 | 813.67 | 2965.78 |
| 16 | 8 | 15 | 142086 | 6702 | 2330.80 | 677 | 977.66 | 3358.46 |
| 15 | 8 | 14 | 142248 | 6411 | 2716.23 | 713 | 1168.94 | 3953.82 |
| 14 | 8 | 13 | 128745 | 5928 | 3073.86 | 764 | 1139.70 | 4282.56 |
| 13 | 8 | 12 | 124263 | 5502 | 3454.59 | 691 | 1054.07 | 4600.39 |
| 12 | 8 | 11 | 115584 | 5385 | 4055.93 | 643 | 1126.87 | 5287.67 |
| 11 | 8 | 10 | 115392 | 5163 | 5089.73 | 770 | 1685.30 | 6918.78 |
| 10 | 8 | 9 | 109299 | 4851 | 4688.47 | 650 | 1147.04 | 5965.95 |
| 9 | 8 | 8 | 97059 | 4122 | 5843.30 | 775 | 1507.39 | 7495.50 |
| 8 | 8 | 7 | 88377 | 3840 | 6700.81 | 510 | 731.81 | 7590.46 |
| 7 | 8 | 6 | 81801 | 3783 | 8147.06 | 553 | 907.70 | 9409.74 |
| 6 | 8 | 5 | 73593 | 3201 | 11594.1 | 665 | 1380.11 | 17423.9 |

Table 5.6: DHS iterative solver timing for CB2_1157354 mesh with DOFs=3472131 for partitions 5 to 23,47,95 and 191. The mesh is used to solve the Elasticity 3D problem.

| CB2_1157354 (3D Elasticity) DOFs=3472131 | | | | | |
|---|---|---|---|---|---|
| MPI Process | OMP Thread | Mesh Partitions | $\boldsymbol{S}_{bb}^{P}$ | Conjugate Gradient | FEDomain Solution |
| 192 | 1 | 191 | 86.94 | 4.67 | 47.07 |
| 96 | 2 | 95 | 30.37 | 3.28 | 25.24 |
| 48 | 4 | 47 | 20.08 | 2.27 | 13.35 |
| 24 | 8 | 23 | 8.84 | 1.67 | 8.02 |
| 23 | 8 | 22 | 9.32 | 1.97 | 8.81 |
| 22 | 8 | 21 | 7.61 | 1.85 | 7.53 |
| 21 | 8 | 20 | 7.01 | 1.63 | 6.85 |
| 20 | 8 | 19 | 4.99 | 1.56 | 5.34 |
| 19 | 8 | 18 | 6.12 | 1.33 | 5.87 |
| 18 | 8 | 17 | 5.28 | 1.38 | 5.37 |
| 17 | 8 | 16 | 5.53 | 1.70 | 5.87 |
| 16 | 8 | 15 | 4.97 | 1.41 | 5.19 |
| 15 | 8 | 14 | 4.27 | 1.18 | 4.41 |
| 14 | 8 | 13 | 3.77 | 1.21 | 4.07 |
| 13 | 8 | 12 | 3.36 | 1.31 | 3.79 |
| 12 | 8 | 11 | 2.86 | 1.22 | 3.30 |
| 11 | 8 | 10 | 2.28 | 0.82 | 2.52 |
| 10 | 8 | 9 | 2.47 | 1.20 | 2.92 |
| 9 | 8 | 8 | 1.98 | 0.92 | 2.32 |
| 8 | 8 | 7 | 1.73 | 1.89 | 2.30 |
| 7 | 8 | 6 | 1.42 | 1.52 | 1.85 |
| 6 | 8 | 5 | 1.00 | 1.00 | 1.00 |

Table 5.7: DHS iterative solver speedup for CB2_1157354 mesh with DOFs=3472131 for partitions 5 to 23,47,95 and 191. The mesh is used to solve the Elasticity 3D problem.

The CB2_1157354 is a 3D mesh with 3472131 DOFs which is triangulated into tetrahedron and triangle elements. FEDomain DHS solver timing information for solving the CB2_1157354 mesh having 5 to 23,47,95 and 191 partitions is given in Table 5.6 and attained relative speedup in Table 5.7. It shows the same timing pattern of both the meshes. The cb2_1157354 and cb3_2205745 meshes cannot be

solved on a single computational node. The cb2_1157354 mesh with less than 5 partitions could not be solved due to shortage of memory required to compute the partitions Schur's complement matrix and vector. The $\boldsymbol{S}_{bb}^{P}$ has achieved 86.84 speeup for 192 processes and Conjugate Gradient algorithm has achieved 4.67 speedup. The DHS solver has achieved speedup of 47.07 in total. The Conjugate Gradient method speedup is not constant as it involved MPI communication and number of iterations are unpredictable.

### 5.5.1 Distributed Hybrid Solver Mathematical Model

In the Conjugate Gradient algorithm the most computationally intensive task is matrix vector multiplication to compute $h$. It has a computational complexity of $\mathcal{O}(B^2)$ for each iteration if it is performed on a processor using $\boldsymbol{S}_{bb}$. In FEDomain DHS matrix vector multiplication is performed in parallel on each partition as shown in Algorithm 4 Line 7. The $\boldsymbol{S}_{bb}^{P}$ matrix is $B_P^2$ dimension matrix which is densely populated so the computational cost of each iteration matrix vector multiplication is $\mathcal{O}(B_P^2)$. The computational complexity of partitions $\boldsymbol{S}_{bb}^{P}$ and $\vec{v}_b^P$ for 2D and 3D meshes are in (5.1) and (5.3) respectively. The computational complexity of FEDomain DHS for 2D mesh is

$$= \mathcal{O}((N/P)^2) + \mathcal{O}(N/P)$$

and for 3D mesh computational complexity is

$$= \mathcal{O}((N/P)^{7/3}) + \mathcal{O}((N/P)^{4/3})$$

and for any arbitrary D dimension mesh computational complexity is

$$= \mathcal{O}((N/P)^{(3D-2)/D}) + \mathcal{O}((N/P)^{2(D-1)/D}).$$

## 5.6 Conclusion

In this chapter we have discussed implementations of distributed memory finite element solvers. The are two distributed memory solvers (direct and hybrid) are implemented in the FEDomain package. These solvers are implemented using domain decomposition methods. The distributed direct solver gathers Schur complement data $\boldsymbol{S}_{bb}$ and $\vec{v}_b$ to compute interface DOFs solution using a third party solver. The $\boldsymbol{S}_{bb}$ is a densely populated matrix and the transfer of data from client objects to master object is an inefficient task. The data count of $\boldsymbol{S}_{bb}$ increases with the number of interface DOFs which is directly proportional to the number of mesh partitions as can be seen in Table 5.1. The table has the upper triangle count of $\boldsymbol{S}_{bb}$ it does not depict the amount of data transferred to master object. The experiment showed that amount of data transferred are much greater than shown in Table 5.1.

The distributed hybrid solver is implemented to overcome the disadvantages of distributed direct solver. In distributed hybrid solver, the interface DOFs solution is computed using Conjugate Gradient solver given in Algorithm 4. This implementation does not require assembly of $\boldsymbol{S}_{bb}$ and $\vec{v}$. The amount of data transferred among the FEDirectSolverMPI objects is small as compare to the distributed direct solver.

# Chapter 6

# FEDomain Residual Methods

The FEDomain package was initially implemented to calculate residual vector $\vec{r}$ for the iterative linear algebra solver algorithms like Jacobi and Conjugate Gradient methods implemented. The residual computation involves matrix vector multiplication in every residual iteration. The residual is a computation intensive task which has to be computed efficiently to reduce the total computation time of the developed application. There are two methods for calculating the residual vector, namely the *Full Assembly* (FA) and *Element by Element* (EBE) methods. The mathematical model of both residual methods has already been discussed in Section 2.2.2. These methods require the same elements data ($\vec{b_k}$,$\boldsymbol{A}_k$ and $\boldsymbol{\Lambda}_k$) but require different internal implementation with regards to storage and calculation algorithms. In FEDomain the residual computation class is added as the FEResidual class which is initially implemented as standalone library for the shared memory machines. This chapter discusses development life cycle for the FEResidual class from very first implementation to the current/latest implementation. During the FEResidual class development, the Jacobi iterative method is used as

linear algebraic solver. The algorithm used is given in Algorithm 5.

$\boldsymbol{D} = diag(a_{ii}^{-1})$;
**while** $\alpha$ `not converged` **do**
  $\vec{r} = \vec{b} - \boldsymbol{A}\vec{\alpha}$;
  $\alpha += \boldsymbol{D}\vec{r}$;
**end**

**Algorithm 5:** Jacobi iterative method used to calculate solution in FEResidual class testing.

## 6.1 FEResidual Version 1

### 6.1.1 Interface

The FEResidual class interface given in Listing 6.1 contains the class definition. The FEResidual class constructor requires the vector of user element pointers and total number of DOFs. The element has two arguments where the first argument is a pointer to a vector of the user elements pointers and second argument is the total number of DOFs in $\mathcal{P}$. The Listing 6.1 has two residual functions. The residual_FA implements the FA residual method and the residual_EBE implements the EBE residual method. The residual functions have same signatures. These methods receive a vector of approximate solution vector $\alpha_g$ and return residual vector $\vec{r}$.

```
template <typename TElement>
class FEResidual{
public:
    FEResidual(std::vector<TElement*> *elements, size_t total_DOFs);
    std::vector<double> residual_FA(const std::vector<double>& X);
    std::vector<double> residual_EBE(const std::vector<double>& X);
};
```

Listing 6.1: FEResidual Interface

The FEResidual class has to collect each elements data ($\vec{b}_K$, $\boldsymbol{A}_K$ and $\boldsymbol{\Lambda}_K$) to calculate the residual. The elements data is collected once at it remains unchanged through the iterations. The user of the FEDomain has to implement their element classes in C++. The FEDomain package has provided definition of the element interface methods in the Listing 6.2. The user has to implement these methods into its C++ element classes to make these compatible with the FEResidual class. The FEResidual is a template class which requires abstract base element class as a template parameter. The user has to implement the abstract base element class and define interface methods in the Listing 6.2 as pure virtual functions. All the user element classes should be inherited from the abstract element class. The standardization of the element interface allows the FEResidual object to gather data from user elements. To carry out this concept we make use of polymorphism.

```
virtual std::vector<double>& getLoad();
virtual gmm::dense_matrix<double>& getStiffness();
virtual gmm::dense_matrix<double>& getConnectivity();
```

Listing 6.2: Element Interface Methods

### 6.1.2 Implementation

In the FEResidual class first version, all the matrix vector manipulations are implemented using third party matrix manipulation library, *GMM++* [3]. The FEResidual object accesses elements data using container references provided by elements objects. FEResidual class includes the GMM++ library and uses to compute the residual vector. The user should have the knowledge about the GMM++ library as element objects have to provide $\boldsymbol{A}_K$ and $\boldsymbol{\Lambda}_K$ in the GMM++ compatible data structures *gmm::dense_matrix*. The element load vector is provided in the $std :: vector < double >$ and it is also compatible with the GMM++. The user elements provide their load vectors and stiffness matrices using local numbering, and connectivity matrices are used to map their data from local DOF numbering

into global DOF numbering. The user interface in the Listing 6.2 requires the user elements to construct and populate these data containers and provide their references to the FEResidual objects for calculations.

### 6.1.3 Drawbacks

The FEResidual class performs all computations using GMM++ matrix manipulation functions like multiplication and addition. The element interface forces the user to implement its element classes using GMM++ or at least provide GMM++ supporting interface in Listing 6.2. The restriction affects previously implemented user element classes which do not support GMM++. These element classes have to be altered to support Listing 6.2. The modifications can be complicated, troublesome and prone to errors in the code. The residual class interface is not generic in nature as it does not support element classes implemented using any data containers. The interface enforces the elements objects to keep a copy of $\vec{b}_K$, $\boldsymbol{A}_K$ and $\boldsymbol{\Lambda}_K$ in specific data containers. It is considered as wastage of memory in some finite elements implementations.

## 6.2 FEResidual Version 2

The FEResidual Version 2 is implemented to overcome the drawbacks for the FEResidual Version 1. The main objective of the FEResidual version 2 is to add support for the user elements classes using the third party data containers. The FEResidual interface is shown in Listing 6.3 where it is templated to elements data types. In FEResidual version 2 the FEResidual class has four template parameters. The first template parameter represents the abstract element class as it was in version 1. The second template parameter represents the one dimensional container used by the elements to return load vector. The third template parameter represents the two dimensional matrix type used as a container in element class to store stiffness and connectivity matrix. The last template parameter type is

also a one dimensional container, which will be used to store the residual vector in residual calculation. It is possible that a user uses different one dimensional data containers in the elements and the residual methods.

### 6.2.1 Interface

The change in FEResidual class interface requires changes in element interface. The abstract element class should have an interface shown in the Listing 6.4. All the user element classes should use the same set of data structures, which are represented by the template parameters TE_Vector and TE_Matrix. The interfaces in the Listings 6.3 and 6.4 keep element class modifications.

```
template<typename TElement, typename TE_Vector,
typename TE_Matrix, typename TG_Vector>
class FEResidual{
public:
    FEResidual(std::vector<TElement*> *element, size_t total_DOFs);
    TG_Vector residual_FA(const TG_Vector& X);
    TG_Vector residual_EBE(const TG_Vector& X);
};
```

Listing 6.3: FEResidual version 2 interface

```
class Element{
    virtual TE_Vector& getLoad()=0;
    virtual TE_Matrix& getStiffness()=0;
    virtual TE_Matrix& getConnectivity()=0;
};
```

Listing 6.4: Generic Element Interface

#### 6.2.1.1 Requirements of Template Parameters

The interface in Listing 6.3 allows a user to template FEResidual with any set of element classes. All the user element classes should be using the same data con-

tainers. All the user selected data containers cannot provide similar signatures for their set and get data functions. The FEResidual class cannot provide support for all the user provided data containers get function and set function signatures. The FEResidual library requires from the users to provide data containers which have specified get function and set function interfaces. The TG_Vector and TE_Vector represent one dimensional data containers which should provide [ ] subscript operator interface to get and set data. The operator should be able to get and set data from and to a specific location in the container represented by an index. The container should also have a *size()* member function to return the dimension of the container. The one dimensional containers used should support an interface in Listing 6.5. The first subscript operator interface in Listing 6.5 is used to access the reference of the memory location needed to update the value. The second subscript interface return the copy of the stored value at the location. The TE_Matrix interface is already discussed in Section 3.1. The round brackets operators ( , ) is used in TE_Matrix interface to set and get data.

```
class Vector{
public:
    Vector(size_t size);
    size_t size() const;
    double& operator[] (size_t index);
    double operator[](size_t index) const;
};
```

Listing 6.5: Vector Interface

### 6.2.2 Implementation

In FEResidual Version 1, the GMM++ library methods were used for all the internal computations. The FEResidual version 2 is designed to support the element classes with the generic data containers. A computation kernel FEMath is implemented in the FEResidual class to perform internal manipulation. It is

added to provide support to the generic data containers. The FEMath contains template methods to perform matrix and vector manipulations such as addition, multiplication, and transpose. Few examples of these methods interfaces are given in Listing 6.6.

```
//Matrix A. and Vector B, C.
//C = A * B.
template <typename TM, typename TV1, typename TV2>
void mult_matrix_vector(const TM& A, const TV1& B, TV2& C);

//Matrix A. and Vector B, C.
//C += A * B.
template <typename TM, typename TV1, typename TV2>
void mult_add_matrix_vector(const TM& A, const TV1& B, TV2& C);
```

Listing 6.6: FEMath Interfaces

The FA method requires $\boldsymbol{A}$ and $\vec{b}$ for the residual calculation. For the linear system of equations, $\boldsymbol{A}$ and $\vec{b}$ have to be constructed once. The construction of $\boldsymbol{A}$ and $\vec{b}$ from the element's data is a time intensive task. The FEResidual class constructs and stores $\boldsymbol{A}$ and $\vec{b}$ before the first residual iteration. These objects remain for the life time of the FEResidual object. Unlike FA method in EBE residual method the residual is calculated at the element level. In a residual iteration, each element data is accessed and its residual vector $r_K$ is computed. All the elements residual vectors are mapped into global residual vector $r = \sum \boldsymbol{\Lambda}_K^T r_K$. This method does not require to store elements data in FEResidual object. These are constructed and provided by the user element objects. These can be accessed by reference for each iteration manipulation. These residual methods can be called in any order. The FA method requires more memory then the EBE method since it needs $s\boldsymbol{A}$ and $\vec{b}$. The FEResidual class requires user element objects to provide their data after applying all the constraints.

### 6.2.3 Performance

Table 6.1 represents the performance of the FEResidual Version 2 for the Poisson 2D problem using both methods. The FEResidual Version 2 is executed for multiple meshes.

| DOFs | Elements | Iter. | Method | Const Time | Convgs Time | Per Iter Time |
|---|---|---|---|---|---|---|
| 239 | 479 | 1705 | FA | 6.178e-05 | 1.524 | 8.941e-04 |
| 239 | 479 | 1705 | EBE | 6.158e-05 | 30.20 | 1.771e-02 |
| 956 | 1913 | 6090 | FA | 2.992e-03 | 102.31 | 1.6799-02 |
| 956 | 1913 | 6090 | EBE | 3.189e-03 | 1800.23 | 2.956-01 |
| 3741 | 7483 | 20511 | FA | 4.855e-02 | 5682.47 | 2.77-01 |
| 3741 | 7483 | - | EBE | 4.917e-02 | - | - |

Table 6.1: FEResidual V2 Timing Table

Table 6.1 shows the timing information of this library using both residual methods. The total number of iterations and the constructor timing are similar for both residual methods. The convergence timing of residual methods have huge timing difference due to the generic interface. In FA method, the element data is accessed only once before the first iteration to generate the global stiffness matrix and load vector. The FEResidual object store these in an internally implemented data containers. These containers enable to implement matrix vector multiplication efficiently. The EBE residual for an iteration, accesses each element data and processes these through FEMath kernel. The FEMath does not have the knowledge about the element class data container algorithms. This results in a poor performance for the residual calculation as shown in Table 6.1. The most time consuming task is multiplication with $\mathbf{\Lambda}_K$. In each row mostly a single non zero entry exists. The FEMath access data from all the data indexes during multiplication.

### 6.2.4 Drawbacks

The manipulation and storage algorithms for sparse matrix data heavily affects the performance of the a finite element software. The idea of using generic third party matrix and vector containers appeared to be the right approach to design a library. These have improved memory consumption but have adversely affected the timing and performance. In object oriented programming, the data container classes encapsulate the data which can only be accessed via class public interface methods. The use of third party data container classes refrains the FEMath kernel to directly access their data and take the advantage of their internal data storage algorithms. It does not have any knowledge of the container class data storage algorithm. In FEResidual class all the calculations are performed using the FEMath computation kernel. The FEMath kernel can access the container data through container class public interface. The element connectivity matrix is sparse in nature where for an element its dimensions are $N_K \times N$ (where $N_K \ll N$). The FEMath kernel is ignorant of the connectivity matrix storage structure and treats it as a dense matrix. During matrix computation the FEMath kernel will access every zero and non-zero entity of the sparse matrix. This method makes the whole computation very inefficient by performing unnecessary memory accesses and computations. For FA residual method, $\boldsymbol{A}$ is stored as a sparse matrix and its container is implemented in the FEDomain package. The knowledge of the data storage algorithm allows to perform data manipulation efficiently as can be seen in Table 6.1.

## 6.3 FEResidual Version 3

### 6.3.1 Interface

FEResidual version 3 is designed to improve the performance of residual methods and to make the FEResidual class independent of the third party data containers.

The solution of the EBE performance bottleneck is to obtain elements data in the FEResidual provided data containers. The current element interface does not allow to provide FEResidual internal data containers, so the only possible solution is to store elements data internally into the FEResidual object. The current element interface methods allow data copying in the FEResidual object by accessing the element data and copying each entry one by one into the FEDomain internal containers through their get functions. The copying to elements load vectors and stiffness matrices is efficient as these are densely populated containers of dimensions $N_K$ and $N_K \times N_K$, respectively. The $\mathbf{\Lambda}_K$ is of dimensions $N_K \times N$ where each row will be sparsely populated. The lack of user provided data containers internal memory storage algorithm for $\mathbf{\Lambda}_K$ make copying inefficient as most of the data is zero and will not be stored into a sparse matrix container. In FEResidual object the copying of the elements data into FEResidual internal data containers cannot be performed efficiently. The user element has the knowledge of the $\mathbf{\Lambda}_K$, and then it will be in better position to populate FEResidual objects internal objects. The new element interface in Listing 6.7 is introduced to increase the performance while keeping support to generic implementation of element classes. The new element interface allows the FEResidual object to provide its internal data container to the element objects. It will be the user elements responsibility to populate these containers. This interface will allow the FEResidual class to be independent of the user element classes internal details.

```
virtual size_t get_DOFs_count() = 0;
virtual void getLoad(FEVector& _l) = 0;
virtual void getStiffness(FEMatrix& _s) = 0;
virtual void getConnectivity(FESparseMatrix& _c) = 0;
```

Listing 6.7: Element Interface

The element interface in Listing 6.7 allows the FEResidual objects to provide their internal data containers to elements objects. The data containers will be

constructed in the FEResidual object but their size will depend on the element's DOFs. The *get_DOFs_count()* is added in element abstract interface in Listing 6.7 to get elements DOF counts. The FEResidual class interface is modified as the FEResidual object does not require knowledge for elements internal DOFs. The latest FEResidual class in Listing 6.8 has two template arguments, the first argument represents the abstract base element class and second represents residual container class.

```
template <typename TElement, typename TGlobal_Vector>
class FEResidual{
      FEResidual(vector<TElement*> &elements, size_t t_DOFs);
      TGlobal_Vector residual_FA(const TGlobal_Vector& X);
      TGlobal_Vector residual_EBE(const TGlobal_Vector& X);
};
```

Listing 6.8: FEResidual Interface Version3

### 6.3.2 Implementation

In FEResidual version 2, the data manipulation, specially for matrices, is the fundamental barrier for achieving efficiency. In FEResidual vesrion 3, the data container required to obtain element data are created and provided to elements by the FEResidual object. In FEDomain package three data container classes (*FEVector*, *FEMatrix* and *FESparseMatrix*) are added to store element data. To take full advantage of their internal storage algorithms, these classes also provide data manipulation functions. This version of FEResidual has omitted FEMath library as these containers are responsible for their data manipulation.

FEVector is a one dimensional dense data container implemented to obtain the element's load vector. The FEVector has [ ] subscript operator for get and set functions interface. FEMatrix is a two dimensional data container class implemented for a densely populated matrices like element stiffness matrix. The FEMatrix

memory storage algorithm is implemented as $std :: vector < std :: vector < double >>$. The FEMatrix get and set functions interfaces are shown in Listing **??**. For a scarcely populated matrix, the FESparseMatrix class is introduced in the FEDomain package. It is a two dimensional data container class and it is provided to an element to get its connectivity matrices. The FESparseMatrix class has the same interface as the FEMatrix class, but its storage algorithm is $std :: map < size\_t, std :: map < size\_t, double >>$. The storage algorithm allows us to add data randomly as the map always keep the data in sorted order, but it takes almost five time more storage space to store an entry.

The FESparseMatrix is used for element connectivity matrix or system stiffness matrix. The connectivity data cannot be provided as a vector because it is possible to have more than one entry in a row.

### 6.3.3 Performance

In FA residual method, the global stiffness matrix and load vector are constructed during the first iteration. The connectivity matrices are used to construct global structures. The global stiffness matrix is also a sparse matrix. While, in the EBE method, during each iteration the residual is calculated for each element, and this calculation involves the connectivity matrix as shown by (2.38). The new element interface should positively affect both methods. The EBE method should present a large improvement in performance. This efficiency is displayed in the performance Table 6.2.

| DOFs | Elements | Method | Iter. | Const Time | Convgs Time | Per Iter Time |
|---|---|---|---|---|---|---|
| 239 | 476 | FA | 1705 | 6.033e-06 | 54.0355 | 3.169e-02 |
| 239 | 476 | EBE | 1705 | 8.01e-06 | 119.587 | 7.014e-02 |
| 956 | 1910 | FA | 6090 | 1.615-e05 | 3454.77 | 5.673e-01 |
| 956 | 1910 | EBE | 6090 | 1.811e-05 | 1749.08 | 2.872e-01 |
| 3741 | 7483 | FA | 20511 | 5.502e-05 | 7144.76 | 3.483e-01 |
| 3741 | 7483 | EBE | 20511 | 5.568e-05 | 429.444 | 2.094e-02 |
| 3721 | 7443 | FA | 20129 | 5.459e-05 | 7060.44 | 3.508e-01 |
| 3721 | 7443 | EBE | 20129 | 5.511e-05 | 424.202 | 2.107e-02 |
| 14573 | 29147 | FA | 64473 | - | - | - |
| 14573 | 29147 | EBE | 64473 | 1.862e-04 | 5349.54 | 8.297e-02 |

Table 6.2: FEResidual V3 Timing Table

The FA residual method speed has reduced significantly when compared to the previous version. In the previous FEResidual version, the elements were responsible for the construction, population and store containers for their local data. The FEResidual in construction stage accesses elements data. The FEResidual object practically spend no time in obtaining elements data as these elements only pass the references of their data containers. In the current FEResidual class implementation testing the user element classes used do not construct their local data structures to store $\boldsymbol{A}_K$, $\vec{b}_K$, and $\boldsymbol{\Lambda}_K$ to store memory. These elements populate the provided data containers by computing local data on request. This makes data retrieval task more time consuming as can be seen in the Table 6.2. The sparse matrix provided in this implementation has generic data manipulation functions like matrix vector, matrix matrix, and transpose matrix matrix multiplication. The current interface forces the application to perform unnecessary computations for mapping $\boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K$. The first two matrices are multiplied efficiently as $\boldsymbol{A}_K$ is a dense matrix but the product of $\boldsymbol{\Lambda}_K^T \boldsymbol{A}_K = \boldsymbol{T}_1$ is a sparse matrix and most of the columns in $\boldsymbol{\Lambda}_K$ are empty. Then, $\boldsymbol{T}_1 \boldsymbol{\Lambda}_K = \boldsymbol{T}_2$ is a time consuming task where most of the columns are empty. Since fetching the value from these columns will return mostly zero, most of the work done has no advantage as $\boldsymbol{T}_2$ is a sparse

matrix. Since the mapping of $\boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K$ has to be performed for each element data, a dedicated function can be added to the sparse matrix class which performs mapping efficiently by reducing computations and avoiding the temporary matrix $\boldsymbol{T}_1$.

In EBE, the data gathering method remained unchanged. The elements data are accessed in each iteration which means that the elements data are computed for every iteration. In each EBE iteration the data containers are constructed for each element according to their DOFs count, plus some extra vectors to assist in internal calculations. One of the extra vectors is required to map the global solution $u$ to local solution $u_K$. The residual method has consumed most of the time creating and deleting elements data containers.

### 6.3.4 Drawbacks

Confining the generality of the FEResidual class and introducing the compatible element class interface (given in Listing 6.7) has dramatically affected the performance of the FEResidual library. The examination of the FEResidual Version 3 has revealed a number of drawbacks. The residual function returns a vector which creates a memory copy of residual vector in each residual iteration. The amount of memory copy in each iteration depends upon $N$ for a mesh. The data copy increases with the bigger meshes. The second factor is the number of iterations. If the iterative algorithm used converge slowly, the amount of data copied will be increased with every iteration.

For the EBE method, the element's data $\boldsymbol{A}_K$, $\vec{b}_K$, and $\boldsymbol{\Lambda}_K$ is accessed for each iteration. The recurrent copying of element's data was selected to keep memory requirements of the library to a minimum but adversely affects the FEResidual performance.

If a user's choice of FEResidual method is known at the construction stage, different optimization techniques can be applied to the FEResidual class. The current implementation allows the user to call both the residual methods in any order, and with in the same execution. This is clearly an inefficient option.

## 6.4 FEResidual Version 4

### 6.4.1 Interface

The FEResidual version 4 is implemented to improve the performance of the FEResidual class by removing the design and implementation drawbacks in FEResidual version 3. FEResidual Version 4 is given in Listing 6.9. A new parameter *method_t* is added in the FEResidual class construct. It defines the user's choice of residual method (FA or EBE) for the FEResidual object life time. The user has to select the residual algorithm during construction stage which will be fixed for the FEResidual object life time. The FEResidual interface in Listing 6.9 has a single residual function. The signature of the residual function has also been modified. This method will implement the residual method selected at the construction stage.

```
template <typename TElement, typename TG_Vector>
class FEResidual{
      FEResidual(vector<TElement*>& e, size_t TDofs, method_t res);
      void residual(const TG_Vector& X, TG_Vector& R);
};
```

Listing 6.9: FEResidual Interface V4

### 6.4.2 Implementation

The confirmation of the residual method at the construction stage enables the FEResidual object to create the internal memory containers. In case of the FA

method, the data containers are constructed to accommodate $\boldsymbol{A}$ and $\vec{b}$. While for the EBE residual method, the data containers are constructed to accommodate all the elements data $\boldsymbol{A}_K$, $\vec{b}_K$, and $\boldsymbol{\Lambda}_K$, individually. In both the residual methods, during the construction stage, all the elements data are stored in these containers and remains for the life time of the FEResidual object.

The new interface has a single residual function which will implement the selected residual algorithm. The signature of the residual method has also been modified to achieve efficiency. It has void return data and has two parameters. The first parameter is a constant reference to the approximate solution vector and the second parameter is a reference to the residual vector. The residual signature does not return residual vector which enables us to avoid unnecessary memory copy in every iteration. The user has to provide the residual vector in which all the values should be set to zero. In FEResidual solver the residual data is added into the residual container. If the container has not set to zero, then the FEResidual provided residual data can be corrupted.

### 6.4.3 Performance

The computational time of the FEResidual Version 4 for both the residual methods is given in Table 6.3. From the comparison of the EBE method timing for FEResidual Version 3 and Version 4, it is observed that the constructor timing has increased and the convergence timing has reduced. The constructor stage computation time is given in 5th column of Table 6.3. In this implementation, an additional task of elements data gathering is performed at the construction stage. The 6th column in Table 6.3 has the time consumed for all the residual iterations. These timings have significantly improved as all the required elements data are present in the FEResidual object.

| DOFs | Elements | Method | Iter. | Const Time | Convgs Time | Per Iter Time |
|---|---|---|---|---|---|---|
| 239 | 479 | FA | 1705 | 2.52E-02 | 3.09E-02 | 1.81E-05 |
| 239 | 479 | EBE | 1705 | 5.96E-02 | 10.33 | 6.07E-03 |
| 956 | 1913 | FA | 6090 | 3.025E-02 | 6.12E-01 | 1.00E-04 |
| 956 | 1913 | EBE | 6090 | 9.9541E-03 | 38.697 | 6.3541E-04 |
| 3741 | 7443 | FA | 20129 | 4.12E-02 | 7.12 | 3.47E-03 |
| 3741 | 7443 | EBE | 20129 | 4.53E-02 | 55.76 | 2.77E-03 |
| 14573 | 29144 | FA | 64473 | 7.08 | 156.90 | 2.43E-03 |
| 14573 | 29144 | EBE | 64473 | 7.54E-02 | 495.10 | 7.67E-03 |

Table 6.3: FEResidual V4 Timing Table

## 6.5 Conclusion

In this chapter the sequential implementation of the FEDomain package residual solvers is discussed. The consecutive different versions of FEResidual class is presented, and their respective short comings are discussed. This process ended when all the requirements were met in version 4. The aim is to implement a parallel version of iterative solver for shared memory architecture is discussed in the next chapter.

# Chapter 7

# FEDomain Shared Memory Residual Method

In Chapter 6, the FEResidual class was developed as a sequential library. In this chapter the multi-threaded version of FEResidual class will be developed for shared memory processor computers.

## 7.1 FEResidual Version 5

### 7.1.1 Interface

The FEResidual version 5 is a parallel implementation for shared memory processors of FEResidual version 4. The objective of the FEResidual version 5 is to obtain execution speedup by distributing the computational load among threads. Each thread will be allocated a fraction of the computational load and each thread should require a fraction of the time to perform the allocated computations. In theory, when the computation load is divided evenly between two threads the computational time should also be reduced to half. These threads will be executed in parallel and each thread should finish in almost half the total execution time. The aim of the FEResidual version 5 is to provide a residual solver which

performs efficiently on the generic multi core processor architectures. It should provide compatible computational time while requiring minimum effort from the user.

The FEDomain package does not have knowledge about the target processor architecture. Every processor can support different number of maximum threads. The FEResidual version 5 interface is shown in Listing 7.1 which has a new parameter *max_thread* to get available threads. This parameter will allow the user to select the number of parallel threads being deployed during execution. Both methods for calculating residual defined in previous versions (FA and EBE), are also present in this version and the residual method signature remains unmodified.

```
template <typename TElement, typename TG_Vector>
class FEResidual{
      FEResidual(vector<TElement*>& elements, size_t& Total_DOFs,
                 method_t& residual_method, size_t& max_thread);
      void residual(const TG_Vector& X, TG_Vector& R);
};
```

Listing 7.1: FEResidual Interface V5

### 7.1.2 Implementation

In FEReisdual version 5 two new residual calculation methods *RANDOM* and *WEIGHTED* are introduced in addition to the FA and EBE. These methods are parallel implementations of EBE residual method. In EBE method, the residual is calculated using elements' data only. Multiple elements residual vectors can be calculated simultaneously in parallel threads. The main idea is to divide the total computational load into sub loads. These sub loads should have similar computational work loads. The number of sub loads should be equal to the number of threads so that a single sub load is allocated to each thread. Each thread should consume similar amount of time to finish execution. For EBE method the

total computation load is the number of allocated mesh elements objects. For the RANDOM and WEIGHTED methods, the set of provided mesh elements are divided into subsets. The number of subsets will be equal to the number of threads but the total number of elements in each subset will depend on the distribution algorithm (RANDOM and WEIGHTED). The aim is for each thread to calculate the residual vector for the allocated subset of elements in similar time and store the results in the global residual vector $\vec{r}$.

For the RANDOM method, it is assumed that all the provided elements have the same amount of DOFs. The user provided element is considered as the unit of computation, and sub computational load is measured as the number of elements provided to each thread. The provided elements are divided into subsets where each subset contains almost the same number of elements. Each element will be present in only one subset. This method of load distribution is useful if the elements have the same number of DOFs, because this would lead to similar computational load. The WEIGHT method is designed for a non uniform mesh. In this case, the elements can have variable number of DOFs. The computational load of these elements will vary and are dependent on the DOFs. The computational cost of each element varies with the type of problem being solved (Poisson, Elasticity, Convection Diffusion, etc), location of the element (boundary, interior) and the mesh dimensions (2D, 3D, etc). The elements are distributed among threads such that each subset has a similar number of DOFs. The weighted scheduling can be implemented without modification of element interface as it already has a method $get\_DOFs\_count()$ to retrieve number of element DOFs.

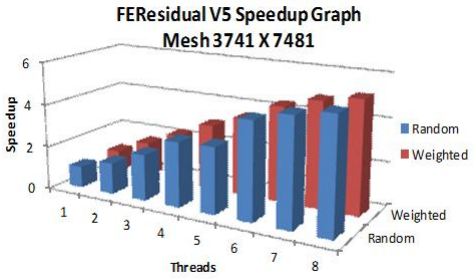
### 7.1.3 Performance

The Poisson 2D mesh with 3741 DOFs and 7483 elements is used to test the multi threaded FEResidual version 5 on a quad core Intel® Xeon 5560 processor. It supports maximum of 8 parallel threads (4 cores in a processor and 2 threads on

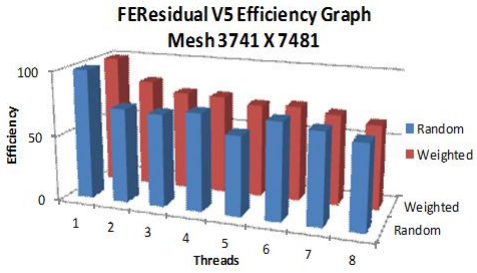
each core). Table 7.1 displays timing data of EBE, Random and Weight methods and the speed up achieved by using multi threaded executions. There is no modification occurred in the FA method so results are not included in the table. The parallel methods have achieved similar speed up. For example, for 8 threads both have achieved 5% as can be seen in the Figure 7.1a. The speed up and efficiency are given in Figures 7.1a and 7.1b respectively.

| DOFs | Elements | Thread | Const Time | Convgs Time | SpeedUP | Efficiency |
|---|---|---|---|---|---|---|
| 3741 | 7483 | EBE | 0.026 | 36.523 | 1 | 100 |
| RANDOM | | | | | | |
| 3741 | 7483 | 2 | 0.026 | 25.07 | 1.46 | 72.85 |
| 3741 | 7483 | 3 | 0.019 | 17.08 | 2.14 | 71.28 |
| 3741 | 7483 | 4 | 0.026 | 12.18 | 3.00 | 74.95 |
| 3741 | 7483 | 5 | 0.026 | 11.95 | 3.06 | 61.12 |
| 3741 | 7483 | 6 | 0.020 | 8.178 | 4.47 | 74.44 |
| 3741 | 7483 | 7 | 0.026 | 7.423 | 4.92 | 70.29 |
| 3741 | 7483 | 8 | 0.020 | 7.013 | 5.21 | 65.10 |
| WEIGHTED | | | | | | |
| 3741 | 7483 | 2 | 0.021 | 21.94 | 1.66 | 83.23 |
| 3741 | 7483 | 3 | 0.025 | 15.93 | 2.29 | 76.44 |
| 3741 | 7483 | 4 | 0.026 | 12.05 | 3.03 | 75.77 |
| 3741 | 7483 | 5 | 0.025 | 10.19 | 3.58 | 71.70 |
| 3741 | 7483 | 6 | 0.017 | 8.333 | 4.38 | 73.05 |
| 3741 | 7483 | 7 | 0.023 | 7.519 | 4.86 | 69.39 |
| 3741 | 7483 | 8 | 0.016 | 7.093 | 5.15 | 64.37 |

Table 7.1: FEResidual V5 timing for 3741 DOFs and 7483 elements mesh. It took 20511 iterations to converge with the Jacobi mehtod.




(a) SpeedUp graph



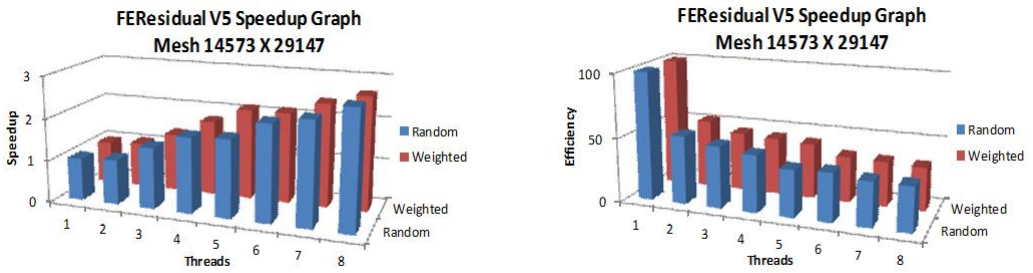

(b) Efficiency graph

Figure 7.1: The data is collected for the 3741 DOFs and 7483 element mesh using FEResidual V5

The 2D mesh with 14573 DOFs and 29147 elements (triangles and edges) is solved with FEResidual methods and their timing results are in Table 7.2. The FEResidual has not achieved the same speed up, as before. Figure 7.4a depicts the speed up graph which increases with the threads and Figure 7.4b depicts the efficiency graph which reduces as the threads increases.

| DOFs | Elements | Threads | Const Time | Convgs Time | Speedup | Efficiency |
|---|---|---|---|---|---|---|
| 14573 | 29147 | 1 | 0.0615 | 530.742 | 1 | 100 |
| RANDOM | | | | | | |
| 14573 | 29147 | 2 | 0.0745 | 501.874 | 1.058 | 52.876 |
| 14573 | 29147 | 3 | 0.0680 | 369.132 | 1.438 | 47.927 |
| 14573 | 29147 | 4 | 0.0741 | 299.272 | 1.773 | 44.336 |
| 14573 | 29147 | 5 | 0.0611 | 292.177 | 1.817 | 36.330 |
| 14573 | 29147 | 6 | 0.0798 | 235.565 | 2.253 | 37.551 |
| 14573 | 29147 | 7 | 0.0821 | 219.535 | 2.418 | 34.537 |
| 14573 | 29147 | 8 | 0.0602 | 192.539 | 2.757 | 34.457 |
| WEIGHTED | | | | | | |
| 14573 | 29147 | 2 | 0.0749 | 501.314 | 1.058 | 52.916 |
| 14573 | 29147 | 3 | 0.0609 | 387.388 | 1.370 | 45.652 |
| 14573 | 29147 | 4 | 0.0644 | 299.554 | 1.771 | 44.278 |
| 14573 | 29147 | 5 | 0.0661 | 249.110 | 2.130 | 42.595 |
| 14573 | 29147 | 6 | 0.0744 | 248.579 | 2.134 | 35.572 |
| 14573 | 29147 | 7 | 0.0684 | 218.348 | 2.430 | 34.712 |
| 14573 | 29147 | 8 | 0.0761 | 198.485 | 2.673 | 33.412 |

Table 7.2: FEResidual V5 Timing Table for mesh with 14573 DOFs and 29147 elements. The solution required 64473 iterations using the Jacobi method.




(a) SpeedUp graph

(b) Efficiency graph

Figure 7.2: Data collected for mesh with 14573 DOFs using FEResidaul V5

## 7.2 FEResidual Version 6

### 7.2.1 Implementation

OpenMP [72] requires its user to define the number of threads, the scheduling scheme, and a suitable chunk size before the execution of the parallel application. For OpenMP, these parameters can be specified at multiple levels. In first approach, the tuning parameters are defined individually for each parallel region. The advantage of this method is that it fine tunes the library. Due to large amount of parallel regions this is not suitable for the FEResidual solvers. These parameters cannot actually be selected until the executing problem type is not defined. The second approach is to set these parameters at the run time dynamically through environment variables. The same parameter values are set for every parallel region in the application. These parameters can either be set in an application using OpenMP routines, or as the environment in the console.

FEResidual Version 6 has two residual FA and EBE methods. The FA is not implemented in parallel as the elements data has to be copied into global stiffness matrix and load vector. It is observed that OpenMP threads spend more time idle waiting for the data resource. The EBE method code is reimplemented using OpenMP parallel regions. The OpenMP enables the user to execute the EBE residual method as a sequential code by setting $OMP\_NUM\_THREADS$ to one, or parallel by setting it greater than one. The OpenMP provides two methods to set parameters dynamically. In the first one, the scheduling parameter is set to *AUTO*. In this case the compiler selects the scheduling algorithm and chunk size at compile time. The compiler does not have the information about the user application implementation and problem size. In the second method, the scheduling is set to *RUNTIME*. It defers the scheduling decision until run time. The FEResidual application parallel regions will take value for scheduler and chunk size from environment. These values are set using OMP_SCHEDULE environment

variable. The environment variable OMP_NUM_THREADS is used to set the maximum number of threads available to parallel regions. In FEResidual version 6 all the parallel regions scheduler scheme is set dynamically, so the library can be tuned for different problem types and sizes, for the most optimised execution time.

The application of the Dirichlet constraints are introduced in FEResidual versions 6. The implementation of the Dirichlet boundary conditions involves a modification in the stiffness matrix and the load vector. In FEResidual class the constraints are applied at constructor stage during data assembly. The FEResidual constructor requires Dirichlet constraints ids and values from the user. In Listing 7.2 the second argument of the FEResidual class constructor represents Dirichlet data. This is of $std :: map$ type, where the key field is Dirichlet DOF id and the value field represents the Dirichlet value.

The enumeration structure FEType is added into FEResidual class to represent the residual method to be implemented. The FEType has three methods (FE_FA, FE_EBE and FE_TBB) of residual computation.

### 7.2.2 Interface

An abstract element class FEElement has been introduced into the FEResidual version 6. This class contains the declaration of all the element interface methods required by the FEResidual object to gather elements' data. In this version the abstract FEElement class defined in Listing 3.11 is introduced. The users are no longer required to provide abstract element class as the template parameter but is required to inherit its element classes from the FEElement class. The FEElement class is provided in the FEDomain package and defined in the Section 3.1.

The FEResidual interface is modified due to the introduction of the FEElement class. The FEResidual class does no longer require a first template parameter. As the FEResidual is implemented for C++ application developer and *std* :: *vector* objectts are used for the consecutive memory data containers. In residual function class the signature is modified and the user has to provide an approximate solution $\alpha_{appr}$ and residual $r$ as vectors of size $N$. The FEResidual Version 6 interface is defined in Listing 7.2. This interface is designed to be simple and does not involve any details about scheduling, chunk size, and threads. The method_t is enumerated type to represent the selected residual method. The ele_max_DOFs is the max number of DOFs in any element of the mesh. This information is required for efficient internal implementation of residual methods by and the default value is 3. The total_sys_DOFs represents total number of DOFs $N$ in a mesh. The Dirichlet_constraints represent the Dirichlet data which includes the Dirichlet DOFs ids and its value.

```
class FEResidual{
public:
      FEResidual(vector<FEElement*>& list_of_elements,
                 map<size_t,double>& Dirichlet_constraints,
                 size_t& total_sys_DOFs, size_t& ele_max_DOFs,
                 FEType& residual_method);
      void residual(const vector<double>& approximate_solution,
                    vector<double>& residual);
};
```

Listing 7.2: FEResidual Interface V6

### 7.2.3 Performance

The FEResidual Version 6 has been executed on a machine having an Intel Xeon X5560 2.6GHz processor (quad core and supports 8 threads), 48290 MB RAM and running GUN/Linux h2.6.18 OS. The Xeon X5560 has 4 computational core and due to hyper threading it provides 8 threads. The mesh used during execution has

3676673 DOFs and 7353344 elements. The Jacobi iterative method is implemented to compute the solution. Each execution is stopped after 100 iterations. The mesh is executed with various sets of scheduling schemes, threads and chunk sizes. The chunk size is selected by dividing the number of elements by the number of threads. This will approximately equally distribute elements among threads.

Table 7.3 represents the timing data for STATIC scheduler. As the number of threads increases the application gains speedup, the maximum speedup attained is 3.3 for 8 threads but on other hand efficiency of the system reduces.

| OpenMP Threads | Chunk Size | Construct Time | Convergence Time | Per Itera. Time | Total Time | Speedup | Efficiency |
|---|---|---|---|---|---|---|---|
| 1 | 7353344 | 7.442 | 157.736 | 1.577 | 165.178 | 1 | 100 |
| 2 | 3676672 | 5.808 | 86.918 | 0.869 | 92.726 | 1.78 | 89.07 |
| 3 | 2451114 | 5.366 | 62.349 | 0.623 | 67.715 | 2.44 | 81.31 |
| 4 | 1838336 | 7.274 | 50.805 | 0.508 | 58.079 | 2.84 | 71.10 |
| 5 | 1470668 | 8.834 | 44.618 | 0.446 | 53.453 | 3.09 | 61.80 |
| 6 | 1225557 | 10.546 | 42.787 | 0.428 | 53.333 | 3.10 | 51.62 |
| 7 | 1050477 | 10.527 | 41.563 | 0.416 | 52.090 | 3.17 | 45.30 |
| 8 | 919168 | 10.673 | 39.378 | 0.394 | 50.051 | 3.30 | 41.25 |

Table 7.3: The table contains the timing data for the FEResidual V6 EBE method. The mesh used has 3676673 DOFs and 7353344 elements. The OpenMP Static scheduler with variable threads and chunk size are used.

Table 7.4 represents the timing data of executions using DYNAMIC scheduler. It shows that best execution time is not always achieved using maximum threads. For DYNAMIC scheduler the total time of 49.24 is achieved using 7 threads which has slightly better than best total execution time attained using STATIC scheduling. The total time consumed for the thread 5, 6, 7 and 8 threads is almost similar.

| OpenMP Threads | Chunk Size | Construct Time | Convergence Time | Per Itera. Time | Total Time | Speedup | Efficiency |
|---|---|---|---|---|---|---|---|
| 1 | 7353344 | 7.5889 | 155.607 | 1.5561 | 163.1959 | 1 | 100 |
| 2 | 3676672 | 5.4899 | 89.0070 | 0.8901 | 94.4969 | 1.7270 | 86.3498 |
| 3 | 2451114 | 5.0073 | 64.7203 | 0.6472 | 69.7276 | 2.3405 | 78.01593 |
| 4 | 1838336 | 6.7425 | 52.0522 | 0.5205 | 58.7947 | 2.7757 | 69.3922 |
| 5 | 1470668 | 9.0666 | 44.4322 | 0.4443 | 53.4988 | 3.0504 | 61.0092 |
| 6 | 1225557 | 10.0679 | 42.5227 | 0.4252 | 52.5906 | 3.1031 | 51.7190 |
| 7 | 1050477 | 10.4172 | 38.8230 | 0.3882 | 49.2402 | 3.3143 | 47.3469 |
| 8 | 919168 | 10.1966 | 40.2208 | 0.4022 | 50.4174 | 3.2369 | 40.4612 |

Table 7.4: FEResidual V6 Timing Table with 3676673 DOFs mesh having 100 iterations for Dynamic Scheduler with variable threads and chunk size.

Table 7.5 represents the timing data using GUIDED scheduler. The best total time of 48.96 is achieved using 7 threads but execution time using 5,6 and 8 threads is very similar. This appeared to the best configuration among all 24 different configurations as shown in Figure 7.3. The current design helps to tune the FEResidual according to hardware and software as it been observed the same software can behave differently on different hardware.

| OpenMP Threads | Chunk Size | Construct Time | Convergence Time | Per Itera. Time | Total Time | Speedup | Efficiency |
|---|---|---|---|---|---|---|---|
| 1 | 7353344 | 7.5807 | 155.976 | 1.5598 | 163.5567 | 1 | 100 |
| 2 | 3676672 | 5.9221 | 94.7639 | 0.9476 | 100.6860 | 1.6244 | 81.2212 |
| 3 | 2451114 | 5.1669 | 65.2900 | 0.6529 | 70.4569 | 2.3214 | 77.3791 |
| 4 | 1838336 | 6.6160 | 51.7542 | 0.5175 | 58.3702 | 2.8021 | 70.0514 |
| 5 | 1470668 | 8.9628 | 44.6589 | 0.4466 | 53.6217 | 3.0502 | 61.0040 |
| 6 | 1225557 | 9.9410 | 44.6084 | 0.4461 | 54.5493 | 2.9983 | 49.9721 |
| 7 | 1050477 | 10.4654 | 38.4932 | 0.3849 | 48.9586 | 3.3407 | 47.7245 |
| 8 | 919168 | 10.4771 | 39.2966 | 0.3930 | 49.7737 | 3.2860 | 41.0751 |

Table 7.5: FEResidual V6 Timing Table with 3676673 DOFs mesh having 100 iterations for Guided Scheduler with variable threads and chunk size.

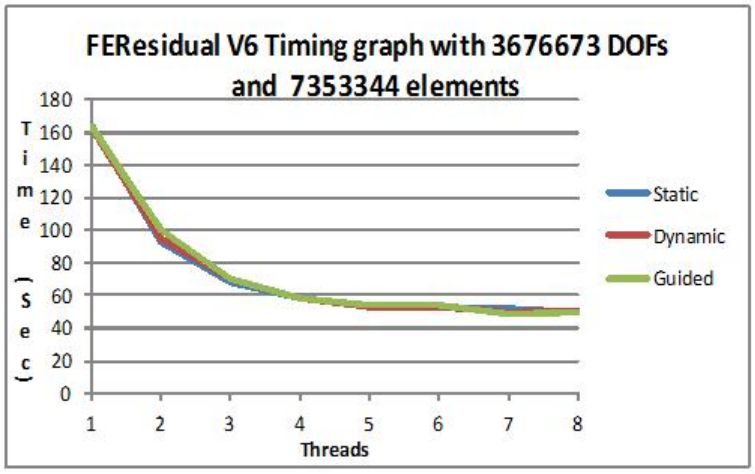
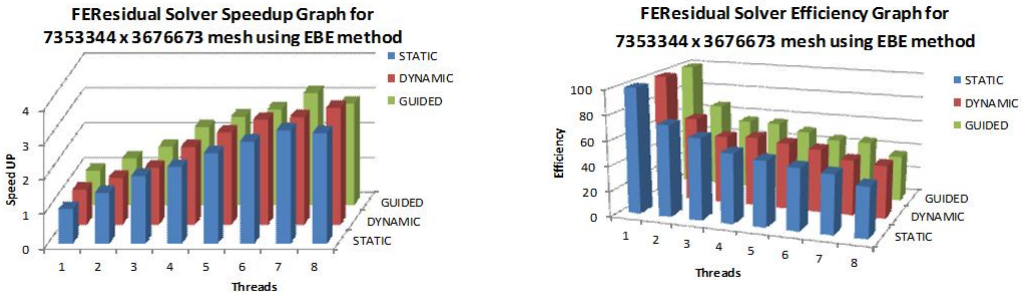



Figure 7.3: Timing Graph for OpenMP scheduling algorithms for 1 to 8 threads.

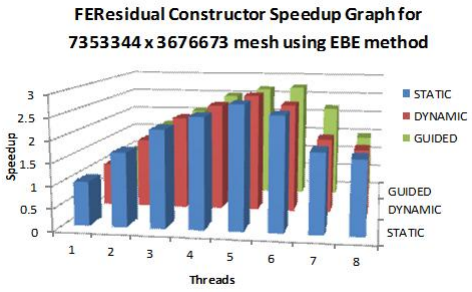



(a) SpeedUp graph

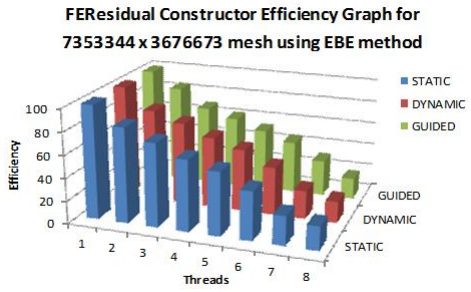
(b) Efficiency graph

Figure 7.4: The data is collected for the mesh with 3676673 DOFs on Intel X5560 by FEResidual Version 6 (EBE method). It displays data for the complete solution.

Figure 7.5 is the constructor speedup and efficiency graph for the mesh with 3676673 DOFs. It considers OpenMP schedulers (STATIC, DYNAMIC and GUIDED) for 1 till 8 threads. These results are obtained by executing on a machine having an Intel X5560 processor with 4 cores and 8 threads. The X5560 processor [4] has DDR3 memory having three memory channels. Figure 7.5a has the construction speedup data. The FEResidual constructor attained speedup till five threads. When the threads increase the constructor timing start to increase. The memory access is the bottle neck for constructor speed up. The increase in the threads



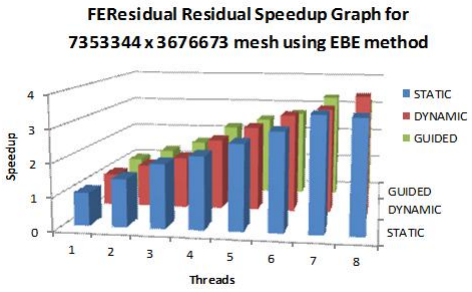

(a) SpeedUp graph



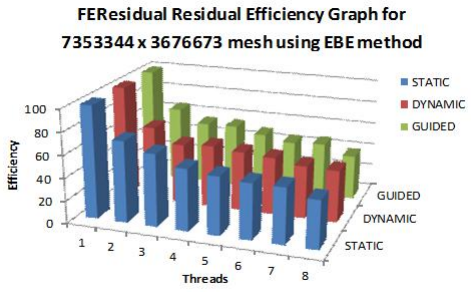

(b) Efficiency graph

Figure 7.5: The data is collected for the mesh with DOFs 3676673 on Intel X5560 by FEResidual Version 6 (EBE method). It displays data for the FEResidual constructor.

required more parallel memory access while the memory channels and their data transfer rate remains constant. It is believed that after 5 threads the memory channels cannot keep up the memory requests from the parallel threads. The rate of memory requests increases with the increase in threads, thus some of the threads have to stall during constructor for response from memory channels.




(a) SpeedUp graph



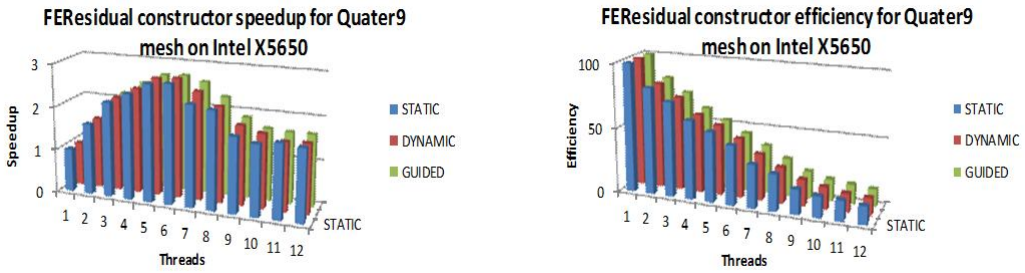

(b) Efficiency graph

Figure 7.6: The data is collected for the mesh with 3676673 DOFs on Intel X5560 by FEResidual Version 6 (EBE method). It displays data for the FEResidual residual calculation.

Figure 7.6 is the residual speedup and efficiency graphs for the 100 iterations. The residual method has achieved a constant speedup with increase in number of OpenMP threads as shown in Figure 7.6a. It has achieved the 3 speedup for the eight threads. The residual member function does not has to save large amounts

of data into memory, and memory channels do not affect the performance in the residual function. The residual has achieved the speed up of 4 for all three OpenMP schedulers in Figure 7.6a. The residual efficiency graph is shown in Figure 7.6b.

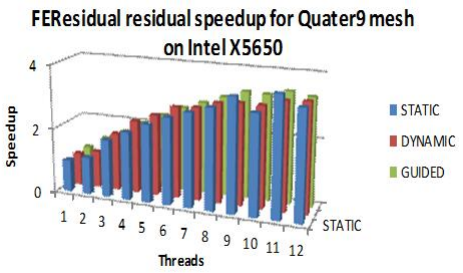
The same mesh is also executed on Intel X5650 [5] processor having 6 cores and 12 threads. The X5650 has DDR3 memory with three memory channels. Figure 7.7a and Figure 7.7b depict the speedup and efficiency graphs. The constructor has the constant speedup rise till 6 threads. The constructor speedup starts to reduce till 9 threads and speedup becomes constant after 9 threads.



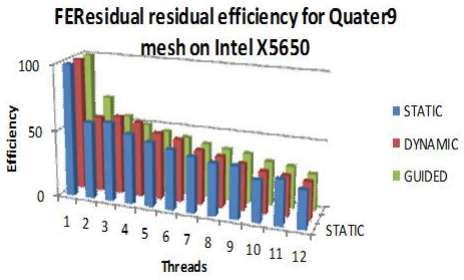

(a) SpeedUp graph (b) Efficiency graph

Figure 7.7: The data is collected for the mesh with 3676673 DOFS on Intel X5650 by FEResidual Version 6 (EBE method). It displays data for the FEResidual constructor.

Figure 7.8a and Figure 7.8b are the speedup and efficiency graphs for the residual functions for 100 iterations. The residual function gained speedup till 10 threads and become almost constant for 10, 11 and 12 threads. The residual behaviour of the FEResidual class residual method has shown the same behaviour on both the processors.



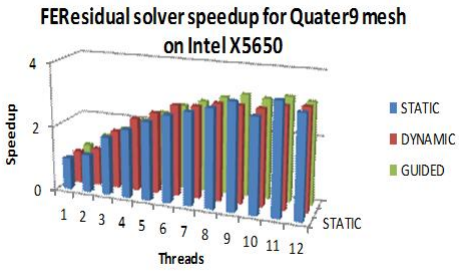

(a) SpeedUp graph



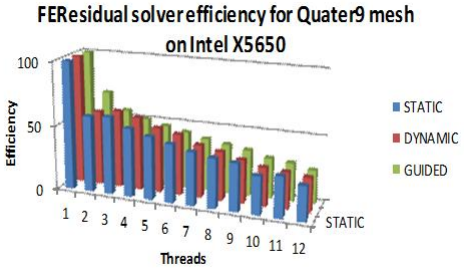

(b) Efficiency graph

Figure 7.8: The data is collected for the mesh with 3676673 DOFs on Intel X5650 by FEResidual Version 6 (EBE method). It displays data for the FEResidual residual method.

Figure 7.9a and Figure 7.9b are the speedup and efficiency graphs for the entire solution. The Jacobi iterative algorithm is used to solve a linear system that represents a large number of iterations to converge to the solution. The constructor timing and behaviour is not significant as most of the execution time has been used for residual iterations. The residual iterations have achieved speedup of almost 4 on both processors for the 2D Poisson problem.



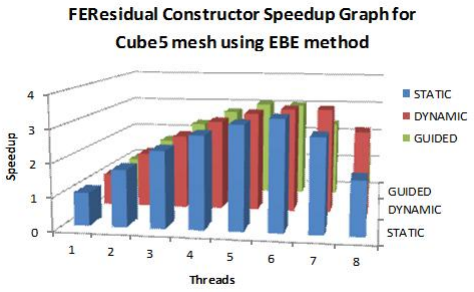

(a) SpeedUp graph



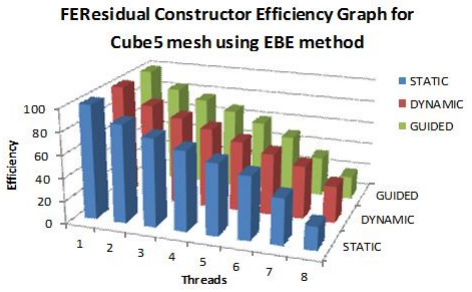

(b) Efficiency graph

Figure 7.9: The data is collected for the mesh with 3676673 DOFs on Intel X5650 by FEResidual Version 6 (EBE method). It displays data for the FEResidual solution.

Now the FEResidual is executed with the 3D elasticity problem. For a 3D mesh, the tetrahedral element performs atleast 144 multiplication and addition for its



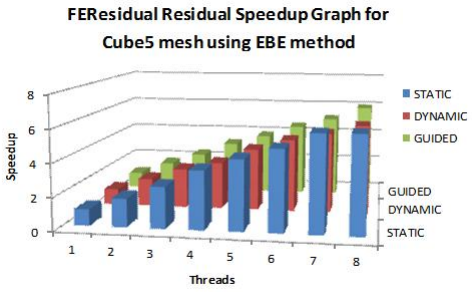

(a) SpeedUp graph



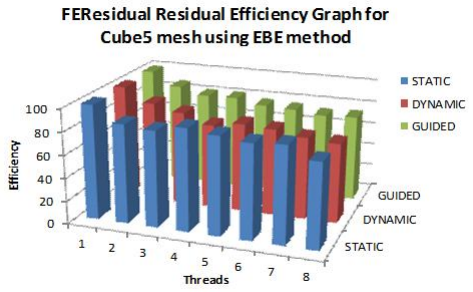

(b) Efficiency graph

Figure 7.10: The data is collected for the 3D mesh with 411939 DOFs on Intel X5560 by FEResidual Version 6 (EBE method). It displays data for the FEResidual constructor.

stiffness matrix calculation. The tetrahedral element will perform about 16 times more computation than triangle elements. The 3D elasticity problem in first case is executed on the Intel X5560 processor for three OpenMP scheduling algorithms. For each case, the execution are stopped after 100 residual iterations. Figures 7.10a and 7.10b are the FEResidual EBE constructor speedup and efficiency graphs. The constructor has showed the same behaviour was in case of 2D Poisson problem. The FEResidual constructor for 3D elasticity has achieved a speedup of 3. In both the problem the efficiency reduction are same. Figures 7.11a and 7.11b is the 100 residuals iterations speedup and efficiency graphs for the 3D Elasticity problem. The residual has speedup of 6 and its efficiency is always above 60. Figures 7.12a and 7.12b are the speedup and efficiency graphs of the FEResidual class for the 3D Elasticity problem. The FEResidual iterative method has achieved the maximum speedup of 4 and efficiency of 60. The FEResidual iterative EBE method has performed better for the 3D elasticity problem as compare to 2D Poisson problem. It has speedup of 4 for 3D Elasticity as compare to 3 for 2D Poisson problem. It has achieved the efficiency of 60 for the 3D Elasticity and 30 for the 2D Poisson problem.

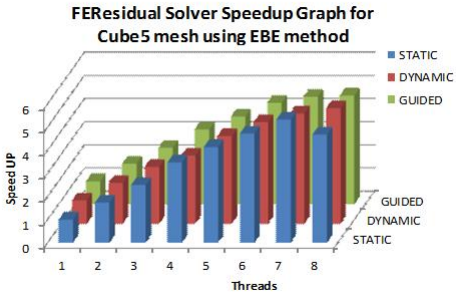



(a) SpeedUp graph

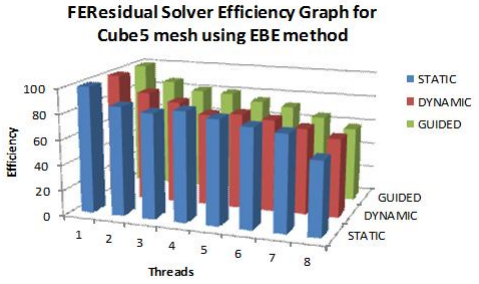



(b) Efficiency graph

Figure 7.11: The data is collected for the 3D mesh with 411939 DOFs on Intel X5560 by FEResidual Version 6 (EBE method). It displays timing data for the FEResidual method.



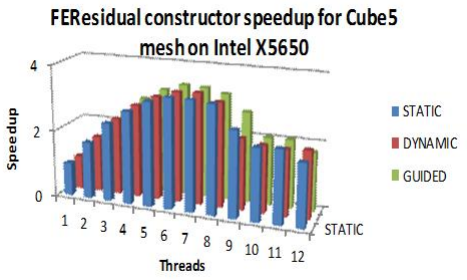

(a) SpeedUp graph



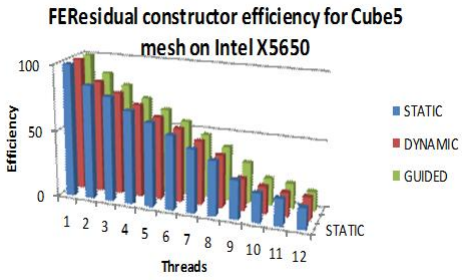

(b) Efficiency graph

Figure 7.12: The data is collected for the 3D mesh with 411939 DOFs on Intel X5560 by FEResidual Version 6 (EBE method). It displays data for the FEResidual solution (construction time + 100 residual iterations time).

Now the 3D Elasticity problem is executed on the Intel X5650 processor for comparison. Figures 7.13a and 7.13b are the FEResidual EBE library constructor speedup and efficiency graphs. The constructor has achieved speedup of 3 till 7 threads. From 8 onward threads the speedup decreases. The constructor efficiency reduces with the rise in number of threads.



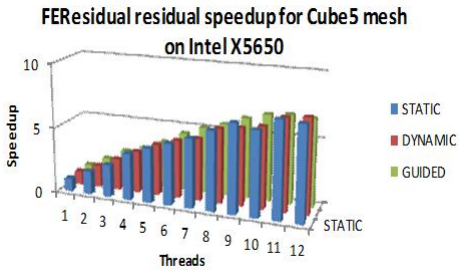

(a) SpeedUp graph



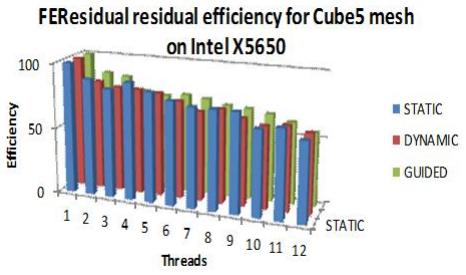

(b) Efficiency graph

Figure 7.13: The data is collected for the 3D mesh with 411939 DOFs on Intel X5650 by FEResidual Version 6 (EBE method). It displays data for the FEResidual constructor.

Figures 7.14a and 7.14b represent the speedup and efficiency graph for the 100 residual iterations of FEResidual residual method. Figure 7.14a represents the constant increase in speedup for the residual method. It has achieved maximum speedup of 6. Figure 7.14b represents the efficiency which slowly decreases with the increase in threads. The lowest efficiency is about 60 percent for 12 threads.




(a) SpeedUp graph

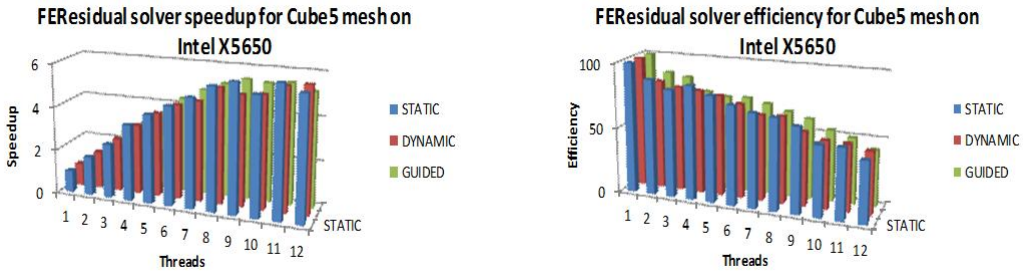



(b) Efficiency graph

Figure 7.14: The data is collected for the 3D mesh with 411939 DOFs on Intel X5650 by FEResidual Version 6 (EBE method). It displays timing data for the FEResidual method.

Finally, Figures 7.15a and 7.15b depicts the speedup and efficiency graphs for the 3D Elasticity problem on the Intel X5650 processor. The FEResidual has achieved speedup till 8 threads after it almost constants. Its has reached the max speedup of 6. The efficiency has reduced with increase in threads. The 3D Elasticity has

achieved speed of 6 as compared to the 2D Poisson problem speedup of 3.5. The same pattern can be observed for the FEResidual constructor and residual.




(a) SpeedUp graph (b) Efficiency graph

Figure 7.15: The data is collected for the 3D mesh with 411939 DOFs on Intel X5650 by FEResidual Version 6 (EBE method). It displays data for the FEResidual solution (construction time + 100 residual iterations time).

During experiments the Intel XEON X5560 processor (quad core) is used. The processor provides eight threads using Hyper Threading technology. It is observed from the experiments that the memory intensive tasks like residual calculation performs much better with Hyper Threading technology. It can be seen in the residual calculation charts given in Figures 7.6a, 7.8a, 7.11a and 7.14a. The multiple threads allow the efficient usage of cores as if one of the threads is waiting for data, other threads with data can use the core.

The experiments depict that the constructor speedup gained is limited to the number of cores in the processors. During data assembly for each element its data container are constructed. The acquisition of the memory efficiency is dependent on the Memory Management Unit (MMU). When the number of threads increases the number of cores the requests to the MMU also increases. The MMU cannot conveniently serve all the threads memory requests and some of the threads have to wait for the response. The FEDomain constructor speedup for Intel® XEON 5560 processor is shown in Figures 7.5a and 7.10a. For Intel® XEON 5650 processor is shown in 7.7a and 7.13a. The speedup reduces as the number of threads increases

the available number of cores.

## 7.3 FEResidual Version TBB

In this section the FEResidual EBE method parallelism is implemented using Intel Thread Building Blocks. TBB has been introduced in Section 1.5.

### 7.3.1 Implementation

The EBE method parallel implementation using TBB library is added into FEResidual library as a FEResidual_TBB class. The TBB provides loop parallelisation structures like parallel_for, parallel_reduce. These parallelisation structures implement the number of loops structures simultaneously without interfering with each other. The tbb::parallel_for template function breaks the iteration space into chunks and runs each chunk on a separate thread. The parallel_for is initialized by implementing the body object as shown in Listing 7.3, in which the operator() is a loop body which processes a chunk. The iteration space is represented by a one dimension range object *blocked_range* provided by the TBB.

```
template <typename DVec, typename DMat, typename SMat>
class Accumulate{
private:
    size_t sDoF;
    vector<Element>* p_E; map<size_t , double>* p_Constrain;
    LOAD* p_L; STIFFNESS* p_S; CONNECTIVITY*  p_C;
public:
    Accumulate(el_cont* p_e, LOAD_DATA* p_l, STIF_DATA* p_s, CONN_DATA* p_c,
            map<size_t , double>* constr, size_t S_DOFs):p_E(p_e), p_L(p_l),
            p_S(p_s), p_C(p_c),p_Constrain(constr),sDoF(S_DOFs){};
        void operator()( const tbb::blocked_range<int>& range) const {
            multimap<size_t , size_t> m_D;
            multimap<size_t , size_t >::iterator it;
            for (size_t i=range.begin(); i!= range.end(); i++){
```

```
                size_t eDoF = (*p_E)[i]->get_DOFs_count();
                DVec* L = new DVec(eDoF);
                DMat* S = new DMat(eDoF,eDoF);
                SMat* C = new SMat(eDoF,sDoF);
                (*p_E)[i]->getLoad((*L));           (*p_L)[i] = L;
                (*p_E)[i]->getStiffness((*S));      (*p_S)[i] = S;
                (*p_E)[i]->getConnectivity((*C)); (*p_C)[i] = C;
                C->apply_constrain((*S),(*L),(*p_Constrain));
            }
        };
    };
```

Listing 7.3: Parallel For example

The loop object is invoked by the parallel_for construct in accumateParallel as shown in Listing 7.4. The parallel_for requires the range object (blocked_range) and functor to loop body (accu) in Listing 7.4.

```
void accumulateParallel( vector<TElement>& Ele_Vector,
                         map<size_t, double>& Constrains)
{
    Accumulate<DVector,DMatrix,SMatrix> accu(&Ele_Vector, &v_L, &v_S,
                                        &v_C, &Constrains, sys_total_DOFs);
    tbb::parallel_for(blocked_range<int>(0, Ele_Vector.size(),
                                                    chunk_size), accu);
}
```

Listing 7.4: Calling Parallel_For

The blocked_range constructor requires 3 arguments. The first argument is the index of the first and the second object is the index of the last object. The third argument is a chunk_size, also known as a grain_size, and specifies the number of iterations for a chunk. If the iteration space is greater than the grain_size iterations, parallel_for splits it into separate sub ranges, that are scheduled separately. The grain_size has to be tuned for every problem size. A too smaller grain_size leads to

relatively high proportion of overhead and for large grain_size reduces this proportion, at the cost of reducing potential parallelism. The overhead as a fraction of useful work depends on the grain_size and not on the number of chunks. The TBB provides auto_partitioner class to automatically choose grain_size heuristically and allows user not to specify the grain_size. The auto_partitioner is an adaptive partitioner that limits the number of splits needed for load balancing by reacting to work-stealing events. It creates additional sub ranges only if threads are actively stealing work.

In FEResidual_TBB class the assembling of data, residual vector computation and data deletion are implemented in parallel. In FEResidual EBE method the work load depends upon the number of elements, as in the data assembling phase elements data can be collected in parallel. The TBB residual method is selected by setting the solver method parameter to RESIDUAL_TBB in the FEDomain constructor. The FEDomain and element interface are not modified and only a new residual method TBB is added into method types.

### 7.3.2 Performance

For TBB there are three variables threads, grain size, problem size, and problem type. For computing the preformance of TBB residual application three meshes are used Cube4, Cube5, and Quater9 and the properties of these meshes are in Table 7.6. The TBB residual is tested using the number of available threads provided by the processing core and the appropriate thread size. The maximum number of threads is computed using the function provided by $tbb :: task\_scheduler\_init :: default\_num\_threads()$. The chunk size increases logarithmically so the chunk size used is in range from 1 to 100000 as shown in Figures 7.16 and 7.17. Both the figures show that, for each grain size, the best results were achieved when the maximum number of threads was used. The smallest grain size using 1 thread has the slowest run time. In Figure 7.16 as the grain size is greater than or equal to the

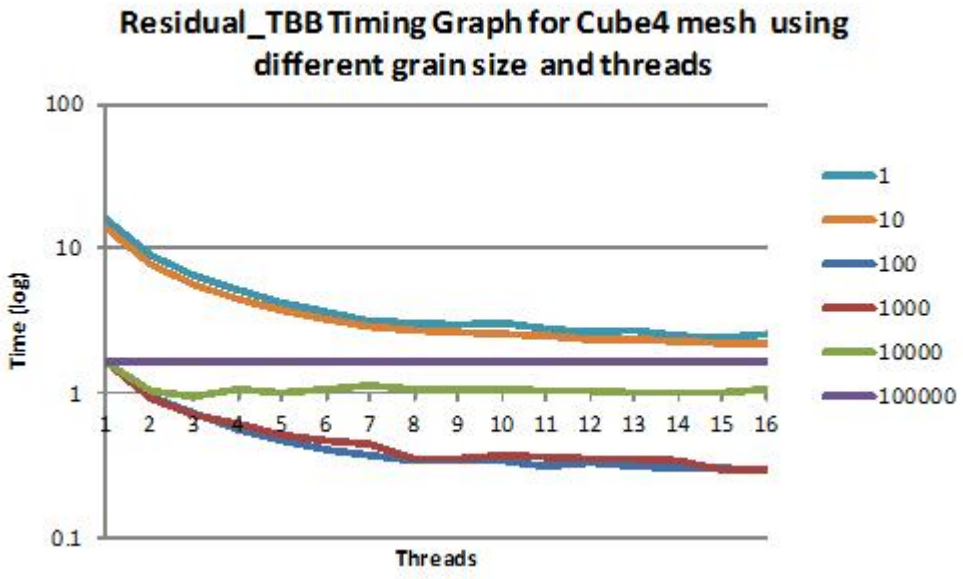



Figure 7.16: Residual TBB timing for Cube4 mesh.

number of elements the FEResidual run time becomes constant. For Cube4, 1000 is most optimized grain size, and for Cube5 1000 and 10000 have similar execution timing. In FEResidual TBB, the number of threads is fixed to the maximum available threads. The chunk size grows as the number of elements in the mesh grows so for the Cube4 mesh with 13920 elements has 1000 grain size while for Cube5 mesh the optimized grain size is 10000. For the 2D meshes Quater9 the grain size does not grow as in case of 3D Elasticity meshes as from the Figure 7.18 the most optimized grain size is 100 while the DOFs and the number of elements is greater than Cube4 and Cube5. The grain size has to be optimized for problems.

| Mesh | Problem_Type | DOFs | Elements |
|---|---|---|---|
| Cube4 | Elasticity 3D | 7370 | 13920 |
| Cube5 | Elasticity 3D | 53906 | 104640 |
| Quater9 | Poisson 2D | 920064 | 1840128 |

Table 7.6: Mesh Properties

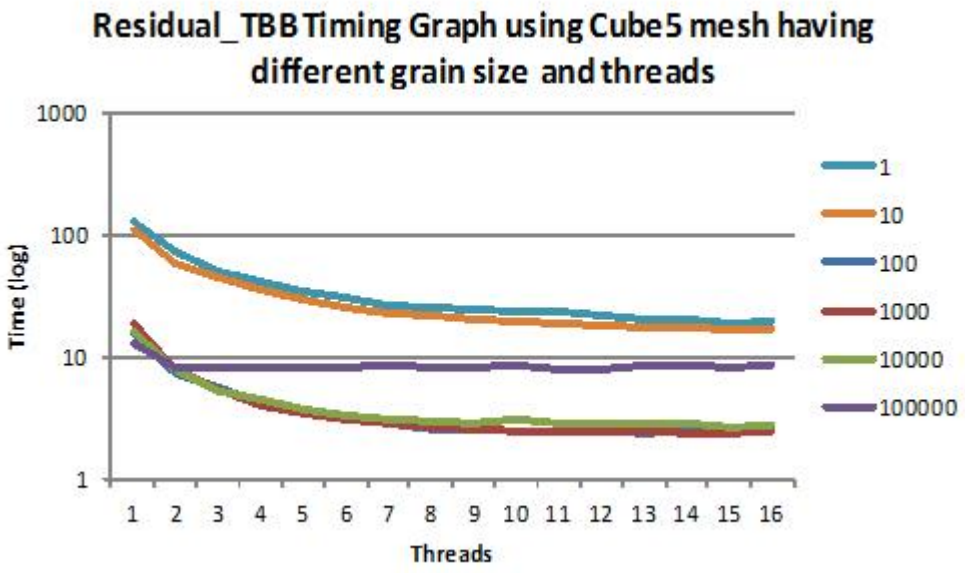



Figure 7.17: Residual TBB timing for Cube5 mesh.

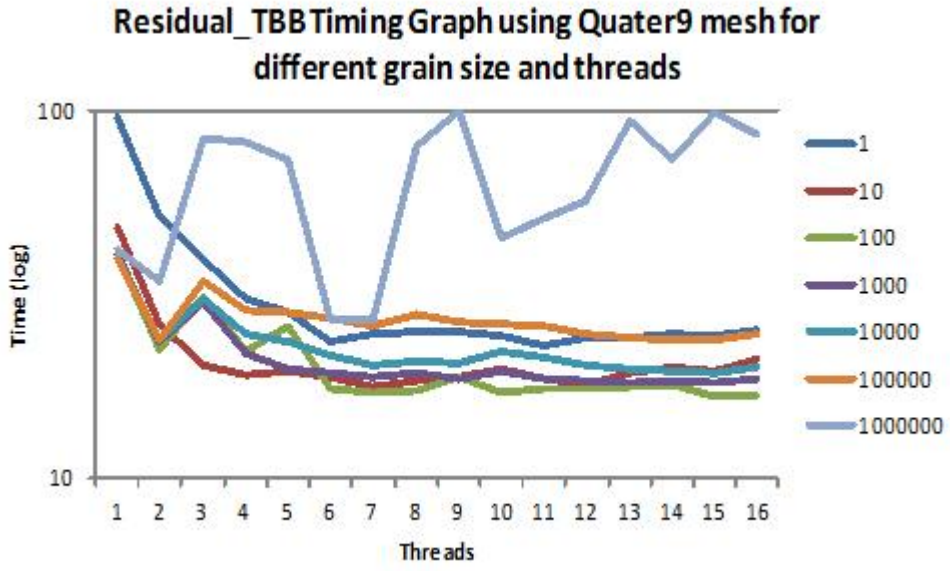



Figure 7.18: Residual TBB timing for Quater9 mesh.

As mentioned earlier that TBB provides simple_partitioner and auto_partitioner classes to implement different algorithms of load balancing among the threads. In simple_partitioner the threads are allocated with a fix sized load chunk to all the threads except from the last chunk and work stealing is not allowed. While for auto_partitioner task scheduler dynamically vary chunk size according the work stealing. Figures 7.19 and 7.20 represent the timing and speedup of these three meshes using both partitioner methods. Figure 7.19 shows the timing information of these threads using maximum threads. As long as the grain size is ten times less than the total number of elements, the execution time remains almost the same. The difference of the timing is negligible. The auto_partitioner has performed better than simple_partitioner for cube4 and quater9 mesh but for cube5 simple_partitioner has performed better. The difference for both partitioners timing as speedup is small. The FEResidual_TBB is implemented using auto_partitioner while keeping grain_size as tunable parameter which is problem size and type dependent variable.

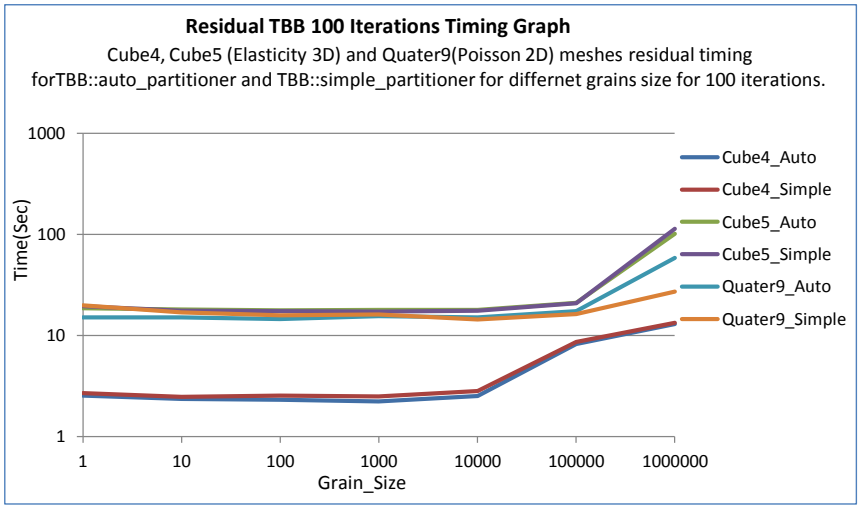



Figure 7.19: Execution time

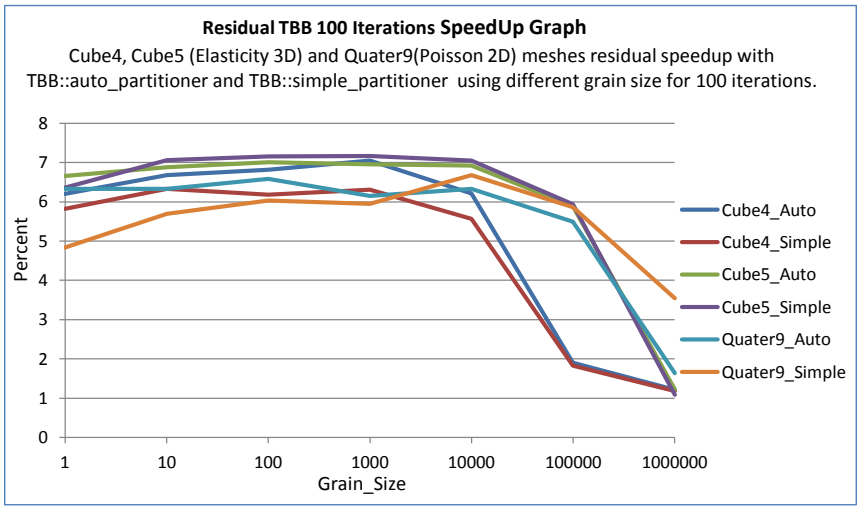



Figure 7.20: Speedup graph

## 7.4 Conclusion

In this chapter the shared memory version of FEDomain residual libraries are discussed. The RESIDUAL_FA and RESIDUAL_EBE methods are implemented using OpenMP while FERESIDUAL_TBB is implemented using TBB library. In next chapter, the distributed memory implementations of the residual methods are discussed.

# Chapter 8

# FEDomain Distributed Memory Residual Methods

The Distributed Memory Residual Methods (DMRM) is a set of residual vector computation methods implemented to calculate a residual vector on a distributed memory architecture. These are aimed for the iterative solver methods which are targeted at computing the finite element solution on the distributed memory architectures. In this chapter the FA and EBE methods are implemented for the distribute memory architectures. As it has been discussed in Chapter 5, due to the limitations of C++ language the FEDomain library cannot partition the provided mesh. The DMRM requires the user to provide the partitioned mesh. Each provided element should be present in only one mesh partition and should be aware of its partition id. The mesh partitioning criteria are discussed in section 2.1.4.

The DMRM is implemented as the part of a FEDomainMPI package discussed in Chapter 5. The user can select the FA or EBE method by setting the DISTRIBUTION_SOLUTION_METHOD to DIS_RESDUAL_FA or DIS_RESIDUAL_EBE respectively at the FEDomainMPI constructor. The FEDomainMPI constructor takes partition ids as a std::vector object so that more than one mesh partition

can be allocated to each MPI process. The users are allowed to provide all the mesh elements to all the MPI processes. All the FEDomainMPI objects filter their allocated mesh element objects to collect allocated partition's elements. Each FEDomainMPI object performs the residual calculation for allocated partition's elements. In setSolution function the DOFs solutions are set to every provided element. The getResidual method for all the MPI process FEDomainMPI object take the $\alpha$ and provides the residual vector $\vec{r}$ in global DOF numbering $\mathcal{N}$.

For the DMRM solver the FEDomainMPI class getResidual function requires the user to provide the global approximate solution $x_\alpha$. The users have to at least provide the solution for the DOFs present in the allocated partitions to each FEDomainMPI object. The allocated DOFs are represented by $\mathbf{\Lambda_P}$ but the solution should be mapped in $\mathcal{N}$. The user should at least provide $\mathbf{\Lambda_P^T \Lambda_P} x_\alpha$ which will be a vector of same size as $\vec{r}$ but will only have solutions for the DOFs present in the allocated partitions. The getResidual function provides the $\vec{r}$ to all the DMRM objects. The DMRM does not have knowledge of the user implemented iterative algorithm requirements. It is preferred that in each MPI process the DMRM FEDomainMPI object will provide $\vec{r}$ to the user. Each MPI process has a single FEDomainMPI object. In FEDomainMPI object all its allocated mesh partitions are considered as a single partition. The DMRM objects provide their data ($\boldsymbol{A_P}$, $\vec{b_P}$ and $\mathbf{\Lambda_P}$) in $\mathcal{N}_\mathcal{P}$ numbering. The FEDomainMPI::getResidual function could have provided $\mathbf{\Lambda_P^T \Lambda_P} \vec{r}$ but during design process it was considered that the user with no experience of MPI communication may perform all its approximate solution calculations on a single or on all the processes. This design decision allows the naive DMRM user to convert their current code (non distributed memory finite element codes) into parallel applications.

In FEDomainMPI package the DMRM solvers are implemented as FEResidualMPI class shown in Listing 8.1. The FEResidualMPI class is a MPI wrapper

around FEResidualFA and FEResidualEBE class for the distributed FA and EBE solver. This class has to perform the communication on behalf of their residual classes objects. In DMRM the set of elements allocated to each MPI process is considered as one element. Each MPI process DMRM object collects allocated partitions elements data ($\boldsymbol{A_K}$ and $\vec{b_K}$) and stores the elements data into process internal DOFs ids $\mathcal{N}_P$ as $\boldsymbol{A_P}$ and $\vec{b_P}$ as shown in (2.47) and (2.55). The global residual vector $\vec{r}$ is computed by summing all the FEResiudalMPI objects' $\vec{r_P}$ as shown in (8.1). The FEResidualMPI object's residual vector $\vec{r_P}$ for the FA method is calculated for each partition through (8.2), and stored in using $\mathcal{N}_\mathcal{P}$. The global residual is calculated through (8.3).

$$r = \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{\Lambda}_p^T r_p \tag{8.1}$$

$$\text{where} \quad r_p = \vec{b}_p - \boldsymbol{A}_p \boldsymbol{\Lambda}_p x \tag{8.2}$$

$$\text{then} \quad r = \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{\Lambda}_p^T (\vec{b}_p - \boldsymbol{A}_p \boldsymbol{\Lambda}_p x) \tag{8.3}$$

```
class FEResidualMPI : public FESystem
{
public:
    FEResidualMPI(std::vector<FEElement*>& Elements,
                  FE_UINT& sys_total_dofs,
                  FE_UINT& max_element_dofs,
                  std::vector<FE_UINT>& part_ids,
                  std::map<FE_UINT,FE_DATA>& dirichlet_ids,
                  DISTRIBUTED_SOLVER_TYPE solver_type);
    ~FEResidualMPI();
    FE_UINT get_dofs_count();
    FE_UINT getPartitionId();
    void getLoad(FEVector &vector);
    void getStiffness(FEDenseMatrix &matrix);
```

```
    void getSystem(FE_equation &equation);
    void getConnectivity(std::vector<FE_UINT> &dof_ids);
    void setSolution(std::vector<FE_DATA> &dirichlet_vals);
    void getResidual(const std::vector<FE_DATA> &app_solution,
                           std::vector<FE_DATA> &residual);
};
```

Listing 8.1: FEResidualMPI Interface

## 8.1 Distributed EBE Residual

The EBE residual method already discussed in section 2.2.2 calculates the element residual vector $r_k$ at the element level using (2.39). The DMRM EBE method has to collect his allocated partitions elements data ($\boldsymbol{A}_K$ and $\vec{b_K}$) for residual calculations. The elements' data can be stored in multiple formats, these formats are discussed in following paragraphs.

- In first design all the elements data ($\boldsymbol{A_K}$ and $\vec{b_K}$) are stored in DMRM in the dense matrix and vector, respectively. For the residual calculations the element data has to be mapped from $\mathcal{N}_\mathcal{K}$ to $\mathcal{N}_\mathcal{P}$. The user element contains the information about its DOFs global numbering. For a partitioned mesh, the elements should have information about their partition ids. The elements do not have information about the total DOFs in a partition and how to map their data from $\mathcal{N}_\mathcal{K}$ to $\mathcal{N}_\mathcal{P}$. The elements' data can be converted into the partition $\mathcal{N}_\mathcal{P}$ by two methods:
  - Either the element data should be firstly converted from $\mathcal{N}_\mathcal{K}$ to $\mathcal{N}$ and then convert the mapped data into partition's DOF numbering $\mathcal{N}_\mathcal{P}$.
  - or, for each element the mapping matrix $\boldsymbol{\Lambda_{KP}}$ (from $\mathcal{N}_\mathcal{K}$ to $\mathcal{N}_\mathcal{P}$) is calculated using (8.4) and stored with element data. The computation of $\boldsymbol{\Lambda}_{KP}$ is computationally expensive. Suppose a partitioned mesh has

10 DOFs, one of its partition has 7 DOFs and contains an element with 3 DOFs. The element's $\mathbf{\Lambda}_K$ is a $3 \times 10$ matrix with three not zero entries and the partition's $\mathbf{\Lambda}_P$ is a $7 \times 10$ matrix with seven non zero entries. The $\mathbf{\Lambda}_{KP}$ will be $3 \times 7$ matrix with three non zero entries. The computation of $\mathbf{\Lambda}_{KP}$ involves 21 multiplications and only 3 of which yield non zero results which will be stored. 85% of the multiplications will be wastage of the computational resource and processing time. This design has to store all the allocated partitions elements data inside the DMRM EBE solver after mapping the data from the $\mathcal{N}_K$ to $\mathcal{N}_P$. The elements data have to be stored into the sparse data structures. If the $\vec{r}_P$ is calculated using (8.5), then $\boldsymbol{\Lambda_{KP}}$ is calculated for every element in every iteration while (8.6) represents the send methods in which the $\boldsymbol{\Lambda_{KP}}$ are calculated once and stored to be used in later residual iterations. This is:

$$\mathbf{\Lambda}_{KP} = \mathbf{\Lambda}_K \mathbf{\Lambda}_P^T \tag{8.4}$$

$$r_p = \sum_{K \in \mathcal{P}_i} (\mathbf{\Lambda}_P \mathbf{\Lambda}_K^T \vec{b_K} - (\mathbf{\Lambda}_P \mathbf{\Lambda}_K^T \boldsymbol{A}_K \mathbf{\Lambda}_K \mathbf{\Lambda}_P^T) \mathbf{\Lambda}_P x) \tag{8.5}$$

$$r_p = \sum_{K \in \mathcal{P}_i} (\mathbf{\Lambda}_{KP}^T \vec{b_K} - \mathbf{\Lambda}_{KP}^T \boldsymbol{A}_K \mathbf{\Lambda}_{KP} \mathbf{\Lambda}_P x) \tag{8.6}$$

- In the second design the elements data is stored in DMRM objects after applying the Dirichlet constraints and being mapped into $\mathcal{N}_\mathcal{P}$. For each element in a partition, $\mathbf{\Lambda}_{KP}^T \boldsymbol{A}_K \mathbf{\Lambda}_{KP}$ and $\mathbf{\Lambda}_{KP}^T \vec{b}_K$ are stored as stiffness matrix and load vector, respectively. The $\mathbf{\Lambda}_{KP}^T \boldsymbol{A}_K \mathbf{\Lambda}_{KP}$ is a sparse matrix of dimension $N_P \times N_P$ with maximum of $N_K^2$ non zero entries see (8.7) below. The $\mathbf{\Lambda}_{KP}^T \vec{b}_K$ is a sparse vector of $N_P$ dimensions and $N_K$ non zero entries see (8.8) below. This design allows the DMRM objects to calculate $\mathbf{\Lambda}_{KP}$ once for each element using (8.4). Then $\vec{r}_p$ is calculated using (8.9).

$$\boldsymbol{A}_{K_P} = \boldsymbol{\Lambda}_{KP}^T \boldsymbol{A}_K \boldsymbol{\Lambda}_{KP} \tag{8.7}$$

$$\vec{b_{K_P}} = \boldsymbol{\Lambda}_{KP}^T \vec{b}_K \tag{8.8}$$

$$r_p = \sum_{K \in P_i} (\vec{b_{K_P}} - \boldsymbol{A}_{K_P} \boldsymbol{\Lambda}_P x) \tag{8.9}$$

- The additional computation due to sparse matrix multiplication can be avoided by storing the elements data in $\mathcal{N}_K$ numbering. The FEDomain-MPI obtains elements data in dense data structures like dense matrix and vector. These data structures are memory and computationally efficient. The $\boldsymbol{A}_K$ and $\vec{b}_K$ have to be stored in the DMRM solver as their data may change after applying Dirichlet constraints. The elements connectivity data remain constant and always remain unmodified so it can be obtained from the elements at any time. In (8.10), the elements residual vector can be directly calculated at the element level and firstly stored in $\mathcal{N}$, but provides residual vector after mapping in partitions' $\mathcal{N}_\mathcal{P}$ numbering.

$$r_p = \boldsymbol{\Lambda}_P \sum_{K \in \mathcal{P}_i} \boldsymbol{\Lambda}_K^T (\vec{b}_K - \boldsymbol{A}_K \boldsymbol{\Lambda}_K x) \tag{8.10}$$

  This design enabled the DMRM EBE solver to store minimum amount of data during execution and avoid unnecessary computation by omitting sparse matrix by sparse matrix multiplications. The (8.1) calculate the global residual vector by mapping and adding each processes residual vector after mapping from these from $\mathcal{N}_\mathcal{P}$ to $\mathcal{N}$ numbering.

## 8.2 Distributed FA Residual

The DMRM solver FA residual mode is selected by setting the DISTRIBUTION_SOLUTION_METHOD to DIS_RESIDUAL_FA. The FA residual method

has been discussed in section 2.2.2.1. In DMRM FA residual method every MPI process will be allocated a single FEResidualMPI object. The user can allocate each FEResidualMPI object to more than one mesh partition, and each FEResidualMPI object treats all its allocated partitions as a single partition. The mesh partitions should be allocated to only one of the MPI processes. The FEResidualMPI object is treated as an element object which will provide its data in partitions numbering $\mathcal{N}_{\mathcal{P}}$. Each of the FEResidualMPI objects in FA mode will allocate its elements' data into its stiffness matrix $\boldsymbol{A}_P$ and load vector $\vec{b}_P$. The element data is mapped into partition data as follows.

$$\boldsymbol{A}_P = \sum_{K \in P_i} \boldsymbol{\Lambda}_P \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K \boldsymbol{\Lambda}_P^T \tag{8.11}$$

$$\vec{b}_P = \sum_{K \in P_i} \boldsymbol{\Lambda}_P \boldsymbol{\Lambda}_K^T \vec{b}_K \tag{8.12}$$

The FEResidualMPI for FA method calculates the residual vector through (8.13) below. For each iteration the $\boldsymbol{A}_P$ and $\vec{b}_P$ remains unchanged and these are stored after applying Dirichlet constraints.

$$r_P = \vec{b}_P - \boldsymbol{A}_P \boldsymbol{\Lambda}_P x_\alpha. \tag{8.13}$$

The residual vector $\vec{r}$ is computed by adding all MPI process residual vector $\vec{r}_P$ after mapping these from $\mathcal{N}_P$ to $\mathcal{N}$. During implementation it is observed that significant time is being spent in transferring the element data between different mapping systems, especially in acquiring data from elements and setting solution to the elements. To reduce the mapping time it is decided that data inside DMRM FA FEResidualMPI object should be stored using global numbering $\mathcal{N}$. In each FEResidualMPI object the partition stiffness matrix $\boldsymbol{A}_P$ and load vector $\vec{b}_P$ are actually stored as $\boldsymbol{\Lambda}_P^T \boldsymbol{A}_P \boldsymbol{\Lambda}_P$ and $\boldsymbol{\Lambda}_P^T \vec{b}_P$, respectively. The stiffness matrix and load

vector are computed as follows:

$$\boldsymbol{\Lambda}_P^T \boldsymbol{A}_P \boldsymbol{\Lambda}_P = \sum_{K \in P_i} \boldsymbol{\Lambda}_K^T \boldsymbol{A}_K \boldsymbol{\Lambda}_K \tag{8.14}$$

$$\boldsymbol{\Lambda}_P^T \vec{b}_P = \sum_{K \in P_i} \boldsymbol{\Lambda}_K^T \vec{b}_K \tag{8.15}$$

Storing the partition in $\mathcal{N}$ has advantages as the data stored in the partition stiffness matrix remains the same. The drawback is that the load vector can not be stored as a dense vector, it has to be stored as a sparse vector.

In the DIS_RESIDUAL_FA solver the *getResidual*() function of every FEResidualMPI object will receive two std::vector objects of size $N$ from the user. Te first parameter object represents the current approximate solution provided by the user. The other object represents the residual vector. The FEResidualMPI object uses the second object to provide the residual vector to the user. These vectors represent their data in global DOF numbering $\mathcal{N}$. The *getResidual*() function implements the following equation.

$$\boldsymbol{\Lambda}_P^T \vec{r}_P = \boldsymbol{\Lambda}_P^T \vec{b}_P - \boldsymbol{\Lambda}_P^T \boldsymbol{A}_P \boldsymbol{\Lambda}_P x_\alpha \tag{8.16}$$

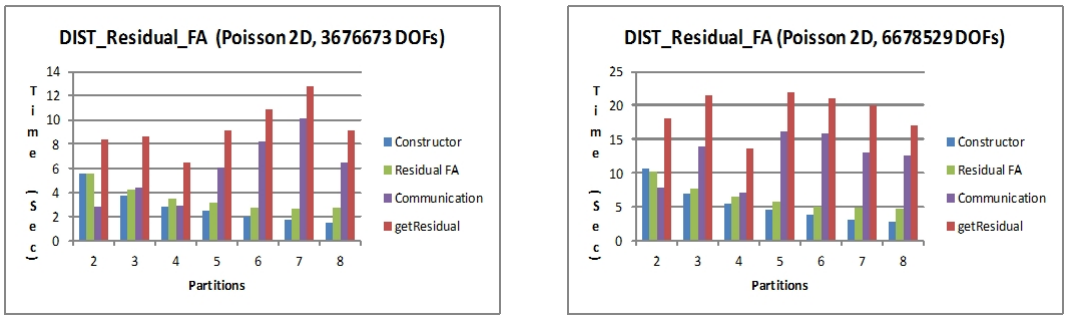
The residual $\vec{r}$ is computed by adding all the FEResidualMPI objects' residual vectors $\boldsymbol{\Lambda}_P^T \vec{r}_P$ as shown in (8.1). All the $\boldsymbol{\Lambda}_P^T \vec{r}_P$ are present in different MPI processes where these processes cannot directly access each other's memory. There are two methods to calculate $\vec{r}$:

- either, all the FEResidualMPI objects $\boldsymbol{\Lambda}_P^T \vec{r}_P$ are gathered on a single MPI process and are added to compute $\vec{r}$. $\vec{r}$ is then broadcast to all other MPI processes. This method will compute the global values of the DOFs which are not Dirichlet DOFs and present in more than one partition.

- or, (8.1) can be implemented by using *MPI_Allreduce* with *MPI_SUM* operator. This method will involve less book keeping. This method is implemented

in the FEResidualMPI DIS_RESIDUAl_FA method.

Table 8.1 contains the timing of the FEDomainMPI DIS_RESIDUAL_FA for the Poisson 2D problem. The table contains the timing data for the meshes with the 3 million and 6 million DOFs. It is observed that the constructor time has reduced with rise in the number of partitions. The third column in Table 8.1 contains the time consumed by the getResidual method for 101 iterations. The time consumed for the getResidual method has changed randomly for the partitions. The fourth and fifth columns in Table 8.1 show the time consumptions for residual calculation and data communication, respectively. In fourth column, the residual time consumption has reduced with the increase in the number of partitions. In fifth column depicts the MPI_Allreduce time consumption which changes randomly. The DIST_Random_FA solver behaviours for 2D problem can be seen in Figures 8.1a and 8.1b.

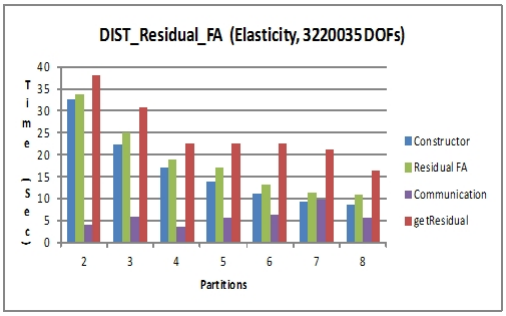



(a) Mesh with 3,676,673 DOFs (b) Mesh with 6,678,529 DOFs

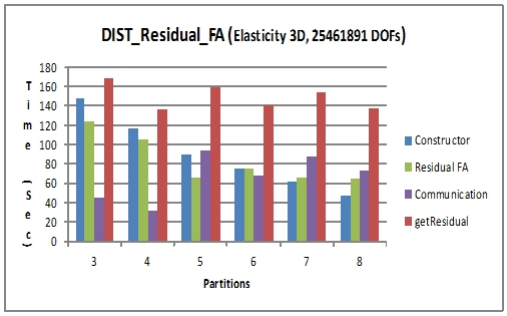
Figure 8.1: The charts display the timing of the DIST_Residual_FA solver for the Poisson 2D problem. Two meshes are used to calculate the construction time and 100 residual iteration time consumption. The getResidual time is sum of the residual time and communication time.

| Processes | Constructor Time | Residual Time | Residual FA | Communication |
|---|---|---|---|---|
| DIS_RESIDUAL_FA (DOFs = 3,676,673, Poisson2D) | | | | |
| 2 | 5.53 | 8.40 | 5.53 | 2.87 |
| 3 | 3.70 | 8.63 | 4.18 | 4.44 |
| 4 | 2.81 | 6.47 | 3.54 | 2.93 |
| 5 | 2.42 | 9.11 | 3.10 | 6.01 |
| 6 | 2.00 | 10.91 | 2.75 | 8.16 |
| 7 | 1.79 | 12.74 | 2.63 | 10.11 |
| 8 | 1.51 | 9.14 | 2.75 | 6.42 |
| DIS_RESIDUAL_FA (DOFs = 6,678,529, Poisson2D) | | | | |
| 2 | 10.61 | 18.01 | 10.26 | 7.85 |
| 3 | 6.93 | 21.53 | 7.66 | 13.90 |
| 4 | 5.45 | 13.52 | 6.43 | 7.09 |
| 5 | 4.49 | 21.89 | 5.71 | 16.19 |
| 6 | 3.79 | 20.97 | 5.14 | 15.83 |
| 7 | 3.25 | 19.94 | 4.93 | 13.02 |
| 8 | 2.91 | 17.10 | 4.70 | 12.56 |

Table 8.1: The FEDomainMPI DIS_RESIDUAL_FA solver timing results for the constructor and 101 residual iterations. The FEDomainMPI DIS_RESIDUAL_FA solver is executed for the Poisson 2D problem.




(a) Mesh with 3,220,035 DOFs




(b) Mesh with 25,461,891 DOFs

Figure 8.2: The figures display the time consumption of the DIST_Residual_FA solver for the Elasticity 3D problem. Two meshes are used to calculate the construction time and 100 residual iterations time consumption. The getResidual time is sum of the residual calculation time and communication time.

| MPI Processes | Assembly Time | Residual Time | Residual Full Assembly | Communication Time | Assembly Speed Up | Residual Speed Up |
|---|---|---|---|---|---|---|
| DIS_RESIDUAL_FA (DOFs = 3,220,035, Elasticity 3D) | | | | | | |
| 2 | 32.40 | 37.94 | 33.71 | 4.22 | 1.0 | 1.0 |
| 3 | 22.23 | 30.77 | 24.93 | 5.85 | 1.5 | 1.7 |
| 4 | 16.83 | 22.51 | 19.02 | 3.49 | 1.9 | 1.7 |
| 5 | 13.85 | 22.49 | 16.96 | 5.54 | 2.3 | 1.7 |
| 6 | 11.17 | 22.48 | 13.12 | 6.25 | 2.9 | 1.7 |
| 7 | 9.36 | 21.06 | 11.51 | 9.55 | 3.5 | 1.8 |
| 8 | 8.53 | 16.51 | 10.98 | 5.53 | 3.8 | 2.3 |
| DIS_RESIDUAL_FA (DOFs = 25,461,891, Elasticity 3D) | | | | | | |
| 3 | 147.93 | 168.93 | 123.50 | 45.53 | 1.0 | 1.0 |
| 4 | 116.89 | 135.89 | 104.83 | 31.50 | 1.3 | 1.2 |
| 5 | 90.68 | 159.86 | 66.30 | 93.58 | 1.6 | 1.1 |
| 6 | 74.90 | 140.84 | 75.38 | 68.48 | 2.0 | 1.2 |
| 7 | 61.37 | 153.71 | 66.28 | 87.42 | 2.4 | 1.1 |
| 8 | 47.13 | 137.43 | 64.68 | 72.75 | 3.1 | 1.2 |

Table 8.2: The FEDomainMPI DIS_RESIDUAL_FA solver timing results for the constructor and 100 residual iterations. The FEDomainMPI DIS_RESIDUAL_FA solver is executed for the Elasticity 3D problem.

Table 8.2 contains the timing of the FEDomainMPI DIS_RESIDUAL_FA for the Elasticity 3D problem. The table contains the timing data for the meshes with the 3 million and 25 million DOFs. It is observed that the constructor timing (given in second column) has reduced with the increase in the number of partitions. The third column in Table 8.2 contains the time consumed by the getResidual method for 101 iterations. The getResidual method time consumption has not reduced as is in second column. The fourth and fifth columns in Table 8.2 show the time consumptions for residual calculation and data communication, respectively. In fourth column the residual time consumption has reduced with the increase in the number of partitions. In fifth column depicts the MPI_Allreduce time consumption which changes randomly. The DIST_Random_FA solver behaviours for 3D problem can be seen in Figures 8.2a and 8.2b.

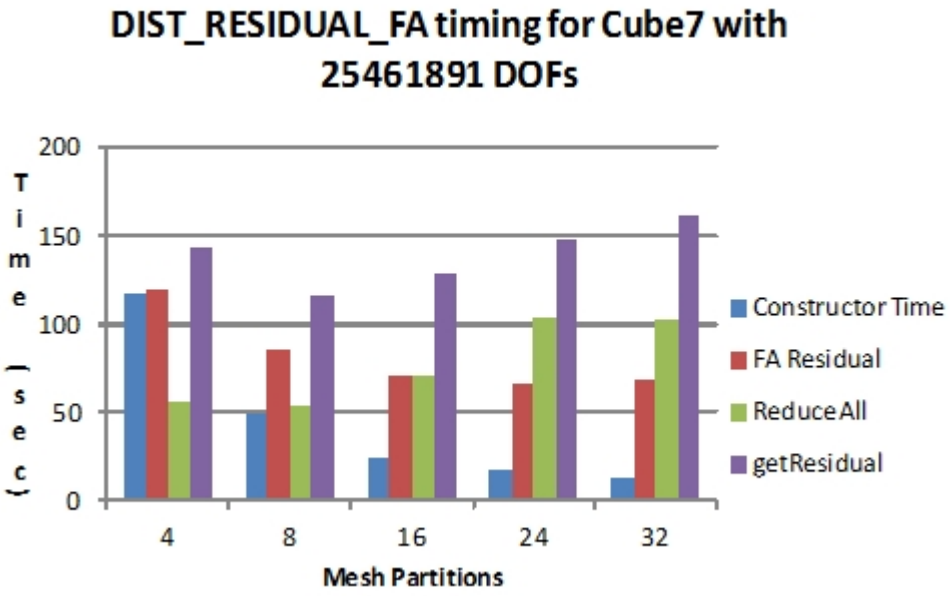



Figure 8.3: The DIS_RESIDUAL_FA solver time comparison for the mesh with 25,462,891 DOFs solved using different partitions.

Figure 8.3 depicts the DIS_Residual_FA solver constructor and 100 residual iterations timing for mesh with 25,462,891 DOFs. The solver has executed the mesh with 4,8,16,24, and 32 partitions. The ReduceAll time consumption in Figure 8.3 has increased with the increase in number of partitions. The getResidual time is directly proportional to the ReduceALL time consumption. In next section a new version of the distributed FA method is implemented to reduce MPI_Allreduce time consumption.

## 8.3 Distributed FA Compressed Residual

The Distributed FA Compressed Residual (DFCR) solver is selected by setting the DISTRIBUTION_SOLUTION_METHOD to DIS_RESIDUAL_FA_COMP in the FEDomainMPI constructor. This method is implemented after the FEEquation class is introduced in the FEDomain package. This method is selected for the advanced C++ application developer which has experience in parallel pro-

gramming and MPI library. The DFCR is implemented as the FEResidualFAMPI class which is inherited from the FESystem class to keep the standard interface. The DFCR is designed to keep the communication to a minimum among the MPI processes, by storing the data in the compressed format.

The DIS_RESIDUAL_FA_COMP solver classifies the mesh DOFs into two categories. The first category has the Dirichelet DOFs (dDOFs), the global number and values for these DOFs are provided by the user at construction time. The second category is the Internal DOFs (iDOFs), all the non Dirichlet DOFs come in this category. The iDOFs which are present in only in a single MPI process are called Process Internal DOFs (pIDOFs), and the iDOFs which are present in more than one MPI process are called Boundary DOFs (bDOFs). Each MPI process has a DIS_RESIDUAL_FA_COMP solver object which provides the residual vector $\vec{r}_p$, which represents the $pDOFs = (pIDOFs \cup bDOFs)$. The pIDOFs are present in one MPI partition and their residual calculation does not require input from other MPI processes. The bDOFs are present in more than one partitions and their residual values require contribution from all the MPI processes they belong to.

Suppose a mesh is partitioned into $N$ partitions. For each partition $P_i$ the internal DOFs are numbered consecutively and the boundary DOFs are numbered after all the partitions internal DOFs are numbered. For each partition, the stiffness matrix $\boldsymbol{A}_P$ and load vector $\vec{b}_P$ are given in (2.58) and (2.59), respectively.

The global residual vector $\vec{r}$ equation can be written in terms of the global system of equations for the mesh with $N$ partitions using (2.65) and can be written as:

$$\begin{bmatrix} \vec{r}_1 \\ \vdots \\ \vec{r}_N \\ \vec{r}_B \end{bmatrix} = \begin{bmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_N \\ \vec{b}_B \end{bmatrix} - \begin{bmatrix} \boldsymbol{A}_{11} & \dots & \mathbf{0} & \boldsymbol{A}_{1B} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \dots & \boldsymbol{A}_{NN} & \boldsymbol{A}_{NB} \\ \boldsymbol{A}_{B1} & \dots & \boldsymbol{A}_{BN} & \boldsymbol{A}_{BB} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \\ \alpha_B \end{bmatrix}, \tag{8.17}$$

where

$$\boldsymbol{A}_{BB} = \sum_{\mathcal{P}_j \in \mathcal{P}} \boldsymbol{A}_{BB}^{P_j}, \tag{8.18}$$

$$\vec{b}_B = \sum_{\mathcal{P}_j \in \mathcal{P}} \vec{b}_B^{P_j}. \tag{8.19}$$

The partition internal DOFs residual values are calculated as follows:

$$\vec{r_i} = \vec{b_i} - (\boldsymbol{A}_{ii}\alpha_i + \boldsymbol{A}_{iB}\alpha_B), \tag{8.20}$$

and the residual values for the boundary DOFs $\vec{r}_b$ are calculated as follows:

$$\begin{aligned} \vec{r}_B &= \vec{b}_B - (\boldsymbol{A}_{B1}\alpha_1 + .... + \boldsymbol{A}_{BN}\alpha_N + \boldsymbol{A}_{BB}\alpha_B) \\ &= \sum_{\mathcal{P}_i \in \mathcal{P}} \vec{b}_B^{P_i} - (\boldsymbol{A}_{B1}\alpha_1 + .... + \boldsymbol{A}_{BN}\alpha_N + \sum_{\mathcal{P}_i \in \mathcal{P}} \boldsymbol{A}_{BB}^{P_i}\alpha_B) \\ &= \sum_{\mathcal{P}_i \in \mathcal{P}} \vec{r}_B^{P_i} = \sum_{\mathcal{P}_i \in \mathcal{P}} (\vec{b}_B^{P_i} - \boldsymbol{A}_{Bi}\alpha_i - \boldsymbol{A}_{BB}^{P_i}\alpha_B). \end{aligned} \tag{8.21}$$

The partition's internal DOFs residual vector can be calculated independently of each other as can be seen in (8.20). The user can obtain the partitions' DOFs through the FEDomainMPI::getConnectivity method. The user has to provide the approximate solution for pDOFs for each partition. The FEDomainMPI object will provide the solution for pDOFs. For the FE_RESIDUAL_FA_COMP solver, the getResidual method takes two vectors of size #pDOFs (number of DOFs in pDOFs). The first vector is $\alpha_P$, provided by the user, and the second vector is $r_P$, used by the

solver to provide the residual vector to the user. The FE_RESIDUAL_FA_COMP solver is implemented after the FEEquation class into the FEDomain package. It is implemented as the FEResiduaFAMPI class in the FEDomain package. The interface of the FEReisudalFAMPI class is given in Listing 8.2. The $\vec{r}_B$ is calculated using (8.21), it requires input from all the FEResidualFAMPI objects. Each FEResidualFAMPI object calculates $\vec{r}_B^{P_i}$. The $\vec{r}_B$ values are computed using the *MPI_Allreduce* function with the *MPI_SUM* operator. This method enables us to reduce the amount of data being transferred among the MPI processes.

```
class FEResidualFAMPI : public FESystem{
    FEResidualFAMPI(std::vector<FEElement*>& Elements,
                    std::map<FE_UINT,FE_DATA>& Constraints,
                    FE_UINT &SysMaxDofs,
                    FE_UINT &EleMaxDofs,
                    std::vector<FE_UINT> &PartitionIDs,
                    MATRIX_TYPE MtrxType, SYSTEM_TYPE SysType);
    FE_UINT get_dofs_count();
    FE_UINT getPartitionId();
    void getLoad(FE_vector& v);
    void getStiffness(FE_dense_matrix& m);
    void getConnectivity(std::vector<FE_UINT>& v);
    void getSystem(FE_equation& e);
    void getResidual(const std::vector<FE_DATA>& approx_solution,
                           std::vector<FE_DATA>& residual_vector);
    void setSolution(std::vector<FE_DATA>& v);
};
```

Listing 8.2: FEResidualFAMPI Interface

The timing of FEResidualFAMPI solver is given in Table 8.3 for the Poisson 2D and Elasticity 3D problems. In FEResidualFAMPI the equation data is acquired from the user elements in construction stage and its timing is shown in the second column of Table 8.3. As the number of partitions increases, the number of elements allocated to single partitions reduces, and so does the construction time. The

third column represents the timing for the first residual iteration. In the first residual vector before the residual calculations the data is assembled smaller data containers given in (2.58) and (2.59). The third column has the timing for the first iteration. The fourth column represents the timing for the next 100 residual iterations, which includes the comparison if the data. The last column represents the time consumed by FEResidualFAMPI solver including constructor and 101 iterations. These timing are obtained using 8 distributed computational nodes. Each node had Intel®X5560 processor (4 cores and 8 threads).

| Processes Count | Constructor Time | 1st Residual Time | 100 Residuals Time | Execution Time | Execution Speed Up |
|---|---|---|---|---|---|
| DIS_RESIDUAL_FA_COMP (DOFs = 14699521, Poisson2D) | | | | | |
| 2 | 33.33 | 18.71 | 31.50 | 83.54 | 1.00 |
| 3 | 22.34 | 10.34 | 26.63 | 59.31 | 1.41 |
| 4 | 16.80 | 9.10 | 26.78 | 52.67 | 1.59 |
| 5 | 13.51 | 7.20 | 27.33 | 47.95 | 1.74 |
| 6 | 11.15 | 6.27 | 24.22 | 41.64 | 2.01 |
| 7 | 9.79 | 5.32 | 24.17 | 39.28 | 2.13 |
| 8 | 8.63 | 4.67 | 18.81 | 27.44 | 3.04 |
| DIS_RESIDUAL_FA_COMP (DOFs = 25461891, Elasticity3D) | | | | | |
| 2 | 262.40 | 51.26 | 150.60 | 464.26 | 1.00 |
| 3 | 213.58 | 42.31 | 136.38 | 392.26 | 1.18 |
| 4 | 164.25 | 30.80 | 103.80 | 301.84 | 1.54 |
| 5 | 141.24 | 25.35 | 101.37 | 267.96 | 1.73 |
| 6 | 103.83 | 20.66 | 63.87 | 191.36 | 2.43 |
| 7 | 91.29 | 21.01 | 61.91 | 174.21 | 2.66 |
| 8 | 65.77 | 13.50 | 49.65 | 128.92 | 3.60 |

Table 8.3: The FEDomainMPI DIS_RESIDUAL_FA_COMP solver timing results for the constructor and 101 residual iterations. The FEDomainMPI DIS_RESIDUAL_FA_COMP solver is executed for the Poisson 2D problem and Elasticity 3D problem.

The DIS_RESIDUAL_FA and DIS_RESIDUAL_FA_COMP for each partition the amount of stiffness matrix and load vector data are exactly the same. The differ-

ence lies in how the data is stored and the amount of data used in communication. The DIS_RESIDUAL_FA_COMP stores the data in the CSR format and only the bDOFs are used for the communication. The number of bDOFs increases with the number of partitions, but the number of pIDOFs reduces. The amount of time spent to calculate the residual for the pIDOFs reduces.

| Partition ID | Constructor Time | getResidual Time | Assembly Time | Residual Time | MPIReduceAll Time |
|---|---|---|---|---|---|
| DIS_RESIDUAL_FA_COMP (DOFs = 25461891, Elasticity3D) | | | | | |
| 1 | 165.93 | 128.51 | 30.21 | 96.25 | 20.18 |
| 2 | 167.12 | 124.24 | 25.72 | 77.07 | 21.44 |
| 3 | 172.37 | 124.68 | 26.19 | 77.20 | 21.28 |
| 4 | 170.58 | 130.43 | 18.63 | 54.51 | 57.28 |

Table 8.4: The table displays the timing for each partition FEResidalFAMPI object stages for the 4 partition cube mesh. Each row displays the timing step during construction and residual calculation stage. The last three columns have the timing data for the three tasks performed in the getResidual method.

| Partition ID | Constructor Time | getResidual Time | Assembly Time | Residual Time | MPIReduceAll Time |
|---|---|---|---|---|---|
| DIS_RESIDUAL_FA_COMP (DOFs = 25461891, Elasticity3D) | | | | | |
| 1 | 65.68 | 52.84 | 13.49 | 35.03 | 4.125 |
| 2 | 63.96 | 64.00 | 13.47 | 35.69 | 12.49 |
| 3 | 66.75 | 64.91 | 10.48 | 34.88 | 21.53 |
| 4 | 64.94 | 66.82 | 11.84 | 36.40 | 18.57 |
| 5 | 67.51 | 68.29 | 9.950 | 33.99 | 24.31 |
| 6 | 66.33 | 65.98 | 11.16 | 35.32 | 19.49 |
| 7 | 64.96 | 65.55 | 10.26 | 35.11 | 20.17 |
| 8 | 65.91 | 66.79 | 12.99 | 37.79 | 15.98 |

Table 8.5: The table displays the timing for each partition FEResidalFAMPI object stages for the 8 partition cube mesh. Each row displays the timing step during construction and residual calculation stage. The last three columns have the timing data for the three tasks performed in the getResidual method.

The FEResidalFAMPI objects calculation is composed of two stages. In first stage the FEResidalFAMPI object is constructor. It involves construction of the internal data structures and acquisition of data from the allocated elements. The second stage involves calculation of the residual vector. In first residual iteration, the assembly of data into efficient data structures is performed. The data in efficient data structures are used to calculate the residual efficiently. The interface DOFs residual vector global values are computed by performing distributed sum. The FEResidalFAMPI::getResidual method is divided into 3 sub stages: assembly of data, residual calculation time and MPIReduceAll time. Table 8.4, Table 8.5, Table 8.6 and Table 8.7 represent the timing for these stages for the mesh with 4, 8, 16 and 24 partitions, respectively. The 3D cube mesh is used for testing. The mesh is partitioned using GMSH package. These partitions have almost the sane number of elements. In these tables it is observed that for each stage, all the processes have consumed similar timing except for the last column. In Table 8.4 last column values varies between 20.18 sec and 57 sec. For Table 8.5 last column values varies between 4.125 sec and 24.13 sec. and in Table 8.6 the MPIResduceAll time column varies between 9.13 sec and 28.86 sec. In the last Table 8.7 the MPI communication time consumption for the getResidual method for all iterations varies between 9.35 sec and 44.04 sec. All the timing data is obtained through the *Intel*® Trace Analyzer. In future work the improvements will be made in the FEDomainMPI DIS_RESIDUAL_FA_COMP solver to make MPI communication timing stable.

| Partition ID | Constructor Time | getResidual Time | Assembly Time | Residual Time | MPIReduceAll Time |
|---|---|---|---|---|---|
| DIS_RESIDUAL_FA_COMP (DOFs = 25461891, Elasticity3D) | | | | | |
| 1 | 34.82 | 37.23 | 5.47 | 19.87 | 11.65 |
| 2 | 34.06 | 36.24 | 5.18 | 20.21 | 10.62 |
| 3 | 34.27 | 49.92 | 5.34 | 18.79 | 25.58 |
| 4 | 33.95 | 49.68 | 5.28 | 19.26 | 24.97 |
| 5 | 33.54 | 57.86 | 5.03 | 18.57 | 28.86 |
| 6 | 34.74 | 51.19 | 5.79 | 18.52 | 26.86 |
| 7 | 34.51 | 34.12 | 5.07 | 19.67 | 09.36 |
| 8 | 35.57 | 34.18 | 5.27 | 19.76 | 09.13 |
| 9 | 35.06 | 48.72 | 5.11 | 18.31 | 25.29 |
| 10 | 34.02 | 48.51 | 4.87 | 18.40 | 25.32 |
| 11 | 34.38 | 42.39 | 6.09 | 19.11 | 17.18 |
| 12 | 34.56 | 42.29 | 6.28 | 19.20 | 16.08 |
| 13 | 34.33 | 49.33 | 5.30 | 18.87 | 25.15 |
| 14 | 34.85 | 48.96 | 5.21 | 18.59 | 25.14 |
| 15 | 34.27 | 42.07 | 5.25 | 19.03 | 17.77 |
| 16 | 34.01 | 42.77 | 5.24 | 19.07 | 18.45 |

Table 8.6: The table displays the timing for each partition FEResidalFAMPI object stages for the 16 partition cube mesh. Each rows display the timing step during construction and residual calculation stage. The last three columns have the timing data for the three tasks performed in the getResidual method.

| Partition ID | Constructor Time | getResidual Time | Assembly Time | Residual Time | MPIReduceAll Time |
|---|---|---|---|---|---|
| DIS_RESIDUAL_FA_COMP (DOFs = 25461891, Elasticity3D) | | | | | |
| 1 | 25.36 | 27.42 | 3.44 | 13.51 | 10.56 |
| 2 | 24.86 | 38.09 | 3.70 | 12.99 | 21.13 |
| 3 | 24.83 | 57.62 | 3.55 | 13.11 | 40.72 |
| 4 | 24.29 | 29.11 | 4.16 | 13.78 | 10.93 |
| 5 | 24.24 | 49.90 | 3.54 | 12.99 | 33.08 |
| 6 | 24.38 | 32.22 | 3.51 | 13.30 | 15.13 |
| 7 | 25.32 | 41.57 | 3.43 | 12.99 | 25.00 |
| 8 | 22.19 | 26.32 | 3.58 | 13.24 | 09.35 |
| 9 | 24.95 | 15.59 | 3.36 | 11.98 | 44.04 |
| 10 | 25.53 | 28.20 | 3.47 | 13.09 | 11.54 |
| 11 | 25.41 | 40.99 | 3.46 | 13.19 | 24.23 |
| 12 | 25.58 | 34.21 | 3.49 | 13.97 | 16.59 |
| 13 | 25.13 | 40.07 | 3.45 | 12.60 | 23.88 |
| 14 | 25.23 | 45.69 | 3.41 | 12.81 | 29.32 |
| 15 | 25.30 | 41.35 | 3.45 | 13.14 | 24.52 |
| 16 | 24.48 | 36.14 | 3.67 | 14.12 | 18.08 |
| 17 | 24.82 | 38.57 | 3.31 | 13.57 | 21.55 |
| 18 | 25.58 | 34.01 | 3.64 | 13.25 | 16.92 |
| 19 | 25.07 | 38.60 | 3.47 | 13.36 | 21.65 |
| 20 | 25.27 | 38.42 | 3.47 | 12.81 | 21.92 |
| 21 | 25.06 | 38.74 | 3.58 | 14.01 | 21.02 |
| 22 | 25.01 | 29.58 | 4.32 | 13.90 | 11.23 |
| 23 | 25.77 | 41.88 | 3.58 | 13.00 | 25.02 |
| 24 | 24.99 | 38.28 | 3.28 | 12.99 | 21.19 |

Table 8.7: The table displays the timing for each partition FEResidalFAMPI object stages for the 24 partition cube mesh. Each rows display the timing step during construction and residual calculation stage. The last three columns have the timing data for the three tasks performed in the getResidual method.

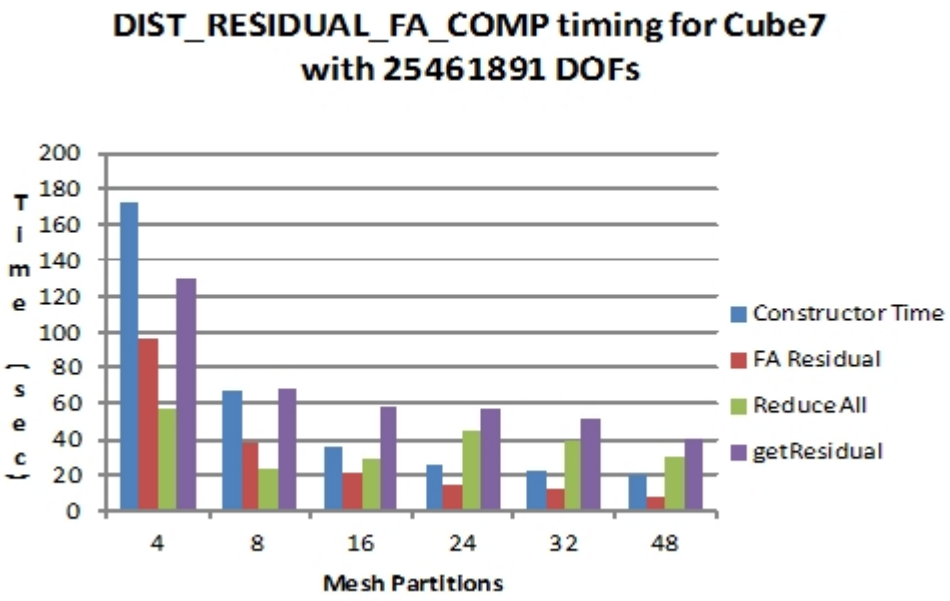



Figure 8.4: The DIS_RESIDUAL_FA_COMP solver time comparison for the mesh with 25,461,891 DOFs solved using different partitions.

Figure 8.4 has the time comparison of the DIS_RESIDUAL_FA_COMP method different stages. In this the Constructor time represents the time consumed by the FEResidualFAMPI class. The FA Residual is the largest time consumed by a partitions to compute $\vec{r}_P$ and the ReduceAll is the largest time spend by a partition for 100 iterations. The getResidual represents the largest time consumed by a partition for FEResidualFAMPI::getResidual 100 iterations. It is sum of the data assemble time, 100 residuals calculation time and the 100 MPI_ReduceALL calls. It can be seen in Figure 8.4 that the Constructor time, and FA Residual has reduced at with the rise in MPI precesses. The time consumed for the MPI_ReduceALL are unpredictable. For the 8 parallel processes the MPI_ReduceALL has consumed lowest time. The MPI_ReduceAll become significant for the meshes with higher partitions like in Figure 8.4 partitions 24, 36 and 48.

## 8.4 Conclusion

In this chapter three algorithms of the FEDomain residual systems for distributed memory architectures are discussed. This chapter explains what were the possible implementations and why a certain implementation is selected. In all these methods it is observed that the MPIReduceAll communication has variable time. In future work, an alternative to MPIReduceAll method will be implemented.

# Chapter 9

# Extension to a non-linear solver for the Convection-Diffusion equation

There are vast range of non-linear problems and it is not possible to develop a totally general non-linear solver. In this chapter the non-linear solver is introduced in the FEDomain package to calculate convection-diffusion equation solution. The non-linear convection-diffusion solver is developed for the shared memory architectures, and is added in the FEDomain library as FENonLinearSolver class. The interface of these files are given below in Listing 9.1.

## 9.1 Interface

```
class FENonLinearSolver{
      FENonLinearSolver(vector<FEElement*>& elements,
            map<FE_UINT, FE_DATA>& dirichlet_constraints,
            FE_UINT total_system_dofs,
            FE_UINT max_elements_dofs,
            MATRIX_TYPE& system_matrix,
            FE_UINT max_iterations);
```

```
        void setSolution(const map<FE_UINT, FE_DATA>& dirichlet_data);
};
```

Listing 9.1: Interface of FENonLinearSolver class

The non linear solver is implemented for the shared memory architecture. It is added into FEDomain package which is selected by setting the solver method in FEDomain constructor equal to NON_LINEAR_SOLVER.

## 9.2 Implementation

The non linear equations solutions are computed using the following fixed point algorithm. In each iteration a system of linear equations is solved. The elements data in the next step is based on the solution of the present, or the previous iteration steps. The FENonLinearSolver object has to assemble the elements' data in every iteration. The memory footprint of the FENonLinearSolver stiffness matrix remains unmodified as the relationship among the DOFs do not change. The internal structures are constructed in the first step and reused in next iterations. Structure of the non linear problem is: Find $u_h \in V_h$ such that:

$$a_h(u_h; v_h) + N_h(u_h; u_h, v_h) = (f, v_h) \qquad \forall v_h \in V_h,$$

We solve it with following fixed point method. For the approximate solution $u_h^n$, $\tilde{u}_h^{n+1}$ is computed as

$$\tilde{u}_h^{n+1}: \quad a_h(\tilde{u}_h^{n+1}; v_h) + N_h(u_h^n; \tilde{u}_h^{n+1}, v_h) = (f, v_h) \qquad \forall v_h \in V_h,$$

where $\boldsymbol{N}_h$ is a non-linear form that depends on the given method. In all cases the dependence of $\boldsymbol{N}$ on the second argument is linear. The next iterate is defined as

$$u_h^{n+1} := u_h^n - \omega_{n+1}(\tilde{u}_h^{n+1} - u_h^n)$$

with the damping factor $\omega_{n+1} > 0$. The damping factors $\omega_{n+1}$ are computed dynamically through the algorithm taken from [49] (given in section 5 on page 2008). As in [49] we have fixed the maximum number of iterations to 100000. We have followed this recommendation at this first stage, mostly due to the fact that the convergence might be slow, and the size of the problems we have to deal with is large. Our intention was to avoid overlooking a case in which there is a slow convergence. If the solution does not converge in 100000 iterations, it stops by displaying a message on the screen. The number of iterations are fixed for this version as current FEDomain class interface does not allow to set number of iterations. Still a user can set by maximum iterations by setting a max_iterations parameter of FENonLinearSolver constructor in the FEDomain class before compilation.

The non linear equations have two sparse matrices ( $\boldsymbol{A}$ to store $a_h$ and $\boldsymbol{N}$ to store $N_h$) in left hand side. These matrices are of same dimensions and contain same amount of data. The dimension of these matrices and their data count remains unchanged in all the iterations. The FEElement interface class provided in FEDomain package does not have the interface method to collect the $\boldsymbol{N}_K$. The interface allows the user to provide his $\boldsymbol{A}_K$. The user has to provide $\boldsymbol{A}_K$ after adding its $\boldsymbol{N}_K$ into it. The $\boldsymbol{A}_K$ is modified in every iteration. The FEEquation class is not implemented for the non linear system of equations. It does not provide a separate interface methods to collect $\boldsymbol{A}_K$ and $\boldsymbol{N}_K$. The FEEquation object creates a single matrix container to collect $\boldsymbol{A}_K + \boldsymbol{N}_K$ as a left hand side. The user has to provide elements' data after applying Dirichlet constraints.

The FENonLinearSolver solver in each iteration collects element data. The computation of element data is dependent on previous iteration solution. For the first iteration, the zero solution vector should be used in element objects to compute stiffness matrix and load vectors. At the end of each iteration the solution

is provided by FEDomain to the element objects which is be used in next iteration element data computations. In FENonLinearSolver setSolution method algorithm is given in Algorithm 6. The calculation initiate by scattering zero solution among the element objects. The FENonLinearSolver class is composed of FEEquation object which is responsible to assemble system of equations and compute direct solution. It also computes residual for the NonLinearSolver. In NonLinearSolver class the solution is refined by using damping factor algorithm taken from [49].

```
   // Convection Diffusion Non Linear Solver setSolution()
 1 void setSolution(){
 2 ScatterSolution(zero);
 3 DataAssembling();
 4 FEEquation.ComputeSolution();
 5 ScatterSolution();
 6 DataAssembling();
 7 FEEquation.ComputeResidual();
 8 while IsNotConverged() do
 9 |   ScatterSolution();
10 |   DataAssembling();
11 |   ComputeResidualUsingDampingFactor();
12 end
13 ScatterSolution();
14 };
```

**Algorithm 6:** Non Linear Solver algorithm

For each iteration the PARDISO solver has to be reconstructed. It is observed that even though only the left hand side data changes but its memory foot print remains unchanged. During design phase it was assumed that symbolic factorization has to the performed in the first iteration. During development it was observed that PARDISO has to perform both symbolic and numerical factorization in each

iteration. This problem will be addressed in future work.

The extension only requires the user to specify, at each step, what linear system needs to be solved. Then, extending this framework to other non linear problem requires the user to specify the matrix that needs to be inverted at each step. This is in user's domain and can be used with other non linear problems.

## 9.3 Future Works

The FEEquation class is implemented for the linear system of equations. It has to be modified for the add the support for the non linear system of equations. The support has to be added in the FEEquation and FEElement classes for $\boldsymbol{N}_K$. The efficient algorithm needs to be implemented to assemble data and compute solution for the system. Finally the non linear solver have to be implemented for the distributed memory architectures.

# Chapter 10

# Convection-Diffusion Equation Examples

## 10.1 Problem

This chapter describes the algorithm used in implementation of the 2D and 3D convection diffusion formulation in C++ element classes. These classes are used with the FEDomain package to compute the solutions for number of 2D and 3D test examples on shared memory architectures. The convection diffusion equation is implemented with the popular streamline upwind/Petrov-Galarkin (SUPG) [24] stabilization and spurious oscillation at layers diminishing (SOLD) methods [49] for smoothing of solutions.

We consider the steady scalar convection-diffusion equation

$$-\varepsilon\Delta u + \vec{c}\cdot\nabla u = f \text{ on } \Omega,\ u = u_b \text{ on } \partial\Omega. \tag{10.1}$$

We assume that $\Omega$ is a bounded domain in $\mathbb{R}^d$ (d=1,2,3,..,N) with a polygonal boundary $\partial\Omega$, $\varepsilon > 0$ is the constant diffusivity, $\vec{c}$ is a smooth, solenoidal convection field, $f \in L^2(\Omega)$ is an outer source, and $u_b$ represents the Dirichlet boundary

condition.

The streamline upwind/Petrov-Galerkin (SUPG) method [24] is frequently used because of its stability and higher order accuracy. In the convection dominated problems the SUPG solutions typically contain oscillations in layer regions. The spurious oscillations at layers diminishing (SOLD) [49] methods are used to suppress the local oscillations present in SUPG discrete solutions.

To define the discretization the triangulation $\mathcal{P}$ of the domain $\Omega$ is introduced which consists of a finite number of triangle elements $K$. All the elements of $\mathcal{P}$ are triangles. The finite element space used is defined as

$$W_h = \{v_K \in C^0(\overline{\Omega}) : v_h|_K \in \mathbb{P}_1(K) \qquad \forall K \in \mathcal{P}\}, \tag{10.2}$$

$$V_h = \{v_K \in W_h : v_h|_{\partial\Omega} = 0\}. \tag{10.3}$$

Let $u_{bh} \in W_h$ be a function whose trace approximates the boundary condition $u_b$. Then the Galerkin finite element discretization of the convection-diffusion equation (10.1) reads: Find $u_h \in W_h$ such that $u_h - u_{bh} \in V_h$ and

$$a(u_h, v_h) = (f, v_h) \qquad \forall v_h \in V_h,$$

where

$$a(u, v) = \varepsilon \int_\Omega \nabla u \cdot \nabla v + \int_\Omega \vec{c} \cdot \nabla u \; v.$$

Where $\vec{c}$ is the convection field. It is well known that this discretization is inappropriate if convection dominates diffusion since then the discrete solution is usually globally polluted by spurious oscillations.

## 10.2 Streamline Upwind/Petrov-Galerkin

An improvement can be achieved by adding a stabilization term to the Galerkin discretization. One of the most efficient procedures of this type is the streamline upwind/Petrov-Galerkin (SUPG) method developed by Brooks and Hughes [24]. To formulate this method the residual we define

$$R_h(u) = -\varepsilon \Delta_h u + \vec{c} \cdot \nabla u - f, \tag{10.4}$$

where $\Delta_h u$ is the Laplace operator defined element wise, i.e., $(\Delta_h v)|_K = \Delta(v|_K)$ for every $K \in \mathcal{P}$ and piecewise smooth function $v$. The SUPG method reads:

Find $u_h \in W_h$ such that $u_h - u_{bh} \in V_h$ and

$$a(u_h, v_h) + (R_h(u_h), \tau \vec{c} \cdot \nabla v_h) = (f, v_h) \qquad \forall v_h \in V_h, \tag{10.5}$$

where $\tau \in L^\infty(\Omega)$ is a non-negative stabilization parameter. The choice of $\tau$ influences the accuracy of the discrete solution but a general optimal definition of $\tau$ is still not known. Here $h_K$ represents the length of the longest side of the triangle and $h := max\{h_K : K \in \mathcal{P}\}$. During our computations for an element $K \in \mathcal{P}$, $\tau_K$ is defined by the formula

$$\tau_K = \begin{cases} \frac{h_K}{2|c|} & \text{if } P_{e_K} \geq 1 \\ \frac{(h_K)^2}{4\varepsilon} & \text{if } P_{e_K} < 1 \end{cases} \qquad \text{with} \quad P_{e_K} = \frac{h_K |c|}{2\varepsilon}. \tag{10.6}$$

The Péclet number $P_{e_K}$ [33] is an element local parameter which is very large if convection strongly dominates diffusion in $\Omega$. The $|c|$ is the euclidean norm of vector $\vec{c}$, it is calculated as:

$$|c| = \sqrt{c_1^2 + c_2^2 + ..... + c_N^2}.$$

### 10.2.1 Error Computation

The $L^2$-norm and $H^1$-norm errors of the solutions are represented as $e_L$ and $e_H$, respectively. The $L^2$-norm and $H^1$-norm errors for an element $K$ are represented as $e_{L_K}$ and $e_{H_K}$ respectively. The error of the computation is calculated by computing every element error $e_K$. The total $L^2$-norm and $H^1$-norm errors are computed as follows

$$e_L = \sqrt{\sum_{K \in \Omega} e_{L_K}^2} \qquad \text{and} \qquad e_H = \sqrt{\sum_{K \in \Omega} e_{H_K}^2}.$$

In a 2D dimensional domain ($\Omega_{2D}$) the triangulation generates triangle elements. For each triangle element error is computed at the three midpoints ($m_0$, $m_1$ and $m_2$) of the edges. The discrete solutions on the midpoints $u_m$ are computed using following equations.

$$\begin{aligned} u_{m_0} &= \frac{u_{h_1} + u_{h_2}}{2}, \\ u_{m_1} &= \frac{u_{h_0} + u_{h_2}}{2}, \\ u_{m_2} &= \frac{u_{h_0} + u_{h_1}}{2}. \end{aligned}$$

The $L^2$ - norm error on a triangle element is computed as follows

$$e_{L_K}^2 = \int_K (u - u_h)^2 \simeq \frac{|K|}{3} \sum_{i=1}^{3} (u - u_h)^2(m_i). \tag{10.7}$$

The $H^1$ - norm error on a triangle element is computed as follows

$$\begin{aligned} e_{H_K}^2 &= \int_K (\partial_x (u - u_h)^2 + \partial_y (u - u_h)^2) \\ &\simeq \frac{|K|}{3} \sum_{i=1}^{3} (\partial_x (u - u_h)^2 + \partial_y (u - u_h)^2)(m_i). \end{aligned} \tag{10.8}$$

In a 3D domain ($\Omega_{3D}$) the triangulation generates tetrahedron elements. For a tetrahedron element class the load vector is calculated using the five points quadrature rules taken from [47]. The rule provides the five evaluation points $\mathcal{X}_i$ and their weights $\mathcal{W}_i$ which are given below.

$$\begin{aligned}
\mathcal{X}_1 &= \frac{n_1}{4}+\frac{n_2}{4}+\frac{n_3}{4}+\frac{n_4}{4} \qquad \mathcal{W}_1 = \frac{-4}{5},\\
\mathcal{X}_2 &= \frac{n_1}{6}+\frac{n_2}{6}+\frac{n_3}{6}+\frac{n_4}{2} \qquad \mathcal{W}_2 = \frac{9}{20},\\
\mathcal{X}_3 &= \frac{n_1}{6}+\frac{n_2}{6}+\frac{n_3}{2}+\frac{n_4}{6} \qquad \mathcal{W}_3 = \frac{9}{20},\\
\mathcal{X}_4 &= \frac{n_1}{6}+\frac{n_2}{2}+\frac{n_3}{6}+\frac{n_4}{6} \qquad \mathcal{W}_4 = \frac{9}{20},\\
\mathcal{X}_5 &= \frac{n_1}{2}+\frac{n_2}{6}+\frac{n_3}{6}+\frac{n_4}{6} \qquad \mathcal{W}_5 = \frac{9}{20}.
\end{aligned}$$

The $L^2-norm$ and $H^1-norm$ errors are calculated for the tetrahedron elements at these evaluation points. The $L^2-norm$ is calculated as follows

$$\begin{aligned}
e^2_{L_K} &= \textstyle\int_K (u-u_h)^2 \simeq |K| \sum_{i=1}^4 (u-u_h)^2(\mathcal{X}_i)\mathcal{W}_i \\
&= |K| \textstyle\sum_{i=1}^4 \mathcal{W}_i (u(\mathcal{X}_i) - \sum_{j=1}^4 u_h(n_j)\lambda_j(\mathcal{X}_i))^2. \qquad (10.9)
\end{aligned}$$

The $H^1-norm$ is calculated as follows

$$e^2_{H_K} = \int_K (\partial_x(u-u_h)^2 + \partial_y(u-u_h)^2 + \partial_z(u-u_h)^2). \qquad (10.10)$$

and is approximated using the same quadrature formula.

### 10.2.2 Error Rate

The rate of reduction of $L^2$-norm errors for $\Omega_{2D}$ and $\Omega_{3D}$ are computed using following equation:

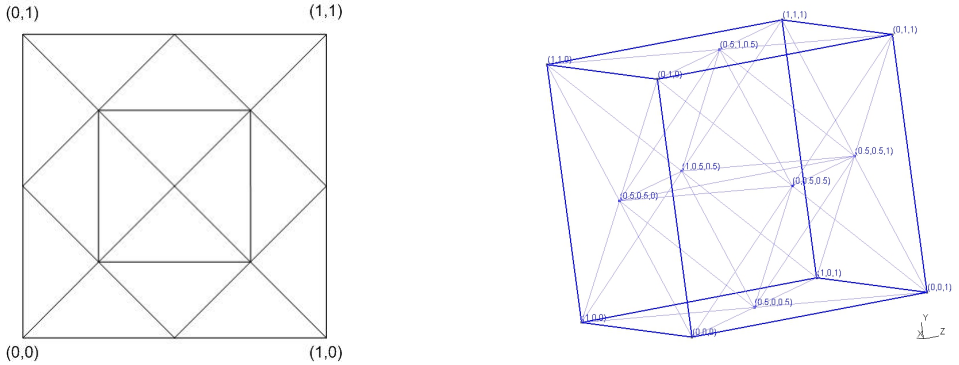
$$rate = \frac{\ln(e_{L(i)}/e_{L(i+1)})}{\ln(h_i/h_{i+1})}, \qquad (10.11)$$

and for $H^1$-norm errors rate of error reduction is calculated in the same way.

As $\| u - u_h \|_{L_2} \leqslant Ch^2$ and $\| u - u_h \|_{H_1} \leqslant Ch$, then we expect the error to reduce by the factor of 2 for $L^2$-norm error and factor of 1 for the $H^1$-norm error.

### 10.2.3 Domains

In 2D computations the domain is $\Omega_{2D} = (0,1)^2$, while in 3D we use the domain $\Omega_{3D} = (0,1)^3$. Figure 10.1 depicts the basic domains and meshes used in the computations.




(a) 2D Mesh (b) 3D Mesh

Figure 10.1: Domains and initial meshes.

| Mesh | Elements | DOFs | $h_i$ | | Mesh | Elements | DOFs | $h_i$ |
|---|---|---|---|---|---|---|---|---|
| sq0 | 24 | 13 | 0.5 | | cb0 | 68 | 14 | 1 |
| sq1 | 80 | 41 | 0.25 | | cb1 | 820 | 63 | 6.12E-01 |
| sq2 | 288 | 145 | 0.125 | | cb2 | 1976 | 365 | 3.95E-01 |
| sq3 | 1088 | 545 | 0.0625 | | cb3 | 13928 | 3457 | 2.65E-01 |
| sq4 | 4224 | 2113 | 0.03125 | | cb4 | 104648 | 17969 | 1.71E-01 |
| sq5 | 16640 | 8321 | 0.015625 | | cb5 | 811400 | 137313 | 1.15E-01 |
| sq6 | 66048 | 33025 | 0.0078125 | | cb6 | 6390536 | 1073345 | 7.57E-02 |

Table 10.1: $\Omega_{2D}$ Square Meshes and $\Omega_{3D}$ Cube Meshes details. These meshes are used during the computations.

### 10.2.4 SUPG Error Results

#### 10.2.4.1 2D Error Test: a constant convectional field

The convection_diffusion equation is tested using the FEDomain linear solver. The FEDomain solver requires the user to provides triangle elements that provide element stiffness matrix $\boldsymbol{A}_K$ and load vector $\vec{b}_K$ for (10.5). The SUPG error testing for $\Omega_{2D}$ in Figure 10.1 is preformed using meshes given in Table 10.1. For test problems all the boundary conditions are set of homogeneous Dirichlet. The exact solution of the first test example used is

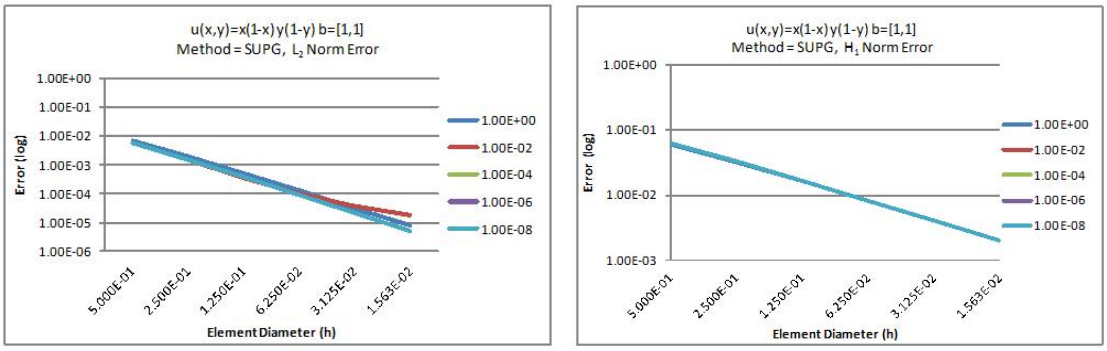
$$u(x, y) = x(1 - x)y(1 - y), \tag{10.12}$$

and the convection vector is defined as

$$\vec{c} = (1, 1)^T, \tag{10.13}$$

and applied force on the domain is

$$f = 2\varepsilon((x - x^2) + (y - y^2)) + (1 - 2x)(y - y^2) + (x - x^2)(1 - 2y). \tag{10.14}$$




(a) $L^2$-Norm Error (b) $H^1$-Norm Error

Figure 10.2: The figure have the $L^2$-norm and $H^1$-norm errors graphs. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

| mesh | $\varepsilon$ = 1E-0 | | $\varepsilon$ = 1E-2 | | $\varepsilon$ = 1E-4 | | $\varepsilon$ = 1E-6 | | $\varepsilon$ = 1E-8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| name | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ |
| SUPG Error | | | | | | | | | | |
| sq0 | 6.94E-3 | 5.92E-2 | 5.75E-3 | 6.20E-2 | 5.76E-3 | 6.27E-2 | 5.76E-3 | 6.27E-2 | 5.76E-3 | 6.27E-2 |
| sq1 | 1.96E-3 | 3.17E-2 | 1.47E-3 | 3.27E-2 | 1.52E-3 | 3.33E-2 | 1.52E-3 | 3.33E-2 | 1.52E-3 | 3.33E-2 |
| sq2 | 5.11E-4 | 1.62E-2 | 3.67E-4 | 1.63E-2 | 3.82E-4 | 1.67E-2 | 3.82E-4 | 1.67E-2 | 3.82E-4 | 1.67E-2 |
| sq3 | 1.29E-4 | 8.16E-3 | 1.03E-4 | 8.17E-3 | 8.97E-5 | 8.25E-3 | 8.99E-5 | 8.25E-3 | 8.99E-5 | 8.25E-3 |
| sq4 | 3.25E-5 | 4.09E-3 | 3.76E-5 | 4.09E-3 | 2.10E-5 | 4.10E-3 | 2.10E-5 | 4.10E-3 | 2.10E-5 | 4.10E-3 |
| sq5 | 8.13E-6 | 2.04E-3 | 1.75E-5 | 2.05E-3 | 5.07E-6 | 2.05E-3 | 5.05E-6 | 2.05E-3 | 5.05E-6 | 2.05E-3 |
| SUPG Error Reduction Rate | | | | | | | | | | |
| | 1.82 | 0.90 | 1.97 | 0.92 | 1.92 | 0.91 | 1.92 | 0.91 | 1.92 | 0.91 |
| | 1.94 | 0.97 | 2.00 | 1.00 | 1.99 | 1.00 | 1.99 | 1.00 | 1.99 | 1.00 |
| | 1.99 | 0.99 | 1.83 | 1.00 | 2.09 | 1.02 | 2.09 | 1.02 | 2.09 | 1.02 |
| | 1.99 | 1.00 | 1.45 | 1.00 | 2.09 | 1.01 | 2.10 | 1.01 | 2.10 | 1.01 |
| | 2.00 | 1.00 | 1.10 | 1.00 | 2.05 | 1.00 | 2.06 | 1.00 | 2.06 | 1.00 |

Table 10.2: The error data for the SUPG method for the first test example using $\Omega_{2D}$. The table has the $L_2$-norm and $H_1$-norm errors data for the meshes given in Table 10.1. The $\varepsilon$ are used for range of values 1E-0, 1E-2, 1E-4, 1E-6 and 1E-8.

Table 10.2 has the SUPG method error data and the error reduction rate for $\Omega_{2D}$ square meshes. Figure 10.2 shows the error data graph where the y-axis has the error data in log scale and x-axis represents $h$. The $L^2$-norm and $H^1$-norm errors are reducing at constant rate with the reduction in $h$ as can be observed in the lower section of Table 10.2. For the $\varepsilon = 1E-02$ the error reduction rate for $L^2$-norm error has not reduced at the constant rate as can be seen in the Table 10.2.

#### 10.2.4.2 SUPG 2D Error Test : A variable convection field

The second test example used for the $\Omega_{2D}$ for SUPG problem is given below. Again the mesh has homogeneous Dirichlet conditions. The exact solution of the second example is:

$$u(x, y) = 100x^2(1-x)^2y(1-y)(1-2y). \tag{10.15}$$

the convection vector for $\Omega_{2D}$ second example is

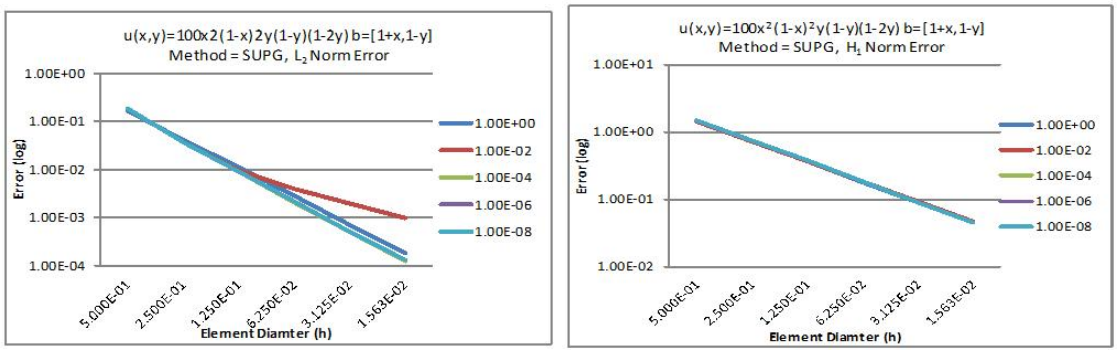
$$\vec{c} = (x+1, 1-y)^T. \tag{10.16}$$

| Mesh Name | $\varepsilon$ = 1E-0 | | $\varepsilon$ = 1E-2 | | $\varepsilon$ = 1E-4 | | $\varepsilon$ = 1E-6 | | $\varepsilon$ = 1E-8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ |
| SUPG Error Data | | | | | | | | | | |
| sq | 1.64E-1 | 1.42 | 1.83E-1 | 1.40 | 1.86E-1 | 1.45 | 1.87E-1 | 1.45 | 1.87E-1 | 1.45 |
| sq1 | 4.10E-2 | 7.15E-1 | 3.65E-2 | 7.09E-1 | 3.70E-2 | 7.39E-1 | 3.70E-2 | 7.40E-1 | 3.70E-2 | 7.40E-1 |
| sq2 | 1.10E-2 | 3.58E-1 | 9.52E-3 | 3.57E-1 | 8.91E-3 | 3.70E-1 | 8.95E-3 | 3.70E-1 | 8.95E-3 | 3.70E-1 |
| sq3 | 2.82E-3 | 1.79E-1 | 3.84E-3 | 1.80E-1 | 2.09E-3 | 1.81E-1 | 2.11E-3 | 1.82E-1 | 2.11E-3 | 1.82E-1 |
| sq4 | 7.08E-4 | 8.96E-2 | 1.99E-3 | 9.11E-2 | 5.01E-4 | 8.99E-2 | 5.12E-4 | 9.01E-2 | 5.12E-4 | 9.01E-2 |
| sq5 | 1.77E-4 | 4.48E-2 | 9.83E-4 | 4.59E-2 | 1.21E-4 | 4.48E-2 | 1.27E-4 | 4.49E-2 | 1.27E-4 | 4.49E-2 |
| SUPG Error Rate | | | | | | | | | | |
| | 2.00 | 0.99 | 2.33 | 0.98 | 2.33 | 0.97 | 2.34 | 0.97 | 2.34 | 0.97 |
| | 1.90 | 1.00 | 1.94 | 0.99 | 2.05 | 1.00 | 2.05 | 1.00 | 2.05 | 1.00 |
| | 1.96 | 1.00 | 1.31 | 0.99 | 2.09 | 1.03 | 2.08 | 1.02 | 2.08 | 1.02 |
| | 1.99 | 1.00 | 0.95 | 0.98 | 2.06 | 1.01 | 2.04 | 1.01 | 2.04 | 1.01 |
| | 2.00 | 1.00 | 1.02 | 0.99 | 2.05 | 1.00 | 2.01 | 1.00 | 2.01 | 1.00 |

Table 10.3: The error data for the SUPG method for the second test example using $\Omega_{2D}$. The table has the $L_2$ norm and $H_1$ norm errors data for the meshes given in Table 10.1. The $\varepsilon$ are used for range of values 1E-0, 1E-2, 1E-4, 1E-6 and 1E-8.

The applied force in $\Omega_{2D}$ second example is:

$$\begin{aligned} f =& 100((12x^2 - 12x + 2)(2y^3 - 3y^2 + y) + (x^4 - 2x^3 + x^2)(12y - 6)) \\ &+100((x+1)(4x^3 - 6x^2 + 2x)(2y^3 - 3y^2 + y)) \\ &+100((1-y)(x^4 - 2x^3 + x^2)(6y^2 - 6y + 1)). \end{aligned} \tag{10.17}$$




(a) $L^2$-Norm Error (b) $H^1$-Norm Error

Figure 10.3: The figure have the $L^2$-norm and $H^1$-norm errors graphs. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

Table 10.3 has the SUPG method error and the error rate data for the $\Omega_{2D}$ error test example with variable convection field. The meshes used during test problem 2 execution are given in Table 10.1. Figure 10.3 has the error graphs for the $L^2$-norm and $H^1$-norm errors. This example displayed the error reduction rate characteristics similar to the example with fixed convection field in Section 10.2.4.1. The execution with the $\varepsilon = 1E-02$ the $L^2$-norm error has not reduced with the required rate of 2.

#### 10.2.4.3 SUPG 3D Error Test Example 1 : variable convective field

The second test problem used to compute the error of the SUPG methods on $\Omega_{3D}$ is defined below. The $\Omega_{3D}$ considered for this example has homogeneous Dirichlet boundary. The actual solution of this test example is given as:

$$u(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z), \tag{10.18}$$

and the convection vector is defined as:

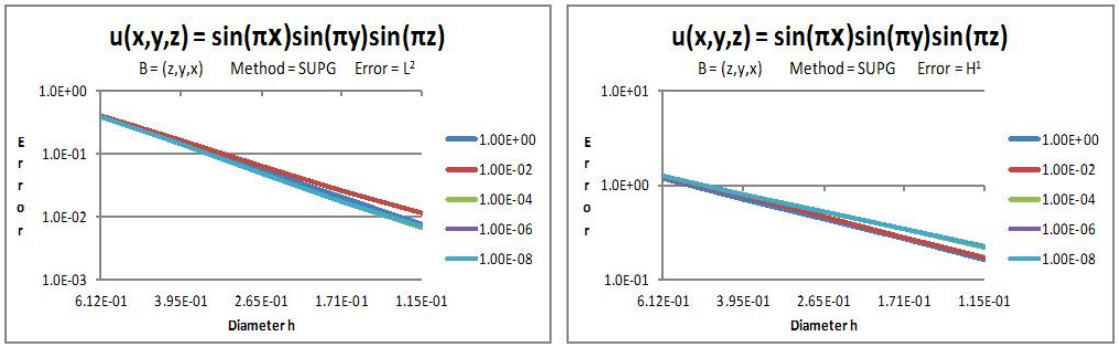
$$\vec{c} = (z,y,x)^T, \tag{10.19}$$

and applied force on the domain is:

$$\begin{aligned} f =& 3\pi^2\epsilon(\sin(\pi x)\sin(\pi y)\sin(\pi z)) \\ &+ \pi z(\cos(\pi x)\sin(\pi y)\sin(\pi z)) \\ &+ \pi y(\sin(\pi x)\cos(\pi y)\sin(\pi z)) \\ &+ \pi x(\sin(\pi x)\sin(\pi y)\cos(\pi z)). \end{aligned} \tag{10.20}$$

| Mesh | $\varepsilon$ = 1E-0 | | $\varepsilon$ = 1E-2 | | $\varepsilon$ = 1E-4 | | $\varepsilon$ = 1E-6 | | $\varepsilon$ = 1E-8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ | $L^2$ | $H^1$ |
| cb1 | 4.04E-1 | 1.19 | 4.01E-1 | 1.22 | 3.91E-1 | 1.24 | 3.91E-1 | 1.24 | 3.91E-1 | 1.24 |
| cb2 | 1.51E-1 | 7.18E-1 | 1.63E-1 | 7.63E-1 | 1.44E-1 | 8.07E-1 | 1.44E-1 | 8.08E-1 | 1.44E-1 | 8.08E-1 |
| cb3 | 5.48E-2 | 4.35E-1 | 6.35E-2 | 4.65E-1 | 4.83E-2 | 5.26E-1 | 4.82E-2 | 5.29E-1 | 4.82E-2 | 5.29E-1 |
| cb4 | 2.03E-2 | 2.66E-1 | 2.62E-2 | 2.81E-1 | 1.75E-2 | 3.40E-1 | 1.75E-2 | 3.44E-1 | 1.75E-2 | 3.44E-1 |
| cb5 | 7.66E-3 | 1.63E-1 | 1.14E-2 | 1.70E-1 | 6.73E-3 | 2.16E-1 | 6.85E-3 | 2.22E-1 | 6.86E-3 | 2.22E-1 |
| Rate of reduction for mesh diameter and error. | | | | | | | | | | |
| | 2.25 | 1.15 | 2.06 | 1.08 | 2.28 | 0.98 | 2.28 | 0.98 | 2.28 | 0.98 |
| | 2.53 | 1.25 | 2.36 | 1.24 | 2.74 | 1.07 | 2.74 | 1.06 | 2.74 | 1.06 |
| | 2.27 | 1.13 | 2.03 | 1.15 | 2.33 | 1.00 | 2.31 | 0.98 | 2.31 | 0.98 |
| | 2.44 | 1.23 | 2.08 | 1.27 | 2.39 | 1.14 | 2.35 | 1.10 | 2.35 | 1.10 |

Table 10.4: The table contains the error rate for the $L^2$-norm and $H^1$-norm errors computed for SUPG method using 3D test example 2 for a cube domain.




(a) $L^2$-Norm Error (b) $H^1$-Norm Error

Figure 10.4: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{3D}$. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

Table 10.4 contains the SUPG errors data and rate for $\Omega_{3D}$ meshes given Table 10.1. The upper half of Table 10.4 has the $L^2$-norm and $H^1$-norm error data for the $\Omega_{3D}$ cube meshes. The lower half of Table 10.4 has the error reduction rate for the $L^2$-norm and $H^1$-norm error for the $\varepsilon$ range. The $L^2$ and $H^2$ norm error rates are computed using the formula in (10.11). Figure 10.4 has graphs for the $L^2$-norm and $H^1$-norm errors. The error show very low sensitivity to $\varepsilon$.

### 10.2.5 Problem with internal and boundary layers

In this section three $\Omega_{2D}$ and two $\Omega_{3D}$ examples are discussed. These examples are executed for all four SOLD methods. $\Omega_{2D}$ used for the problems is defined as:

$$\begin{aligned} \Omega_{2D} &= (0,1)^2, \\ \Gamma_D &= (0 \times [0,1]) \cup ([0,1] \times 0), \\ \Gamma_N &= (1 \times [0,1]) \cup ([0,1] \times 1). \end{aligned}$$

#### 10.2.5.1 SUPG 2D Layer problem constant convection field

The first example selected for the layer visualization is defined below:

$$\begin{aligned} f &= 0, \qquad \vec{c} = (1,1)^T, \\ \frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_N, \\ u &= \begin{cases} 0 & \text{for x} = 0 \text{ and y} = 0, \\ 1 & \text{for x} = 0, \\ 0 & \text{for y} = 0 \end{cases} \qquad \text{on } \Gamma_D. \end{aligned}$$

Figure 10.5 depicts the solution given by the SUPG method. The solution contains the oscillations along the upper and lower curves of the bend as can be observed in Figure 10.5. The oscillations are not desired in the solution, it should be smooth at upper and lower boundary of the bend. In the next section smoothing methods will be used in conjunction with SUPG to remove or reduce these oscillations. The solution is computed for the $\varepsilon = 1E-4$.

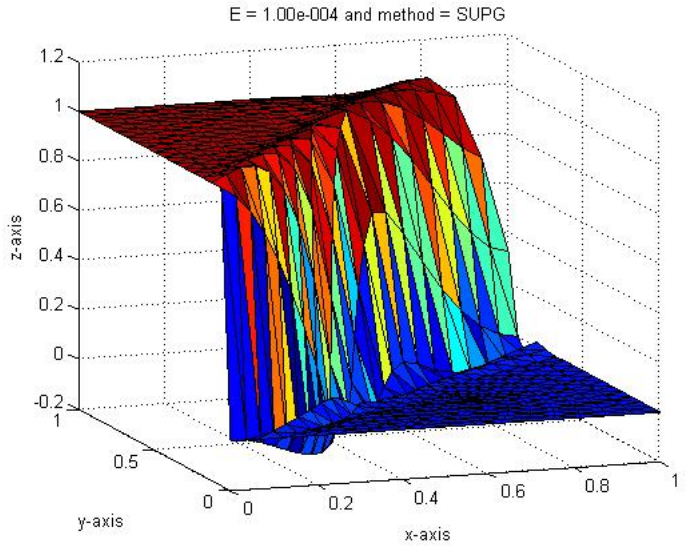
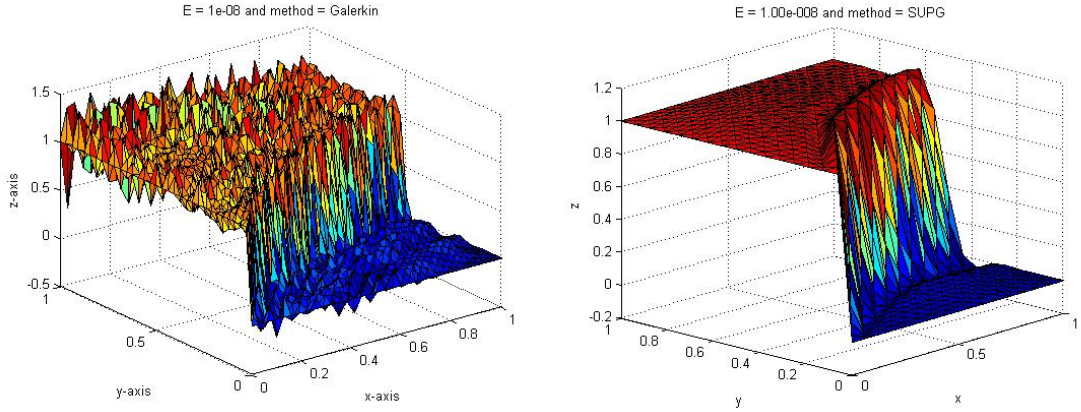



Figure 10.5: Discrete solution obtained using the SUPG method.

#### 10.2.5.2 SUPG 2D Layer problem rotating convection field

The second example used to visualize the oscillations in the SUPG solution is given below:

$$
\begin{aligned}
\vec{c} &= (1+x, 1-y)^T, \qquad f = 0, \\
\frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_N, \\
u &= \begin{cases} 0 & \text{for x} = 0 \text{ and y} = 0, \\ 1 & \text{for x} = 0, \\ 0 & \text{for y} = 0 \end{cases} \qquad \text{on } \Gamma_D.
\end{aligned}
$$




(a) Galerkin solution (b) SUPG solution

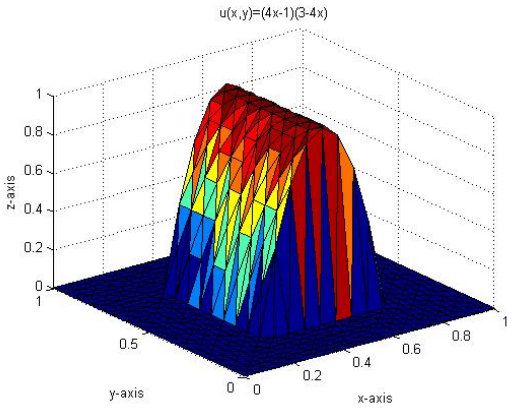
Figure 10.6: figure
Discrete solution obtained using the Galerkin and the SUPG methods. These are computed using sq3 mesh using the $\varepsilon = 1E-8$.

Figure 10.6 shows the Galerkin and SUPG solutions for the sq3 mesh using the $\varepsilon = 1E-8$. The solution contains the oscillations above and under the interior layers. This example will be again used in later sections with the SOLD methods to remove unwanted oscillations.

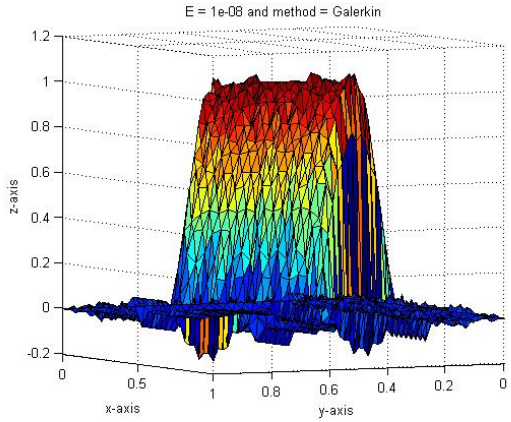
#### 10.2.5.3 SUPG 2D Layer problem 3

The third example, the mesh $\Omega_{2D}$ has homogeneous boundary condition. The solution of this problem is visualization is given below:

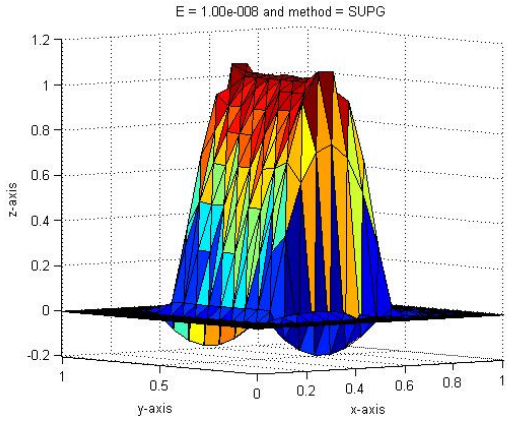
$$\begin{aligned} \vec{c} &= (1,0)^T, \\ f(x,y) &= \begin{cases} 16(1-2x) & \text{for (x,y)} \in [0.25,0.75]^2, \\ 0 & \text{else.} \end{cases} \end{aligned} \tag{10.21}$$




(a) Exact Solution




(b) Galerkin Solution




(c) SUPG solution

Figure 10.7: These graphs represents the exact solution, Galerkin solution and SUPG solutions of the sq3 mesh using $\varepsilon = 1E-8$.

The exact solution and the SUPG solution for the problem are depicted in Figure 10.7 for the sq3 mesh and $\varepsilon = 1E-08$. The actual solution does not have any oscillations, on other hand the SUPG presents have oscillations along the interior layers $[0.25, 0.75]^2$ as well as the top of the curve as can be seen in Figure 10.7c. To remove these oscillations the smoothing SOLD methods will be applied in later sections.

**SUPG 3D Layer problem 1**

The first example considered to visualise the computed solution on $\Omega_{3D}$ is given below. The boundary conditions applied on $\Omega_{3D}$ for this problem is given below.

$$\vec{c} \;=\; (0,1,0)^T, \qquad f = 0,$$

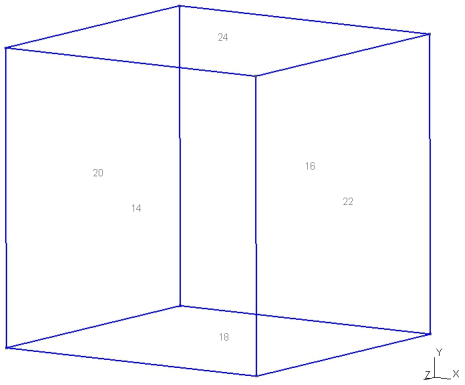
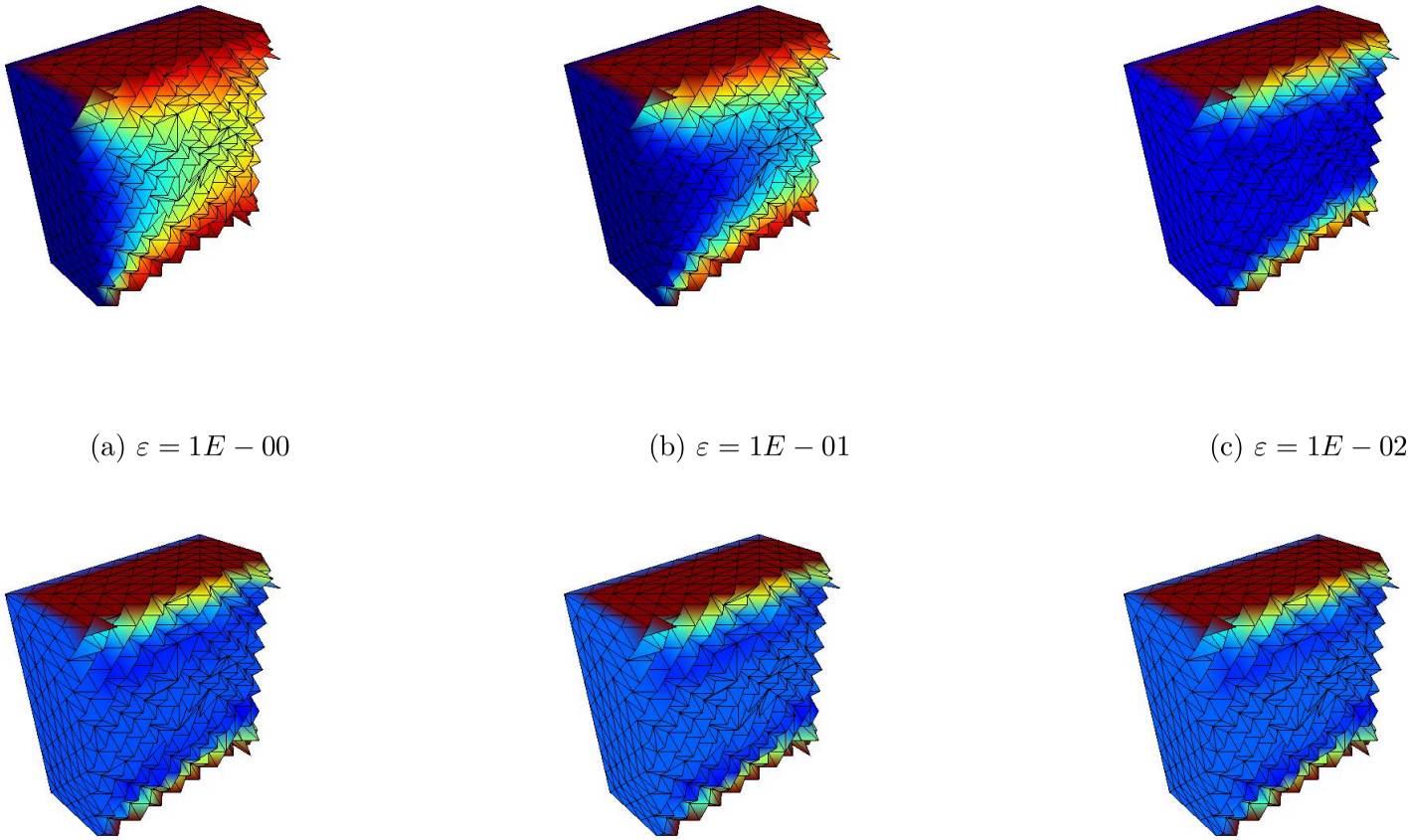



Figure 10.8: Cube with the boundary numbering just included to specify boundary conditions.

The boundary 22 and 24 in Figure 10.8 are Neumann ($\Gamma_N$) having zero value ($\frac{\partial u}{\partial n} = 0$). The boundaries 14, 16, 18 and 20 are Dirichlet ($\Gamma_D$) where boundary 14 and 18 are Neumann with value = 1 while 16 and 20 boundaries are Neumann with zero value. Figure 10.9 have SUPG method solutions obtained using $\varepsilon$ from 1 till $1E-05$ respectively. For the $\varepsilon = 1$ the solution is scattered in whole cube as can be seen in cross section Figure 10.9(a). In Figure 10.9(a) the solution has started moving toward the edges which continues as the $\varepsilon$ value increases. The difference cannot be observed when the $\varepsilon \geq 1E-03$.



(a) $\varepsilon = 1E-00$ (b) $\varepsilon = 1E-01$ (c) $\varepsilon = 1E-02$

(d) $\varepsilon = 1E-03$ (e) $\varepsilon = 1E-04$ (f) $\varepsilon = 1E-05$

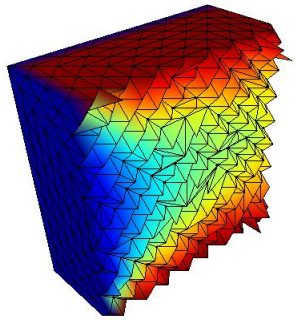
Figure 10.9: Discrete solution obtained using the SUPG method using mesh cb3.

#### 10.2.5.4 SUPG 3D Layer problem 2

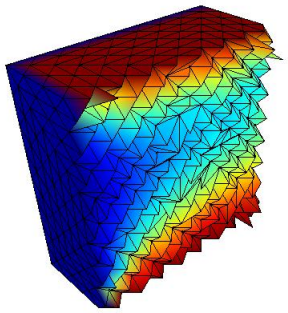
The second example problem used to visualize the SUPG solution for $\Omega_{3D}$ is given below.

$$\vec{c} = (z, y, x)^T, \qquad f = 0.$$

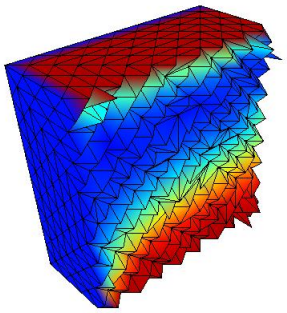
The boundary 22 and 24 in Figure 10.8 are Neumann ($\Gamma_N$) having zero value ($\frac{\partial u}{\partial n} = 0$). The boundaries 14, 16, 18 and 20 are Dirichlet ($\Gamma_D$) where boundary 14 and 18 has -1 value while 16 and 20 boundaries have value equal to 1. Figure 10.10 have SUPG method solutions using $\varepsilon$ from 1 till $1E-05$ respectively. For the $\varepsilon = 1$ the solution is scattered in whole cube as can be seen in cross section Figure 10.10a. In Figure 10.10b the solution has started moving toward the edges which exceed as the $\varepsilon$ value increases. The difference cannot be observed when the $\varepsilon > 1E-03$.



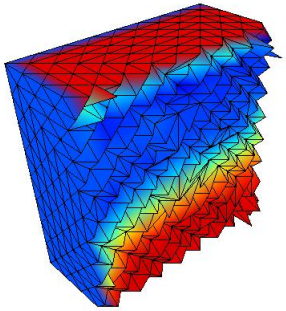
(a) $\varepsilon = 1E-00$



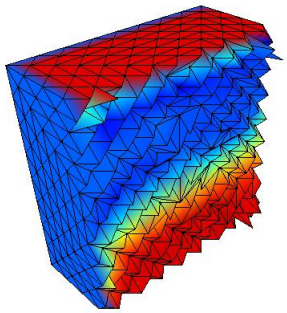
(b) $\varepsilon = 1E-01$



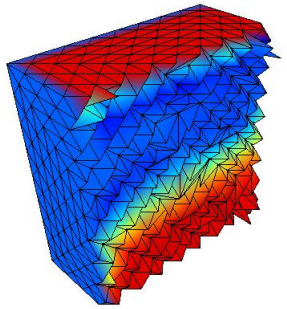
(c) $\varepsilon = 1E-02$



(d) $\varepsilon = 1E-03$



(e) $\varepsilon = 1E-04$



(f) $\varepsilon = 1E-05$

Figure 10.10: Discrete solutions obtained using the SUPG method for the SUPG 3D Layer problem 2. These solutions are obtained using mesh cb3 with $\varepsilon$ from 1E-00 till 1E-05.

## Spurious Oscillation at Layer Diminishing

The discrete solutions provided by the SUPG method still contains spurious oscillations. These oscillations are localized in narrow regions along sharp layers. These are not avoidable and not permissible in many applications. The *Spurious Oscillation at Layer Diminishing* (SOLD) methods [48, 49] are used to add suitable artificial diffusion to the SUPG method. The SOLD methods are nonlinear algorithms for computing discrete solutions. There are two classes of SOLD methods, methods that add isotropic artificial diffusion, and methods adding crosswind artificial diffusion. In this work we have only considered methods that add crosswind diffusion. A typical SOLD method has the form

$$(\widetilde{\varepsilon}\mathbf{c}^{\perp}.\nabla u_h, \mathbf{c}^{\perp}.\nabla v_h)$$

$$a(u_h, v_h) + (R_h(u_h), \tau\vec{c}.\nabla v_h) + (\widetilde{\varepsilon}\mathbf{c}^{\perp}.\nabla u_h, \mathbf{c}^{\perp}.\nabla v_h) = (f, v_h) \qquad \forall v_h \in V_h, \quad (10.22)$$

where for $\Omega_{2D}$ the $\mathbf{c}^{\perp}$ is

$$\mathbf{c}^{\perp} = \frac{1}{|\mathbf{c}|}\begin{bmatrix} -c_2 \\ c_1 \end{bmatrix}, \tag{10.23}$$

and for $\Omega_{3D}$ the $\mathbf{c}^{\perp}$ is

$$\mathbf{c}^{\perp} = \begin{cases} I - \frac{\mathbf{c}\otimes\mathbf{c}}{|\mathbf{c}|^2} & \text{if } \mathbf{c} \neq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{10.24}$$

The parameter $\widetilde{\varepsilon}$ is non negative and usually depends on $u_h$. Every SOLD method has its own definition for $\widetilde{\varepsilon}$. The four crosswind artificial diffusion methods and their results are given below.

### 10.2.6 Codina Method C93

For any $K \in \mathcal{P}$ the C93 [28, 49] definition of $\widetilde{\varepsilon}$ is

$$\widetilde{\varepsilon}|_K = max\left\{0, C\frac{h_K|R_h(u_h)|}{2|\nabla u_h|} - \varepsilon\frac{|R_h(u_h)|}{\vec{c}\cdot\nabla u_h}\right\} \tag{10.25}$$

where $C$ is a suitable constant, where recommended value is 0.7 (cf. [49]).

### 10.2.7 Modified Codina Method KLR02_3

In [49] [49] the following modification of (10.25) was purposed

$$\widetilde{\varepsilon}|_K = max\left\{0, C\frac{h_K|R_h(u_h)|}{2|\nabla u_h|} - \varepsilon\right\}. \tag{10.26}$$

### 10.2.8 Burman and Ern Method BE02_1

For any $K \in \mathcal{P}$ the BE02_1 [49] definition of $\widetilde{\varepsilon}$ is

$$\widetilde{\varepsilon}|_K = \frac{\boldsymbol{\tau}_K|\mathbf{c}||R_h(u_h)|}{|\nabla u_h|}\frac{|\mathbf{c}||\nabla u_h|}{|\mathbf{c}||\nabla u_h| + |R_h(u_h)|}\frac{|\mathbf{c}||\nabla u_h| + |R_h(u_h)| + \mathbf{tan}\alpha_K|\mathbf{c}||\mathbf{c}^{\perp}\cdot\nabla u_h|}{|R_h(u_h)| + \mathbf{tan}\alpha_K|\mathbf{c}||\mathbf{c}^{\perp}\cdot\nabla u_h|}, \tag{10.27}$$

The parameter $\alpha_K$ is equal to $\pi/2 - \beta_K$ where $\beta_K$ is the largest angle of $K$. If $\beta_K = \pi/2$ it is recommended to set $\alpha_K = \pi/6$. The $\boldsymbol{\tau}_K$ is already defined in (10.6).

### 10.2.9 Modified Burman and Ern Method BE02_2

For any $K \in \mathcal{P}$ the BE02_2 [49] definition of $\widetilde{\varepsilon}$ is

$$\widetilde{\varepsilon}|_K = \frac{\boldsymbol{\tau}_K|\mathbf{c}||R_h(u_h)|}{|\nabla u_h|}\frac{|\mathbf{c}||\nabla u_h|}{|\mathbf{c}||\nabla u_h| + |R_h(u_h)|}. \tag{10.28}$$

### 10.2.10 SOLD Methods Error Results

#### 10.2.10.1 2D Error Test: constant convection field

The description of this $\Omega_{2D}$ test example is given in section 10.2.4.1. This section will discuss the error for all the four SOLD methods. The $L^2$-norm and $H^1$-norm errors for the Codina C93 and Codina KLR02_3 are in Table 10.5. For these methods the error reduction rate are approximately 2 and 1 for the $L^2$-norm and $H^1$-norm errors, respectively. The $L^2$-norm error reduction for $\varepsilon = 1E-2$ does not reduce at the required rate in both methods. The error graphs for the C93 and KLR02_3 methods are in Figures 10.11 and 10.12, respectively. These graphs dispaly an optimal reduction rate.

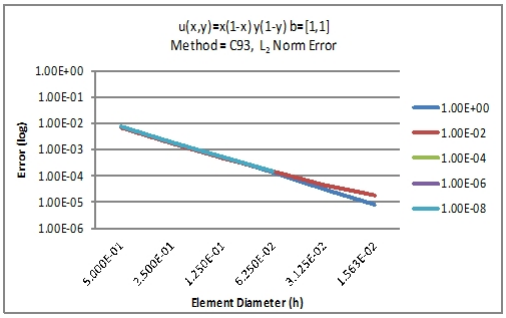
The Burman and Ern methods (BE02_1 and BE02_2) error results are in Table 10.6. The BE02_1 method has converged for all the meshes when the $\varepsilon = E-0$. The $L^2$-norm and $H^1$-norm errors has reduced at the required rate. For smaller $\varepsilon$ values it has converged for sq and sq1 meshes. Even for converged cases the error has not achieved the required reduction rate. The BE02_2 method has converged for all the meshes for the $\varepsilon$ values 1E-0 and 1E-2. This method has not converged for the sq4 and sq5 meshes when the $\varepsilon$ values 1E-4, 1E-6 and 1E-8.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| C93 Error | | | | | | | | | | | | | | | |
| sq | 1 | 6.94E-3 | 5.92E-2 | 8 | 6.64E-3 | 6.10E-2 | 10 | 7.23E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 |
| sq1 | 1 | 1.96E-3 | 3.17E-2 | 10 | 1.75E-3 | 3.25E-2 | 18 | 1.99E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 |
| sq2 | 1 | 5.11E-4 | 1.62E-2 | 10 | 4.86E-4 | 1.65E-2 | 23 | 5.42E-4 | 1.68E-2 | 27 | 5.55E-4 | 1.70E-2 | 27 | 5.55E-4 | 1.70E-2 |
| sq3 | 1 | 1.29E-4 | 8.16E-3 | 6 | 1.48E-4 | 8.23E-3 | 291 | 1.45E-4 | 8.46E-3 | 224 | 1.53E-4 | 8.43E-3 | 228 | 1.54E-4 | 8.43E-3 |
| sq4 | 1 | 3.25E-5 | 4.09E-3 | 3 | 4.31E-5 | 4.10E-3 | N.C | | | N.C | | | N.C | | |
| sq5 | 1 | 8.13E-6 | 2.04E-3 | 1 | 1.75E-5 | 2.05E-3 | N.C | | | N.C | | | N.C | | |
| C93 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.82 | 0.90 | | 1.92 | 0.91 | | 1.86 | 0.92 | | 1.86 | 0.92 | | 1.86 | 0.92 |
| | | 1.94 | 0.97 | | 1.85 | 0.98 | | 1.88 | 0.97 | | 1.85 | 0.96 | | 1.85 | 0.96 |
| | | 1.99 | 0.99 | | 1.72 | 1.00 | | 1.90 | 0.99 | | 1.86 | 1.01 | | 1.85 | 1.01 |
| | | 1.99 | 1.00 | | 1.78 | 1.01 | | | | | | | | | |
| | | 2.00 | 1.00 | | 1.30 | 1.00 | | | | | | | | | |
| KLR02_3 Error | | | | | | | | | | | | | | | |
| sq | 40 | 8.27E-3 | 6.42E-2 | 9 | 6.87E-3 | 6.18E-2 | 10 | 7.23E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 |
| sq1 | 33 | 2.19E-3 | 3.25E-2 | 9 | 1.62E-3 | 3.26E-2 | 18 | 1.99E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 |
| sq2 | 28 | 5.33E-4 | 1.63E-2 | 9 | 3.78E-4 | 1.64E-2 | 25 | 5.46E-4 | 1.69E-2 | 27 | 5.55E-4 | 1.70E-2 | 27 | 5.55E-4 | 1.70E-2 |
| sq3 | 24 | 1.31E-4 | 8.17E-3 | 6 | 1.04E-4 | 8.18E-3 | 191 | 1.44E-4 | 8.41E-3 | 228 | 1.53E-4 | 8.43E-3 | 228 | 1.54E-4 | 8.43E-3 |
| sq4 | 21 | 3.26E-5 | 4.09E-3 | 6 | 3.76E-5 | 4.09E-3 | N.C | | | N.C | | | N.C | | |
| sq5 | 18 | 8.14E-6 | 2.04E-3 | 5 | 1.75E-5 | 2.05E-3 | N.C | | | N.C | | | N.C | | |
| KLR02_3 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.92 | 0.98 | | 2.08 | 0.92 | | 1.86 | 0.92 | | 1.86 | 0.92 | | 1.86 | 0.92 |
| | | 2.04 | 1.00 | | 2.10 | 0.99 | | 1.87 | 0.97 | | 1.85 | 0.96 | | 1.85 | 0.96 |
| | | 2.02 | 1.00 | | 1.86 | 1.00 | | 1.92 | 1.01 | | 1.86 | 1.01 | | 1.85 | 1.01 |
| | | 2.01 | 1.00 | | 1.47 | 1.00 | | | | | | | | | |
| | | 2.00 | 1.00 | | 1.10 | 1.00 | | | | | | | | | |

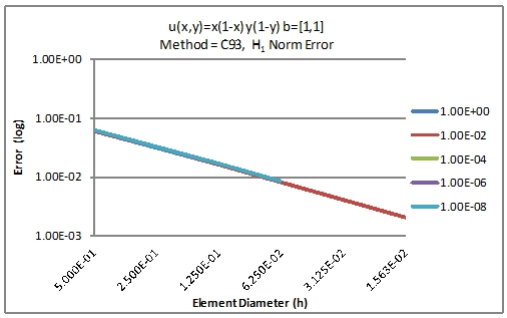
Table 10.5: The table contains all the SOLD methods error data and rate of error reduction rate for the $\Omega_{2D}$ Test Example with fix convection field.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| BE02_1 Error | | | | | | | | | | | | | | | |
| sq | 4 | 8.34E-3 | 6.00E-2 | 629 | 2.14E-2 | 9.91E-2 | 282 | 9.16E-3 | 9.77E-2 | 193 | 1.76E-2 | 1.02E-1 | 673 | 2.32E-2 | 1.20E-1 |
| sq1 | 3 | 2.44E-3 | 3.18E-2 | 694 | 8.28E-3 | 6.98E-2 | 3348 | 1.40E-2 | 1.38E-1 | 4929 | 7.48E-3 | 8.18E-2 | 4936 | 1.09E-2 | 8.37E-2 |
| sq2 | 2 | 6.41E-4 | 1.62E-2 | N.C | | | 50219 | 9.00E-3 | 1.43E-1 | N.C | | | N.C | | |
| sq3 | 2 | 1.63E-4 | 8.16E-3 | N.C | | | N.C | | | N.C | | | N.C | | |
| sq4 | 1 | 4.08E-5 | 4.09E-3 | N.C | | | N.C | | | N.C | | | N.C | | |
| sq5 | 1 | 1.02E-5 | 2.04E-3 | N.C | | | N.C | | | | | | N.C | | |
| BE02_1 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.77 | 0.92 | | 1.37 | 0.51 | | -0.61 | -0.50 | | 1.23 | 0.32 | | 1.09 | 0.52 |
| | | 1.93 | 0.97 | | | | | | | | | | | | |
| | | 1.98 | 0.99 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |
| BE02_2 Error | | | | | | | | | | | | | | | |
| sq | 4 | 7.94E-3 | 5.97E-2 | 12 | 7.64E-3 | 6.18E-2 | 13 | 7.57E-3 | 6.24E-2 | 13 | 7.57E-3 | 6.24E-2 | 13 | 7.57E-3 | 6.24E-2 |
| sq1 | 3 | 2.30E-3 | 3.17E-2 | 11 | 2.23E-3 | 3.27E-2 | 27 | 2.18E-3 | 3.30E-2 | 28 | 2.18E-3 | 3.30E-2 | 28 | 2.18E-3 | 3.30E-2 |
| sq2 | 2 | 6.02E-4 | 1.62E-2 | 18 | 7.03E-4 | 1.67E-2 | 40 | 6.44E-4 | 1.71E-2 | 47 | 6.44E-4 | 1.71E-2 | 47 | 6.44E-4 | 1.71E-2 |
| sq3 | 2 | 1.53E-4 | 8.16E-3 | 16 | 2.73E-4 | 8.42E-3 | 741 | 1.86E-4 | 8.51E-3 | 951 | 1.86E-4 | 8.52E-3 | 955 | 1.86E-4 | 8.52E-3 |
| sq4 | 1 | 3.83E-5 | 4.09E-3 | 6 | 1.26E-4 | 4.25E-3 | N.C | | | N.C | | | N.C | | |
| sq5 | 1 | 9.59E-6 | 2.04E-3 | 3 | 6.15E-5 | 2.14E-3 | N.C | | | N.C | | | N.C | | |
| BE02_2 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.79 | 0.91 | | 1.78 | 0.92 | | 1.80 | 0.92 | | 1.80 | 0.92 | | 1.80 | 0.92 |
| | | 1.93 | 0.97 | | 1.67 | 0.97 | | 1.76 | 0.95 | | 1.76 | 0.95 | | 1.76 | 0.95 |
| | | 1.98 | 0.99 | | 1.36 | 0.99 | | 1.79 | 1.01 | | 1.79 | 1.01 | | 1.79 | 1.01 |
| | | 2.00 | 1.00 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |

Table 10.6: The table contains all the SOLD methods error data and rate of error reduction rate for the $\Omega_{2D}$ Test Example with fixed convectional field.

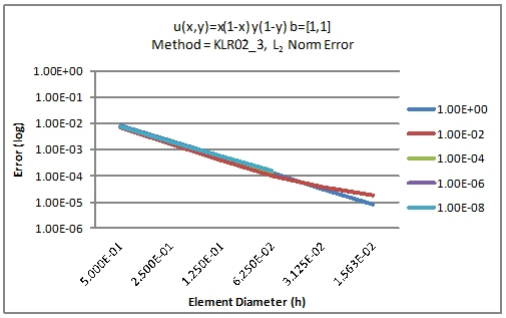



(a) $L^2 - Norm$

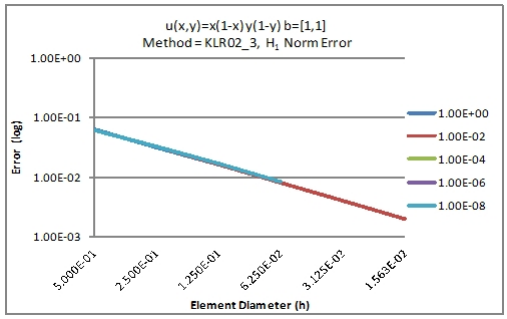



(b) $H^1 - Norm$

Figure 10.11: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using C93 method. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

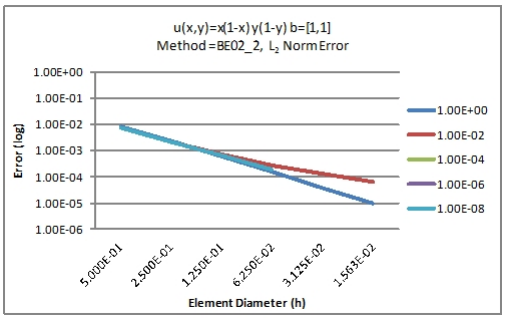



(a) $L^2 - Norm$

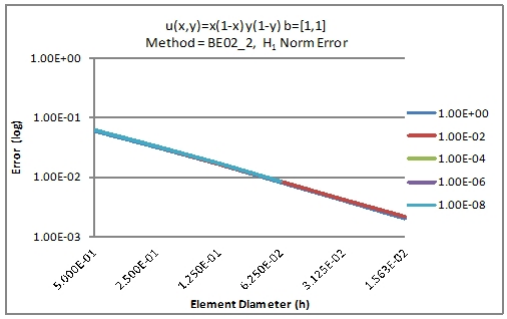



(b) $H^1 - Norm$

Figure 10.12: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using KLR02_3 method. The x-axis represents the $h$ and the y-axis represents the errors in log scale.




(a) $L^2 - Norm$




(b) $H^1 - Norm$

Figure 10.13: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using BE02_2 method. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

#### 10.2.10.2 2D Error Test: a variable convection field

The description of this $\Omega_{2D}$ test example is given in section 10.2.4.2. This section will discuss the error results obtained by computing solution using the SOLD methods. The $L^2$-norm and $H^1$-norm errors for the Codina C93 and Codina KLR02_3 are in Table 10.7. For these methods the error reduction rates are approximately 2 and 1 for the $L^2$-norm and $H^1$-norm errors, respectively. The C93 and KLR02_3 methods have displayed similar behaviour. These methods have converged for all the cases except for the sq4 and sq5 meshes when $\varepsilon$ is greater than 1E-02. The error graphs for these C93 and KLR02_3 methods are in Figure 10.14 and Figure 10.15, respectively. The error reduction rate in $L^2$-norm graphs have deviated from the expected rate.

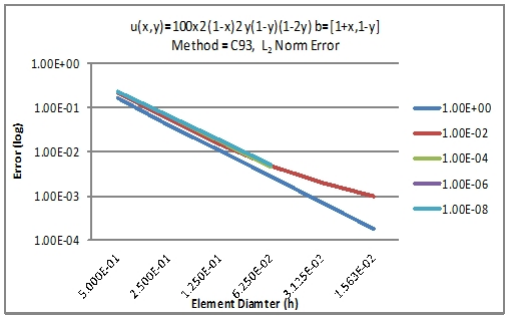
The Burman and Ern methods (BE02_1 and BE02_2) error results are in Table 10.8. The BE02_1 method has converged for all the meshes when the $\varepsilon = 1$. The $L^2$-norm and $H^1$-norm errors has reduced at required rate. For smaller $\varepsilon$ values it has converged for sq, sq1 and sq2 meshes. Even for these meshes the errors have not achieved the required reduction rate. The BE02_2 method has converged for all the meshes for the $\varepsilon = 1$. This method has not converged for the sq3, sq4 and sq5 meshes when the $\varepsilon$ values less than 1. The BE02_2 method error reduction graphs are shown in Figure 10.16. In both the graphs the error reduction rate has deviated for bigger meshes using lower values of $\varepsilon$.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| C93 Error | | | | | | | | | | | | | | | |
| sq | 1 | 1.64E-1 | 1.42 | N.C | 2.21E-1 | 1.52 | 2409 | 2.31E-1 | 1.58 | 3138 | 2.32E-1 | 1.58 | 3144 | 2.32E-1 | 1.58 |
| sq1 | 1 | 4.10E-2 | 7.15E-1 | 155 | 5.56E-2 | 7.43E-1 | 3171 | 6.59E-2 | 7.91E-1 | 47911 | 6.61E-2 | 7.92E-1 | 1889 | 6.61E-2 | 7.92E-1 |
| sq2 | 1 | 1.10E-2 | 3.58E-1 | 17 | 1.46E-2 | 3.72E-1 | 19673 | 1.79E-2 | 3.90E-1 | 23574 | 1.82E-2 | 3.94E-1 | 23390 | 1.82E-2 | 3.94E-1 |
| sq3 | 1 | 2.82E-3 | 1.79E-1 | 14 | 4.76E-3 | 1.84E-1 | 176 | 4.48E-3 | 1.88E-1 | 231 | 4.92E-3 | 1.98E-1 | 230 | 4.93E-3 | 1.98E-1 |
| sq4 | 1 | 7.08E-4 | 8.96E-2 | 6 | 2.04E-3 | 9.14E-2 | N.C | | | N.C | | | N.C | | |
| sq5 | 1 | 1.77E-4 | 4.48E-2 | 1 | 9.83E-4 | 4.59E-2 | N.C | | | N.C | | | N.C | | |
| C93 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 2.00 | 0.99 | | 1.99 | 1.03 | | 1.81 | 1.00 | | 1.81 | 1.00 | | 1.81 | 1.00 |
| | | 1.90 | 1.00 | | 1.93 | 1.00 | | 1.88 | 1.02 | | 1.86 | 1.01 | | 1.86 | 1.01 |
| | | 1.96 | 1.00 | | 1.62 | 1.02 | | 2.00 | 1.05 | | 1.89 | 0.99 | | 1.88 | 0.99 |
| | | 1.99 | 1.00 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |
| KLR02_3 Error | | | | | | | | | | | | | | | |
| sq | 40 | 8.27E-3 | 6.42E-2 | 9 | 6.87E-3 | 6.18E-2 | 10 | 7.23E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 | 10 | 7.24E-3 | 6.23E-2 |
| sq1 | 33 | 2.19E-3 | 3.25E-2 | 9 | 1.62E-3 | 3.26E-2 | 18 | 1.99E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 | 18 | 2.00E-3 | 3.30E-2 |
| sq2 | 28 | 5.33E-4 | 1.63E-2 | 9 | 3.78E-4 | 1.64E-2 | 25 | 5.46E-4 | 1.69E-2 | 27 | 5.55E-4 | 1.70E-2 | 27 | 5.55E-4 | 1.70E-2 |
| sq3 | 24 | 1.31E-4 | 8.17E-3 | 6 | 1.04E-4 | 8.18E-3 | 191 | 1.44E-4 | 8.41E-3 | 228 | 1.53E-4 | 8.43E-3 | 228 | 1.54E-4 | 8.43E-3 |
| sq4 | 21 | 3.26E-5 | 4.09E-3 | 6 | 3.76E-5 | 4.09E-3 | N.C | | | N.C | | | N.C | | |
| sq5 | 18 | 8.14E-6 | 2.04E-3 | 5 | 1.75E-5 | 2.05E-3 | N.C | | | N.C | | | N.C | | |
| KLR02_3 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.92 | 0.98 | | 2.08 | 0.92 | | 1.86 | 0.92 | | 1.86 | 0.92 | | 1.86 | 0.92 |
| | | 2.04 | 1.00 | | 2.10 | 0.99 | | 1.87 | 0.97 | | 1.85 | 0.96 | | 1.85 | 0.96 |
| | | 2.02 | 1.00 | | 1.86 | 1.00 | | 1.92 | 1.01 | | 1.86 | 1.01 | | 1.85 | 1.01 |
| | | 2.01 | 1.00 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |

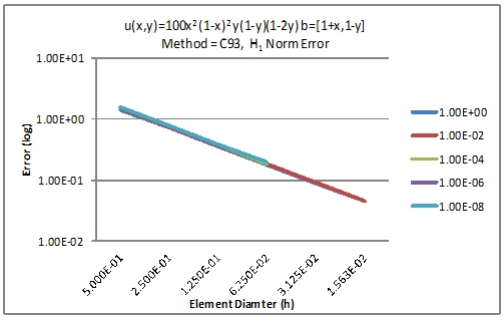
Table 10.7: The table contains all the SOLD methods error data and rate of error reduction rate for the $\Omega_{2D}$ Test Example with converging convection field.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| BE02_1 Error | | | | | | | | | | | | | | | |
| sq | 10 | 1.67E-1 | 1.40 | 1079 | 2.21E-1 | 1.51 | 12 | 2.45E-1 | 1.65 | 12 | 2.45E-1 | 1.65 | 12 | 2.45E-1 | 1.65 |
| sq1 | 4 | 4.34E-2 | 7.12E-1 | 1623 | 1.66E-1 | 1.42 | 1384 | 1.31E-1 | 1.57 | 2710 | 1.27E-1 | 1.34 | 384 | 1.65E-1 | 1.40 |
| sq2 | 3 | 1.18E-2 | 3.58E-1 | 7740 | 1.27E-1 | 2.18 | 46172 | 1.49E-1 | 1.53 | 11944 | 1.54E-1 | 2.01 | 23669 | 1.43E-1 | 1.57 |
| sq3 | 3 | 3.02E-3 | 1.79E-1 | N.C | | | N.C | | | N.C | | | N.C | | |
| sq4 | 6 | 7.63E-4 | 8.96E-2 | N.C | | | N.C | | | N.C | | | N.C | | |
| sq5 | 27 | 1.90E-4 | 4.48E-2 | N.C | | | N.C | | | N.C | | | N.C | | |
| BE02_1 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.94 | 0.98 | | 0.41 | 0.09 | | 0.90 | 0.07 | | 0.95 | 0.30 | | 0.57 | 0.24 |
| | | 1.88 | 0.99 | | 0.39 | -0.62 | | -0.19 | 0.04 | | -0.28 | -0.58 | | 0.21 | -0.17 |
| | | 1.97 | 1.00 | | | | | | | | | | | | |
| | | 1.98 | 1.00 | | | | | | | | | | | | |
| | | 2.01 | 1.00 | | | | | | | | | | | | |
| BE02_2 Error | | | | | | | | | | | | | | | |
| sq | 4 | 1.67E-1 | 1.39 | 13 | 2.19E-1 | 1.51 | 12 | 2.22E-1 | 1.54 | 12 | 2.22E-1 | 1.54 | 12 | 2.22E-1 | 1.54 |
| sq1 | 3 | 4.30E-2 | 7.12E-1 | 18 | 7.61E-2 | 8.14E-1 | 22 | 7.30E-2 | 8.19E-1 | 22 | 7.30E-2 | 8.19E-1 | 22 | 7.30E-2 | 8.19E-1 |
| sq2 | 3 | 1.16E-2 | 3.58E-1 | 37 | 2.58E-2 | 4.14E-1 | 63 | 2.20E-2 | 4.11E-1 | 65 | 2.20E-2 | 4.11E-1 | 65 | 2.20E-2 | 4.12E-1 |
| sq3 | 2 | 2.97E-3 | 1.79E-1 | 33 | 1.04E-2 | 2.09E-1 | 335 | 6.22E-3 | 2.06E-1 | 536 | 6.23E-3 | 2.07E-1 | 542 | 6.23E-3 | 2.07E-1 |
| sq4 | 2 | 7.48E-4 | 8.96E-2 | 14 | 5.00E-3 | 1.07E-1 | N.C | | | N.C | | | N.C | | |
| sq5 | 1 | 1.87E-4 | 4.48E-2 | 7 | 2.45E-3 | 5.46E-2 | N.C | | | N.C | | | N.C | | |
| BE02_2 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 1.96 | 0.97 | | 1.52 | 0.89 | | 1.60 | 0.91 | | 1.60 | 0.91 | | 1.60 | 0.91 |
| | | 1.89 | 0.99 | | 1.56 | 0.98 | | 1.73 | 0.99 | | 1.73 | 0.99 | | 1.73 | 0.99 |
| | | 1.97 | 1.00 | | 1.31 | 0.99 | | 1.82 | 1.00 | | 1.82 | 0.99 | | 1.82 | 0.99 |
| | | 1.99 | 1.00 | | | | | | | | | | | | |
| | | 2.00 | 1.00 | | | | | | | | | | | | |

Table 10.8: The table contains all the SOLD methods error data and rate of error reduction rate for the $\Omega_{2D}$ Test Example with converging convection field.

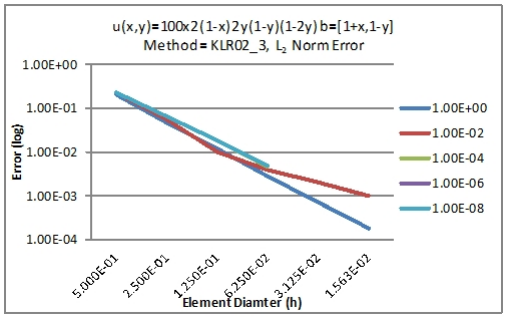



(a) $L^2 - Norm$

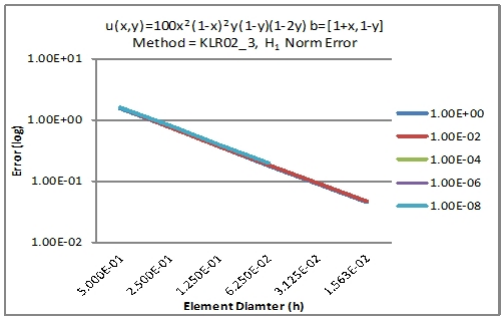



(b) $H^1 - Norm$

Figure 10.14: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using C93 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

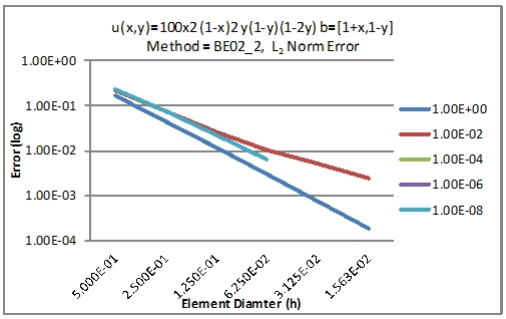



(a) $L^2 - Norm$

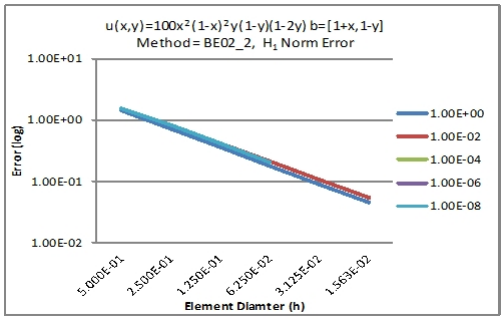



(b) $H^1 - Norm$

Figure 10.15: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using KLR02_3 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.




(a) $L^2 - Norm$




(b) $H^1 - Norm$

Figure 10.16: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{2D}$ obtained using BE02_2 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

#### 10.2.10.3 3D Error Test: a variable convectional field

The description of this $\Omega_{3D}$ test example is given in section 10.2.4.3. This section will discuss the error results obtained by computing solution using the SOLD methods. The $L^2$-norm and $H^1$-norm errors for the Codina C93 and Codina KLR02_3 are in Table 10.9. For these methods the reduction rates are approximately 2 and 1 for the $L^2$-norm and $H^1$-norm errors, respectively. The C93 method has converged for all the meshes and $\varepsilon$ values except for one case. It has not converged for the cb5 mesh for $\varepsilon < 1$. The KLR02_3 method has converged for the cb1 for all the $\varepsilon$ values. It has converged into for all the meshes for $\varepsilon = 1$. It has not converged for cb5 for $\varepsilon < 1$. The method has converged for both methods at the required rate. The $L^2$-norm error graphs for C93 and KLR02_3 methods are depicted in Figures 10.17 and 10.18, respectively.

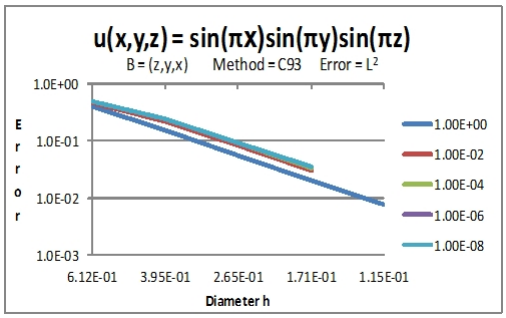
The Burman and Ern methods (BE02_1 and BE02_2) error results are in Table 10.10. The BE02_1 method has converged only for cb mesh. The BE02_2 method has converged for all the meshes for the all the $\varepsilon$ values. There are few exceptions as the method has not converged for the cb5 mesh for $\varepsilon < 0.01$. The BE02_2 method error reduction graphs are depicted in Figure 10.19. The errors have reduced at the required rates.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| C93 Error | | | | | | | | | | | | | | | |
| cb1 | 1 | 4.04E-1 | 1.19 | 12 | 4.60E-1 | 1.26 | 16 | 4.81E-1 | 1.29 | 15 | 4.82E-1 | 1.29 | 15 | 4.82E-1 | 1.29 |
| cb2 | 1 | 1.51E-1 | 7.18E-1 | 15 | 2.14E-1 | 8.16E-1 | 3063 | 2.39E-1 | 8.81E-1 | 2209 | 2.40E-1 | 8.82E-1 | 1171 | 2.40E-1 | 8.82E-1 |
| cb3 | 1 | 5.48E-2 | 4.35E-1 | 35355 | 8.12E-2 | 4.88E-1 | 14291 | 9.07E-2 | 5.73E-1 | 16442 | 9.13E-2 | 5.75E-1 | 15730 | 9.13E-2 | 5.75E-1 |
| cb4 | 1 | 2.03E-2 | 2.66E-1 | 9595 | 2.97E-2 | 2.87E-1 | 31253 | 3.37E-2 | 3.78E-1 | 1270 | 3.42E-2 | 3.83E-1 | 557 | 3.42E-2 | 3.83E-1 |
| cb5 | 1 | 7.66E-3 | 1.63E-1 | N.C | | | N.C | | | N.C | | | N.C | | |
| C93 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 2.25 | 1.15 | | 1.75 | 0.99 | | 1.60 | 0.86 | | 1.59 | 0.86 | | 1.59 | 0.86 |
| | | 2.53 | 1.25 | | 2.42 | 1.28 | | 2.43 | 1.08 | | 2.42 | 1.07 | | 2.42 | 1.07 |
| | | 2.27 | 1.13 | | 2.29 | 1.22 | | 2.26 | 0.95 | | 2.24 | 0.93 | | 2.24 | 0.93 |
| | | 2.44 | 1.23 | | | | | | | | | | | | |
| KLR02_3 Error | | | | | | | | | | | | | | | |
| cb1 | 29 | 4.45E-1 | 1.24 | 11 | 4.60E-1 | 1.26E | 15 | 4.81E-1 | 1.29 | 15 | 4.82E-1 | 1.29 | 15 | 4.82E-1 | 1.29 |
| cb2 | 22 | 1.60E-1 | 7.30E-1 | 18 | 2.04E-1 | 8.07E-1 | N.C | | | N.C | | | N.C | | |
| cb3 | 23 | 5.61E-2 | 4.37E-1 | 14652 | 7.05E-2 | 4.78E-1 | 22032 | 9.03E-2 | 5.73E-1 | 15545 | 9.13E-2 | 5.75E-1 | N.C | | |
| cb4 | 31 | 2.04E-2 | 2.66E-1 | 60556 | 2.68E-2 | 2.83E-1 | N.C | | | N.C | | | N.C | | |
| cb5 | 28 | 7.66E-3 | 1.63E-1 | N.C | | | N.C | | | N.C | | | N.C | | |
| KLR02_3 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 2.33 | 1.21 | | 1.86 | 1.01 | | 1.60 | 0.86 | | 1.59 | 0.86 | | 1.59 | 0.86 |
| | | 2.63 | 1.28 | | 2.66 | 1.32 | | 2.44 | 1.08 | | 2.42 | 1.07 | | | |
| | | 2.32 | 1.13 | | 2.21 | 1.20 | | | | | | | | | |
| | | 2.45 | 1.23 | | | | | | | | | | | | |

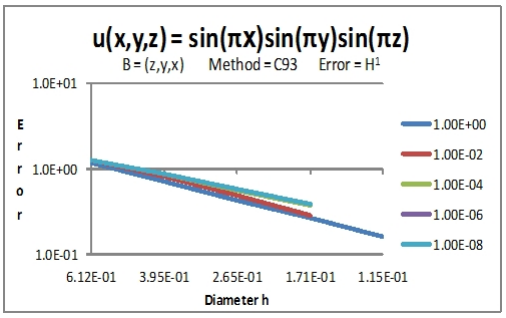
Table 10.9: The table contains C93 and KLR02_3 SOLD methods error data and rate of error reduction for the $\Omega_{3D}$ Test Example with converging convectional field.

| Mesh | $\varepsilon$ = 1E-0 | | | $\varepsilon$ = 1E-2 | | | $\varepsilon$ = 1E-4 | | | $\varepsilon$ = 1E-6 | | | $\varepsilon$ = 1E-8 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err | Iter | $L^2$ Err | $H^1$ Err |
| BE02_1 Error | | | | | | | | | | | | | | | |
| cb1 | 39 | 4.11E-1 | 9.96E-1 | 1044 | 3.48E-1 | 9.96E-1 | 1564 | 4.62E-1 | 9.99E+1 | 1730 | 4.43E-1 | 9.98E-1 | 2103 | 3.81E-1 | 9.96E-1 |
| cb2 | 196 | 1.53E-1 | 1.02 | N.C | | | N.C | | | N.C | | | N.C | | |
| cb3 | N.C | | | N.C | | | N.C | | | N.C | | | N.C | | |
| BE02_1 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 2.26 | -0.06 | | | | | | | | | | | | |
| BE02_2 Error | | | | | | | | | | | | | | | |
| cb1 | 4 | 4.13E-1 | 1.19 | 13 | 4.88E-1 | 1.27 | 14 | 4.87E-1 | 1.28 | 14 | 4.87E-1 | 1.28 | 14 | 4.87E-1 | 1.28 |
| cb2 | 3 | 1.55E-1 | 7.18E-1 | 12 | 2.59E-1 | 8.55E-1 | 16 | 2.46E-1 | 8.73E-1 | 16 | 2.46E-1 | 8.73E-1 | 16 | 2.46E-1 | 8.73E-1 |
| cb3 | 2 | 5.63E-2 | 4.35E-1 | 14 | 1.19E-1 | 5.36E-1 | 35 | 9.95E-2 | 5.76E-1 | 35 | 9.94E-2 | 5.77E-1 | 35 | 9.94E-2 | 5.77E-1 |
| cb4 | 2 | 2.07E-2 | 2.66E-1 | 13 | 5.27E-2 | 3.23E-1 | 1066 | 3.84E-2 | 3.87E-1 | 662 | 3.84E-2 | 3.90E-1 | 665 | 3.84E-2 | 3.90E-1 |
| cb5 | 1 | 7.77E-3 | 1.63E-1 | 10 | 2.40E-2 | 1.91E-1 | N.C | | | N.C | | | N.C | | |
| BE02_2 Error Reduction Rate | | | | | | | | | | | | | | | |
| | | 2.24 | 1.16 | | 1.45 | 0.91 | | 1.56 | 0.88 | | 1.56 | 0.88 | | 1.56 | 0.88 |
| | | 2.54 | 1.25 | | 1.95 | 1.17 | | 2.27 | 1.04 | | 2.27 | 1.04 | | 2.27 | 1.04 |
| | | 2.28 | 1.13 | | 1.86 | 1.16 | | 2.18 | 0.91 | | 2.17 | 0.90 | | 2.17 | 0.90 |
| | | 2.46 | 1.23 | | 1.96 | 1.31 | | | | | | | | | |

Table 10.10: The table contains BE02_1 and BE02_2 SOLD methods error data and rate of error reduction for the $\Omega_{3D}$ Test Example with converging convectional field.

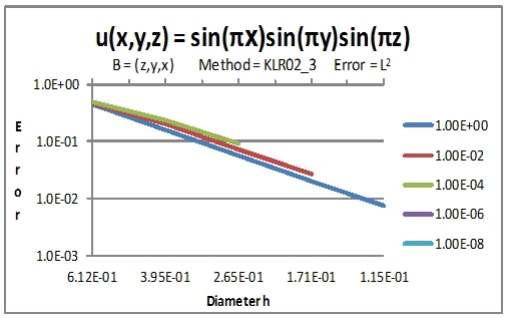



(a) $L^2 - Norm$

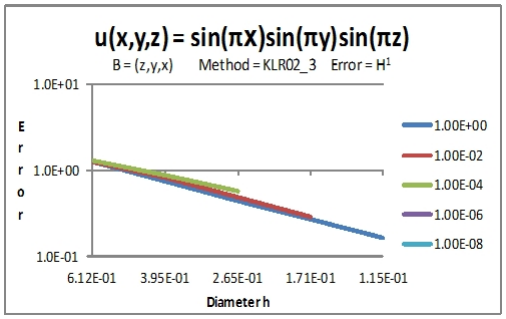



(b) $H^1 - Norm$

Figure 10.17: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{3D}$ obtained using C93 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

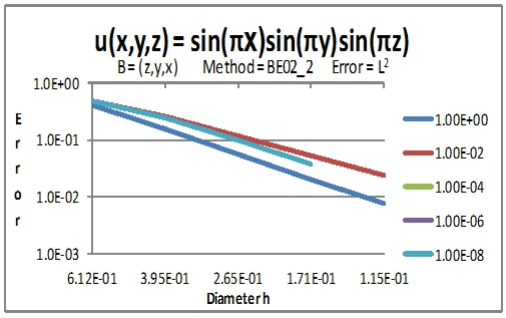



(a) $L^2 - Norm$

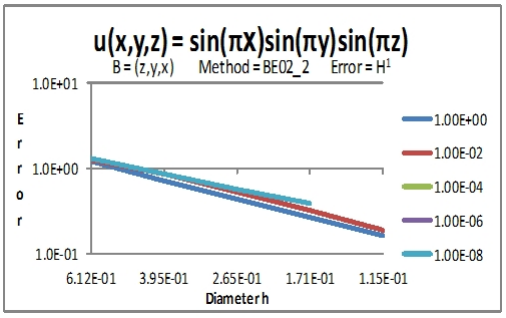



(b) $H^1 - Norm$

Figure 10.18: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{3D}$ obtained using KLR02_3 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.




(a) $L^2 - Norm$



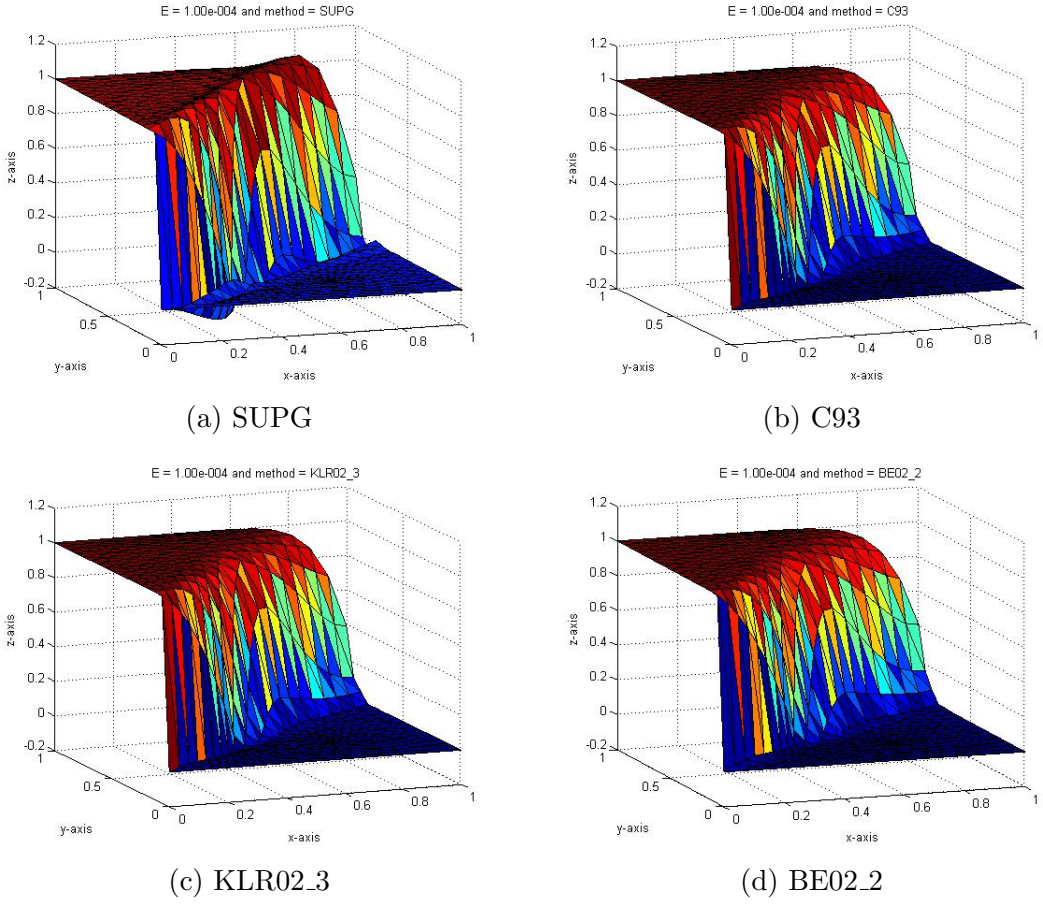

(b) $H^1 - Norm$

Figure 10.19: The figure have the $L^2$-norm and $H^1$-norm errors graphs for $\Omega_{3D}$ obtained using BE02_2 method for a variable convection field. The x-axis represents the $h$ and the y-axis represents the errors in log scale.

### 10.2.11 Problem with internal and boundary layers

#### 10.2.11.1 SOLD 2D Layer problem constant convection field

This section compares the solution of SOLD methods. The test example used in this section is given in section 10.2.5.1. The SUPG and SOLD methods solutions are given in Figure 10.20 for $\Omega_{2D}$ using sq4 mesh and $\varepsilon = 1E-4$.




(a) SUPG

(b) C93

(c) KLR02_3

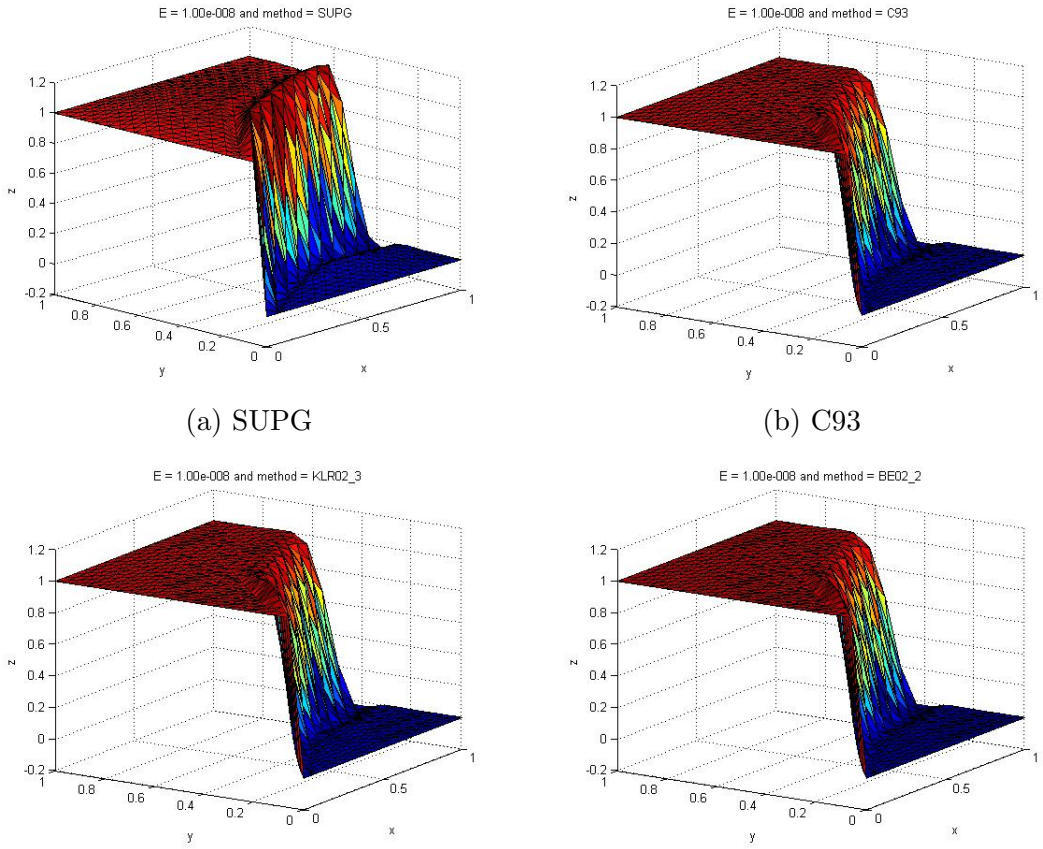
(d) BE02_2

Figure 10.20: The figure have the solution visualization of the SUPG, C93, KLR02_3 and BE02_2 methods solutions for the constant convective field. The oscillations present in the SUPG solution have beeb removed by the SOLD methods.

#### 10.2.11.2 SOLD 2D Layer problem variable convection field

This section compares the solution of SOLD methods. The test example used in this section is given in section 10.2.5.2. The SUPG and SOLD methods solutions

are given in Figure 10.21 for $\Omega_{2D}$ using sq4 mesh using $\varepsilon = 1E-8$.



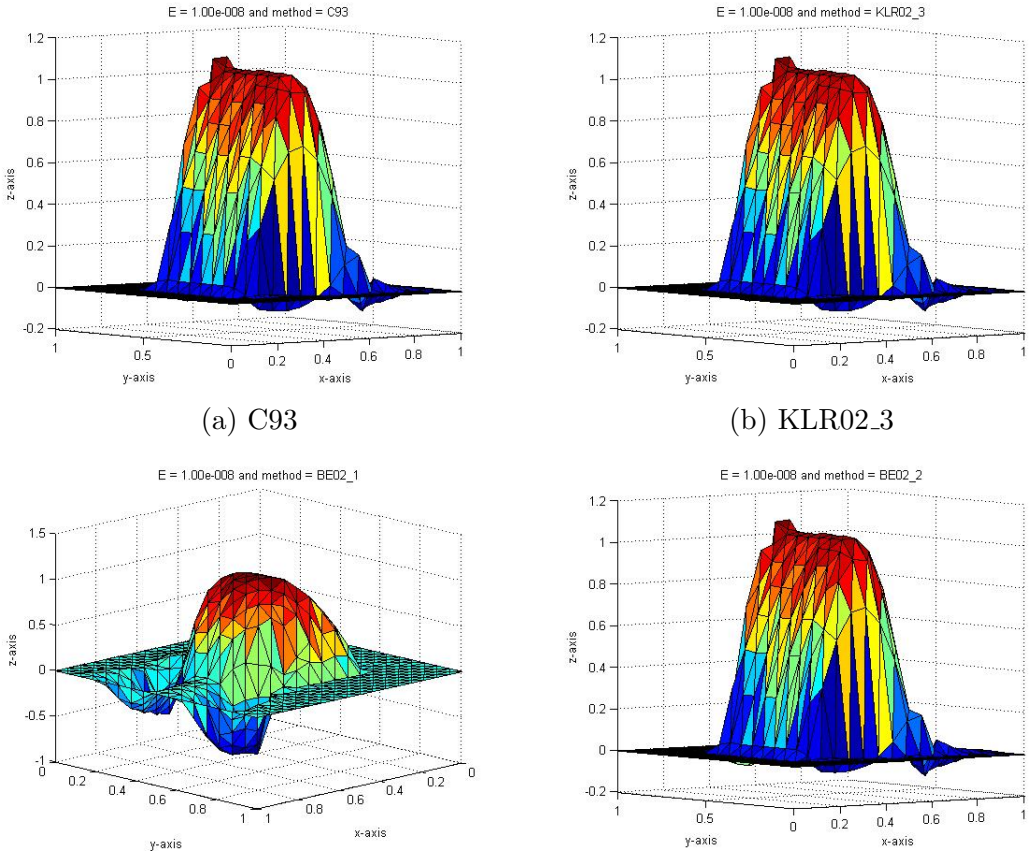

(a) SUPG (b) C93

(c) KLR02_3 (d) BE02_2

Figure 10.21: The figure have the solution visualization of the SUPG, C93, KLR02_3 and BE02_2 methods solutions for the variable convection field. The oscillations present in the SUPG solution have reduced around the inner and outer boundary of the curve in the SOLD methods.

The SOLD methods have removed the unwanted osculations from the first SOLD 2D layer problem can be seen in Figure 10.20. In the second problem the SOLD method has reduced the oscillations to minimum as shown in Figure 10.21. The SOLD methods have produced desired results but these are computational expensive as compare to SUPG method. These have to solve the system of linear equations in every iteration.

#### 10.2.11.3 SOLD 2D Layer problem 3




(a) C93 (b) KLR02_3

(c) BE02_1 (d) BE02_2

Figure 10.22: Elevation of the solution provided by the C93, KLR02_3, BE02_1 and BE02_2 methods for the constant convectional field.

This section compares the solution of SOLD methods. The test example used in this section is given in Section 10.2.5.3. The SUPG and SOLD methods solutions are given in Figure 10.22 for $\Omega_{2D}$ using sq3 mesh using $\varepsilon = 1E-8$. The C93, KLR02_3 and BE02_2 solutions have reduced the oscillations in $[0.25, 0.75]^2$ but also has added the oscillations in outer regions. The discrete solution produced by the SOLD methods are not desired solutions as the extra oscillations are present in the region (0.75,1)×(0,1) which are absent in the actual solutions given as Figure 10.7. The huge undesired oscillation in the BE02_1 solution make this solution least desirable.

# Chapter 11

# Conclusion and Future Work

## 11.1 Conclusion

The objectives of this work are given in Section 1.2. This section explains how these objectives are achieved.

1. *Simple interface:* The user selects the mathematical problem and domain data and provides these to FEDomain as a list of elements' objects. The FEDomain collects element objects' data and assembles it according to the target third party solvers to compute solution. The solution is then provided to the element objects. The user does not have to implement parallel data assembly and does not have to know about third party linear algebra solver specifications. FEDomain maintains the interface between the user application and the third party linear algebra solver.

2. *Generic interface:* The user implements C++ classes for each type of finite element. The implementation of the element classes depends on the problem type (Poisson, Elasticity, etc), the shape of the element (triangle, quadrangle, tetrahedron, hexahedron, prism, pyramid etc), and the dimension of the mesh (2D, 3D, etc). The FEDomain package provides an interface class called FEElement defined in Listing 3.16. The user has to inherit element

classes from the FEElement class. The FEElement class standardize the data access interface between the FEDomain package and mesh element. This design allows the user to implement any type of constraints (Neumann, Dirichlet, Robin etc) and degrees of freedoms (point, fluxes, means, etc).

The simple interface of FEDomain package introduces generality in implementation. This interface divides the finite element application into two segments. The first segment is FEDomain responsibility which includes assembling of system data and calculation of the solution. The second segment includes the user's specific tasks, such as selection of domain and its dimensions, problem types, element topologies, etc. The FEElement class enhances this generality as it allows user to implement any element topology. The first segment is not affected by this user's choice and computes solution.

3. *Operating system independent:* FEDomain package is implemented in standard C++. It uses OpenMP, MKL and MPI libraries which are available for all commonly used operating systems. FEDomain is not operating system dependent. It can be used on commonly used operating systems like Linux, Windows, and Mac OS, etc.

4. *Support multiple solution methods:* The user's mathematical problem and domain data is dynamic and can be changed at runtime. The FEDomain package's interface selection is kept dynamic so that user can select appropriate solver at runtime.

5. *Support for shared and distributed architectures:* The FEDomain package is implemented for shared and distributed architecture machines. FEDomain shared memory interface is given in Listing 3.14 and the distributed memory interface in given in Listing 3.17. The objective was to design these interfaces as simple as possible. In both cases the user has to provide a set of mesh element objects, a list of Dirichlet DOFs, partition ids, type of

matrix, and solution methods to package constructor. The FEDomain and FEDomainMPI classes have two methods. The getResidual method is to support iterative linear algebra solver and setSolution method is to compute the solution using linear algebra direct solver methods. The user can convert his sequential finite element application C++ code into parallel by adding FEDomain into their application. FEDomain will perform data assembly and calculation in parallel. The conversion from shared memory architecture code into distributed memory code only requires altering a few lines. This involves initiation and finalization of MPI and calling FEDomainMPI interface instead of FEDomain interface.

6. *Solution distribution:* The FEElement class defines the interface between the FEDomain package and user's element objects. In this interface a setSolution function is defined which provides solution to element objects. The solution provided to element is already mapped according to its local DOF numbering.

7. *Extendible:* The FEDomain package allows its users to select and implement linear algebra iterative solvers like Jacobi, Conjugate Gradients, etc. To help with this it provides the calculation of the residual vector, which is a very computationally intensive task. This can be done in both scenarios (shared and distributed architectures).

The aim given in Section 1.1 is achieved as all the above mentioned objectives are fulfilled. The package and element interfaces proposed in this work can support all finite element problems. The proposed distinction between user and package domains provides the user independence to define its problem. It also enable the package to provide solvers for these problems.

## 11.2 Future Work

The FEDomain package uses the third party linear algebra solvers PARDISO and MUMPS to compute the solution of finite element problem. The support to the other well known linear algebra solver like PETSc, and SuperLU should be added. This will allows the FEDomain user to select the available linear algebra solver.

The calculation of the $\boldsymbol{S}_{bb}^{P}$ algorithm has to be reimplemented to increase the performance of the DIS_DIRECT_SOLVER, and DIS_HYBRID_SOLVER methods. It is the most time consuming task in DIS_HYBRID_SOLVER methods.

The FEDomain package non linear solver has to be implemented for the distributed memory architectures. The shared memory implementation of the non linear solver has to be reimplemented for better memory management.

As every solution and residual method can be tuned appropriately to get a better performance. Auto tuning strategies will be explored in future work.

# Appendix A

# FEDomain Installation Guide

This section illustrates how to use FEDomain package. It includes a Poisson 2D example as well as demonstrates how the application can be converted into Elasticity 3D solver.

## A.1 Requirements

The FEDomain package is developed for the shared memory and the distributed memory architectures. The FEDomain supports PARDISO and MUMPS linear algebra libraries. The shared memory version of these libraries is required by the package and is not provided as the part of the package. The open source PARDISO solver can be downloaded from www.pardiso-project.org. The user can purchase the paid version of PARDISO solver from Intel® as a part of Math Kernel Library (MKL). The MUMPS solver can be downloaded from http://mumps.enseeiht.fr.

The FEDomain package is provided as source code as well as the compiled libraries. The user can obtain the package by email the author at omer.riaz@strath.ac.uk. The FEDomain library has to be compiled according to the operating system. The user has to mention the target operating system in the email.

The FEDomain package implements parallel regions using OpenMP pragma. The pragmas are the directives for the compiler to implement parallel code regions. The user has to use the C++ compiler which supports OpenMP pragma. The commonly used Windows operating system compilers are Microsoft Visual C++ compiler and Intel C++ compiler. For Linux and Mac OS operating systems GNU C++ and Intel C++ compilers can be used. The Microsoft Visual C++ and GNU C++ compilers are free compilers that can be obtained online. The list of other OpenMP compilers can be seen at http://openmp.org/wp/openmp-compilers.

## A.2 FEDomain Preprocessors

The FEDomain is developed for for the 32-bit and 64-bit machines. It uses FE_UINT for indexing type and FE_DATA for data type. The FEDomain package has to be indicated by setting preprocessors from Table A.1 at compile time.

| Architecture | Preprocessor | FE_UINT | FE_DATA |
|---|---|---|---|
| 32-bit | FE_32 | unsigned int | double |
| 64-bit | FE_64 | unsigned long long | double |

Table A.1: The data type and preprocessor table.

The FEDomain package support two third party linear algebra packages. The user has to select these packages at the compile time. Table A.2 shows the preprocessors for these processors.

| Package | Preprocessor |
|---|---|
| PARDISO | FE_PARDISO |
| MUMPS | FE_MUMPS |

Table A.2: The linear algebra package and their preprocessors.

## A.3 Poisson 2D element classes examples

In this section the Poisson 2D example will be solved on the square domain $\Omega = (0,1)^2$ given in Fig A.1. The domain is triangulated into edge and triangle elements. The edge elements lie at the domain boundaries which will contribute to the right hand side. We define the Dirichlet boundary conditions on $\Gamma_1$ and $\Gamma_2$ and Neumann conditions on $\Gamma_3$ and $\Gamma_4$. The data for the boundary conditions and right hand sizes are

$$\begin{aligned} \text{on } \Gamma_3 \quad &: \partial_h u &&= sin(\pi x) \\ \text{on } \Gamma_4 \quad &: \partial_h u &&= sin(\pi y) \\ \text{in } \Omega \quad &: f &&= 1 \end{aligned}$$

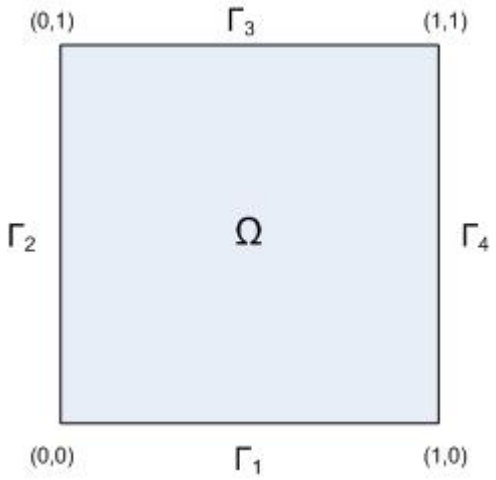



Figure A.1: Poisson 2D domain.

This section discusses the Poisson 2D example code. The code in Listing A.1 represents the user application. In line 14 the Element_Factory class object is created. This class has to be defined by the user. The Element_Factory object is responsible for creating element objects. The constructor returns the list of elements, nodes and partition ids. The Problem class in line 17 defines the problem

specific data. This includes the force applied on the domain, the type of boundaries and what are the Neumann and Dirichlet values on the boundaries. The FEDomain object is created into in Line 19. All the element objects are created to FEDomain object to compute the solution in using DIRECT_SOLVER. In the following line, the solution is computed and provided back to the elements.

```
1  #include "FEDomain/FEDomain.h"
2  #include "FEDomain/FEElement.h"
3  int main(){
4      string mesh_file = "file_path/file_name";
5      FE_UINT Max_Ele_DOFs;
6      FE_UINT Total_Sys_DOFs;
7      std::vector<NODES> Nodes;
8      std::vector<FEElement*> Elements;
9      std::vector<FE_UINT> Partitions;
10     std::map<FE_UINT,FE_DATA> Dirichlet;
11     // Read mesh adapter
12     Mesh_Adapter adapter(mesh_file);
13     // Create node and mesh elements.
14     Element_Factory Factory(adapter);
15     Factory.getData(Nodes,Elements,Partitions);
16     // Apply Poisson problem data (specify boundary types and applied forces).
17     Poisson2D Data(Elements,Nodes,Dirichlet,Total_Sys_DOFs,Max_Ele_DOFs);
18     // Creating FEDomain object using DIRECT_SOLVER.
19     FEDomain Domain(Elements,Dirichlet,Partitions,Total_Sys_DOFs,
20                     Max_Ele_DOFs,DIRECT_SOLVER,DEFINITE_SYMMETRIC);
21     // Computing solution.
22     Domain.setSolution(Dirichlet);
23     // Perform post processing (like visualization).
24     return 0;
25 }
```

Listing A.1: User Application

The Mesh_Reader class in Listing A.2 is implemented to read mesh files. It is good practice to have adapter for different mesh formats.

```
class Mesh_Reader{
public:
    enum ELEMENT_TYPE = {EDGE=1, TRIANGLE=2};
    Mesh_Reader(string mesh_path) { .. };
    bool getPoint(std::vector<FE_DATA>& point) { .. };
    bool getElement(ELEMENT_TYPE& element_type, FE_UINT& partition_id, FE_UINT& domain_id,
                    std::vector<FE_UINT>& ele_data){ .. };
}
```

Listing A.2: Mesh Reader

The example of Element_Factory class is given in Listing A.3. The objective of this class is to create element and node objects. Listing A.3 shows the prototype of the class implementation. In constructor stage, the Element_Factory have to open the mesh file and setup internal data structures. The getData method provides mesh nodes, elements and partition ids.

```
1  class Element_Factory{
2  private:
3      // Collect all nodes and create objects for these nodes.
4      void getNodes(std::vector<NODES> &nodes){
5          // Construct all node objects.
6          std::vector<FE_DATA> coordinate(2);
7          while (reader.getNode(coordinate))
8              nodes.pushback(coordinate);
9      };
10     // Create objects for mesh elements and collect partition ids.
11     void getElements(std::vector<NODES> &nodes, std::vector<FEElement*> &elements,
12                      std::vector<FE_UINT> &partIDs){
13         FE_UINT pid, did;
14         std::vector<FE_DATA> data(5);
15         Mesh_Adapter::ELEMENT_TYPE type;
16         std::map<FE_UINT,FE_UINT> gpid;
17         while (reader.getElement(type,pid,did,data))
18         {
19             gpid[pid]=0;
20             if (type == EDGE){
21                 // Create and collect edge elements.
22                 Poisson2DEdge *e = new Poisson2DEdge(nodes,data,did,pid);
23                 element.push_back(e);
24             }
25             else if (type == TRIANGLE){
26                 // Create and collect triangle elements
27                 Poisson2DTriangle *e = Poisson2DTriangle(nodes,data,did,pid);
28                 element.push_back(e);
29             }
30             else if (type == TETRAHEDRON){
31                 // Create and collect tetrahedron elements. It will be used in 3d mesh.
32             }
```

```
33          }
34          // Collect partition ids.
35          std::transform(gpid.begin(),gpid.end(),back_inserter(partIDs),RetrieveKey());
36      };
37  public:
38      Element_Factory(Mesh_Reader& mesh_file){/*Open File*/};
39      ~Element_Factory(){};
40      void getData (std::vector<NODES>& nodes, std::vector<FEElement*>& elements,
41                    std::vector<FE_UINT>& partIDs){
42          getNodes(nodes);
43          getElements(nodes,elements,partIDs);
44      };
45  }
```

Listing A.3: Element Factory

The Poisson2D class will provide the problem data to the element objects. The prototype of this class is given in Listing A.6. This class defines the problem constants, function to calculate force on $\Omega$, the type of the domain boundaries and how to calculate load on these boundaries. The Element_Factory and Poisson2D classes cannot be implemented by FEDomain package. Their implementation will depend on the user's choice of problem, and domain geometry, etc.. The TFunctor and TSpecificFunctor classes given in Listing A.4 are implemented to provide function pointers to the element classes. A new abstract class called Element (given in Listing A.5) is introduced to collect domain identification id from the user. The user has to add Element class (given in Listing A.5 ) as the FEElement class does not provide any method to object Domain identification id. This information is required to set Functor to the element class. The getElementID() in Element class provides element identification number which will be unique for each element class.

```
// Abstract Functor class.
class TFunctor{
public:
    virtual double operator()(double,double,double,int) = 0;
};
// Specific Functor class.
template <class TClass>
class TSpecificFunctor : public TFunctor{
private:
    TClass* pt2Object;
    double (TClass::*fpt)(double,double,double,int);
public:
    TSpecificFunctor(TClass *_pt2Object, double (TClass::*_fpt)(double,double,double,int))
    {
```

```
        pt2Object = _pt2Object;
        fpt = _fpt;
    };
    virtual double operator()(double x, double y, double z, int i)
    {   return (*pt2Object.*fpt)(x,y,z,i);  };
};
```

Listing A.4: Abstract Functor and Specific Functor Class

```
class Element : public FEElement
{
public:
    virtual int getElementID() = 0;
    virtual int getDomainID() = 0;
}
```

Listing A.5: User Element Abstract Class

```
1  class Poisson2D{
2  private:
3      TSpecificFunctor <Poisson2D> *E1, *E2, *E3, *E4, *D1;
4
5      void setElement(std::vector<FEElement*> &Elements){
6          FE_UINT did;
7          Element* pElement;
8          for (FE_UINT i=0; i<Elements.size(); i++){
9              pElement = Elements[i];
10             did = pElement->getDomainID();
11             if (did == 1){
12                 Poisson2DEdge *e = pElement[i];
13                 e->setForce(E1);
14             }
15             else if (did == 2){
16                 Poisson2DEdge *e = pElement[i];
17                 e->setForce(E2);
18             }
19             else if (did == 3){
20                 Poisson2DEdge *e = pElement[i];
21                 e->setForce(E3);
22             }
23             else if (did == 4){
24                 Poisson2DEdge *e = pElement[i];
25                 e->setForce(E4);
26             }
27             else if (did == 5){
28                 Poisson2DTriangle *e = pElement[i];
29                 e->setForce(D1);
30                 e->setCoefficient(1);
31             }
32         }
33     };
34     void setTotalSysDOFs(std::vector<NODES> &Nodes, FE_UINT &Total_Sys_DOFs) {
35         // Compute total DOfs in the system.
```

```
36          Total_Sys_DOFs = Nodes.size();
37      };
38      void setMaxEleDOFs(std::vector<FEElement*> &Elements, FE_UINT &Max_Ele_DOFs) {
39          FE_UINT count = 0;
40          Max_Ele_DOFs = 0;
41          for (i=0; i<Elements.size(); i++){
42              count = Elements[i].getDofsCount();
43              if (count > Max_Ele_DOFs)
44                  Max_Ele_DOFs = count;
45          }
46      };
47      void setDirichlet(std::vector<FEElement*> &Elements, std::map<FE_UINT, FE_DATA> &Dirichlet){
48         int did;
49         for(FE_UINT i=0; i<Elements.size(); i++){
50              Element* e = Elements[i];
51              eid = e->getElementID();
52              if (eid == 1){
53                  did = e->getDomainID();
54                  if (did == 1 || did == 2){
55                      std::vector<FE_UINT> dofs(e->getDofsCount());
56                      e->getConnectivity(dofs);
57                      for (FE_UINT i=0; i<dofs.size(); i++)
58                          Dirichlet[dofs[i]] = 0;
59                  }
60              }
61          }
62      };
63
64  public:
65      Poisson2D(std::vector<FEElement*> &Elements, std::vector<NODES> &Nodes,
66                std::map<FE_UINT,FE_DATA>& Dirichlet, FE_UINT &Total_Sys_DOFs,
67                FE_UINT &Max_Ele_DOFs){
68          E1 = new TSpecificFunctor<Poisson2D>(this, Poisson2D::ForceAt1);
69          E2 = new TSpecificFunctor<Poisson2D>(this, Poisson2D::ForceAt2);
70          E3 = new TSpecificFunctor<Poisson2D>(this, Poisson2D::ForceAt3);
71          E4 = new TSpecificFunctor<Poisson2D>(this, Poisson2D::ForceAt4);
72          D1 = new TSpecificFunctor<Poisson2D>(this, Poisson2D::ForceAt5);
73          setTotalSysDOFs(Nodes,Total_Sys_DOFs);
74          setMaxEleDOFs(Elements,Max_Ele_DOFs);
75          setElement(Elements);
76          setDirichlet(Elements,Dirichlet);
77       };
78      ~Poisson2D(){ delete E1; delete E2; delete E3; delete E4; delete D1; };
79      // Force applied on Γ1.
80      double ForceAt1(double x, double y, double z, int i) { return 0.0; };
81      // Force applied on Γ2.
82      double ForceAt2(double x, double y, double z, int i) { return 0.0; };
83      // Force applied on Γ3.
84      double ForceAt3(double x, double y, double z, int i) { return sin(180*x); };
85      // Force applied on Γ4.
86      double ForceAt4(double x, double y, double z, int i) { return sin(180*y); };
87      // Force applied on Ω.
88      double ForceAt5(double x, double y, double z, int i) { return 1.0; };
89  }
```

Listing A.6: Poisson2D Class

The Edge element class called Poisson2DEdge is given in Listing A.7. The edge element has two nodes and two DOFs. The class is inherited from the FEElement class (given in Listing 3.16). The edge element lies at the domain boundary. These contribute only to the load vector. The getLoadAt method in Poisson2DEdge class calculate load vector using Simpson's rule.

```
class Poisson2DEdge : public Element
{
private:
    FE_DATA eLength;                    // Length of Edge
    TFunctor* f_Force;                  // Functor to the force function
    FE_UINT TotalDofs;                  // Total DOFs in element.
    FE_UINT ePartIDs;                   // Element partition id (default=0)
    FE_UINT eDomainIDs;                 // Element domain id (default=0)
    std::vector<NODES> &Nodes;          // List of mesh nodes
    std::vector<FE_UINT> nIDs;          // List of element node ids
    std::vector<FE_UINT> eDOFs;         // List of element DOFs
    double getLoadAt(FE_UINT &idx){
        // ∫_l G(s)ds ≃ |l|/6 (G(x0)λ(x0) + 4G(m)λ(m) + G(x1)λ(x1))
        assert(idx < TotalDofs);
        double load = 2 * f_Force * ((Nodes[nIDs[0]].x + Nodes[nIDs[1]].x)/2,
                                     (Nodes[nIDs[0]].y + Nodes[nIDs[1]].y)/2, 0,0);
        load += f_Force (Nodes[nIDs[idx]].x, Nodes[nIDs[idx]].y,0,0);
        return (e_Length / 6) * load;
    };
public:
    Poisson2DEdge(vector<FE_UINT>& node_ids, vector<NODES>& mesh_nodes, FE_UINT& domain_id,
                  FE_UINT& partition_id):nIDs(node_ids), Nodes(mesh_nodes), TotalNodes(2),
                  eDomainIDs(domain_id), ePartIDs(partition_id){  .. };
    ~Poisson2DEdge(void){ .. };
    void setForce(TFunctor* function) { f_Force = function; };
    void assemble(std::map<FE_UINT,FE_DATA>& dirichlet) { .. };
    virtual int getDomainID() { return eDomainIDs; }
    virtual int getElementID() { return eType; }
    virtual FE_UINT getDofsCount() { return eDOFs.size(); };
    virtual FE_UINT getPartitionID() { return ePartIDs };
    virtual void getLoad(FEVector& v) {
        for (FE_UINT i=0; i<TotalDofs; i++)  v[i] = getLoadAt(i);
    };
    virtual void getStiffness(FEMatrix& m) {  };
    virtual void getSystem(FEEquation& e){
       for (FE_UINT i=0; i<TotalDofs; i++)  e.setValue(eDOFs[i],getLoadAt(i));
    };
    virtual void getConnectivity(FESparseMatrix& m) {
        for (FE_UINT i=0; i<TotalDofs; i++)  m.setValue(i,eDOFs[i],1);
    };
    virtual void getConnectivity(std::vector<FE_UINT>& v) {
        for (FE_UINT i=0; i<TotalDofs; i++)  v[i] = eDOFs[i];
    };
    virtual void setSolution(FEVector&) { ... };
};
```

Listing A.7: Poisson 2D Edge Class

The Poisson2DTriangle class in Listing A.8 implements the formulation of the triangle elements for Poisson 2D problem. The Poisson2DTriangle have three nodes and three DOFs. The load function is provided as the functor which is provided by Poisson2D class. The nodes and node_ids are allocated at the construction time. The triangle load vector is calculated using the mid-points rule. The triangle stiffness matrix is calculated using tangent vectors on triangle edges.

```
class Poisson2DTriangle : public Element
{
    FE_DATA eArea;                  // Area of triangle
    FE_DATA ePoisson;               // Poisson Eq coefficient
    FE_UINT TotalDofs;                  // Total DOFs in element.
    TFunctor* f_Force;              // Functor to the force function
    FE_UINT ePartIDs;               // Element partition id (default=0)
    FE_UINT eDomainID;              // Element domain id (default=0)
    std::vector<NODES> &Nodes;      // List of mesh nodes
    std::vector<FE_UINT> nIDs;      // List of element node ids
    std::vector<FE_UINT> eDOFs;     // List of element DOFs
    std::vector<NODES> eMPoint;     // List of edges mid point
    std::vector<NODES> eTangent;    // List of tangents

    FE_DATA getLoadAt(FE_UINT& idx) {
        // \int_K F(x)\lambda_j(x)dx \simeq \frac{|K|}{3}(F(m_0)\lambda_j(m_0)+F(m_1)\lambda_j(m_1)+F(m_2)\lambda_j(m_2))
        assert( idx < TotalDofs);
        FE_DATA load = 0;
        if (idx != 0) load += f_Force(eMPoint[0].x,eMPoint[0].y,0,0);
        if (idx != 1) load += f_Force(eMPoint[1].x,eMPoint[1].y,0,0);
        if (idx != 2) load += f_Force(eMPoint[2].x,eMPoint[2].y,0,0);
        return (e_Area / 6) * load;
    };

    FE_DATA getStiffnessAt(FE_UINT& i, FE_UINT& j) {
        // B_K(\lambda_j,\lambda_i) \simeq \frac{a_K}{4|K|}(\vec{t}_1.\vec{t}_j)
        assert((i<TotalDofs)&&(j<TotalDofs));
        FE_DATA data = eTangent[j].y * eTangent[i].y + eTangent[j].x * eTangent[i].x;
        return data * ePoisson / ( 4 * eArea);
    };

public:
    Poisson2DTriangle(std::vector< FE_UINT >& node_ids, std::vector< NODES >& mesh_nodes,
                      FE_UINT &domain_id, FE_UINT& partition_id):nIDs(node_ids),
                      Nodes(mesh_nodes), eDomainID(domain_id), ePartIDs(partition_id),
                      TotalDofs(3) { .. };
    ~Poisson2DTriangle(void) { .. };
    void assemble(std::map<FE_UINT,FE_DATA>& dirichlet) { .. };
    void setForce(TFunctor* function) { f_Force = function; };
    void setCoefficient( FE_DATA& coefficient) { ePoisson = coefficient; };
    virtual int getDomainID() { return eDomainID; }
    virtual int getElementID() { return eType; }
    virtual FE_UINT getDofsCount() { return eDOFs.size(); };
    virtual FE_UINT getPartitionID() { return ePartIDs; };
    virtual void getLoad(FEVector& v) {
        for (FE_UINT i=0; i<TotalDofs; i++) v[i] = getLoadAt(i);
    };
```

```
48      virtual void getStiffness(FEDenseMatrix& m) {
49          double value = 0;
50          for (FE_UINT i=0; i<TotalDofs; i++)
51              for (FE_UINT j=i; j<TotalDofs; j++){
52                  value = getStiffnessAt(i,j);
53                  m.addValue(i,j,value);
54                  m.addValue(j,i,value);
55              }
56      };
57      virtual void getSystem(FE_equation& e) {
58          double value = 0;
59          for (FE_UINT i=0; i<TotalDofs; i++){
60              e.setValue(eDOFs[i],getLoadAt(i));
61              for (FE_UINT j=i; j<TotalDofs; j++){
62                  value = getStiffnessAt(i,j);
63                  e.setValue(eDOFs[i],eDOFs[j],value);
64                  e.setValue(eDOFs[j],eDOFs[i],value);
65              }
66          }
67      };
68      virtual void getConnectivity(FESparseMatrix& m) {
69          for (FE_UINT i=0; i<TotalDofs; i++) m.setValue(i,eDOFs[i],1);
70      };
71      virtual void getConnectivity(std::vector<FE_UINT>& v) {
72          for (FE_UINT i=0; i<TotalDofs; i++) v[i] = eDOFs[i];
73      };
74      virtual void setSolution(FEVector&) { .. };
75  };
```

Listing A.8: Poisson 2D Triangle Class

NODES class in the above classes are used to represents nodes. It encapsulates x and y coordinates of the point. NODES class is given in Listing A.9.

```
1  class NODES
2  {   `
3  public:
4      FE_DATA x, y, z;
5      NODES(NODES& n):x(n.x),y(n.y),z(n.z){};
6      NODES(FE_DATA &_x, FE_DATA &_y):x(_x),y(_y),z(0){};
7      NODES(FE_DATA &_x, FE_DATA &_y, FE_DATA &_z):x(_x),y(_y),z(_z){};
8      NODES(std::vector<FE_DATA> &n):x(n[0]),y(n[1]),z(n[2]){};
9      ~NODES(){};
10 }
```

Listing A.9: NODES Class

In this chapter an example of the code to solve the 2D Poisson problem is given. The code is complete up to the calculation of geometry. Specific quantities such as length of the edge, area of the triangle, and tangent vectors. These should be implemented by the user.

## A.4 Elasticity 3D Modifications

Apart from geometry and dimension specific changes, such as the calculation of the normals and quadrature rules etc, the following are the only changes needed to solve an Elasticity 3D problem. The fact that very few changes are needed in order to apply the system to a different problem domain demonstrates its effectiveness.

The implementation of the Elasticity 3D problem requires the implementation of the problem definition classes like Poisson2D, Poisson2DEdge and Poisson2DTriangle classes. Let the 3D domain is triangulated into triangle and tetrahedron elements. The boundary elements have triangle shape and these will contribute only for the right hand side. Let Elasticity3DTriangle class is the name of the boundary elements. This class will be similar to the Poisson2DEdge class given in Listing A.7 with small modifications. For Elasticity 3D, each nodal point will have 3 DOFs so triangle element will have 9 DOFs. For internal identifications the DOFs allocated to single node are labelled with consecutive ids. The getLoadAt function will be changed to the getLoadAt function given in Listing A.10.

```
1  FE_DATA getLoadAt(FE_UINT& idx) {
2      assert( idx < TotalDofs);
3      FE_UINT node_id = idx/DOFsPerNode;
4      FE_UINT dof_id  = idx%DOFsPerNode;
5      FE_DATA x,y,z;
6      x = e_MPoints[node_id].x;
7      y = e_MPoints[node_id].y;
8      z = e_MPoints[node_id].z;
9      return (*f_Force)(x,y,z,dof_id)*e_Area/3.0;
10 }
```

Listing A.10: Elasticity 3D Triangle Class load method.

Let Elasticity3DTetrahedron class is the name of the tetrahedron element. This class will be similar to Poisson2DTriangle class given in Listing A.8. The tetrahedron element will have 12 DOFs and the DOFs allocated to single node are internally labelled consecutive ids. There will be two Elasticity equation constants $\lambda$ and $\mu$ represented in this class as e_Lambda and e_Mu respectively. The load

vector and stiffness matrix implementation for this class is given in Listing A.11.In Elasticity3DTetrahedron class the normals are used to calculate stiffness matrix.

```
FE_DATA getLoadAt(FE_UINT& idx) {
    assert(idx < eTotalDofs);
    FE_UINT node_pos = idx / DOFsPerNode;
    FE_UINT dof_pos  = idx % DOFsPerNode;
    FE_DATA f = 0;
    FE_DATA l = 0;
    for (FE_UINT i=0; i<TotalNodes; i++){
        f = (*f_Force)((*p_node)[e_NodeIDs[i]].x, (*p_node)[e_NodeIDs[i]].y,
                       (*p_node)[e_NodeIDs[i]].z, dof_pos);
        if (node_pos == i) l += f / 10;
        else               l += f / 20;
    }
    return l * e_Volume;
};

FE_DATA getStiffnessAt(FE_UINT& r, FE_UINT& c) {
    assert((r < eTotalDofs)&&(c < eTotalDofs));
    FE_DATA value = 0;
    FE_UINT i, j, fi, fj;
    i  = r % DOFsPerNode;        fi = r / DOFsPerNode;
    j  = c % DOFsPerNode;        fj = c / DOFsPerNode;
    if (fi == fj)
    {
        if (fi == 0)
            value = ( 2 * e_Mu * ( e_Normals[j].x * e_Normals[i].x
                  + e_Normals[j].y * e_Normals[i].y / 2
                  + e_Normals[j].z * e_Normals[i].z / 2 )
                  + e_Lambda * ( e_Normals[j].x * e_Normals[i].x ));
        else if (fi == 1)
            value = ( 2 * e_Mu * ( e_Normals[j].x * e_Normals[i].x / 2
                  + e_Normals[j].y * e_Normals[i].y
                  + e_Normals[j].z * e_Normals[i].z / 2 )
                  + e_Lambda * ( e_Normals[j].y * e_Normals[i].y ));
        else if (fi == 2)
            value = ( 2 * e_Mu * ( e_Normals[j].x * e_Normals[i].x / 2
                  + e_Normals[j].y * e_Normals[i].y / 2
                  + e_Normals[j].z * e_Normals[i].z )
                  + e_Lambda * ( e_Normals[j].z * e_Normals[i].z ));
    }
    else
    {
        if ((fi == 0) && (fj == 1))
            value = (e_Mu * e_Normals[j].x * e_Normals[i].y
                  +  e_Lambda * e_Normals[j].y * e_Normals[i].x );
        else if ((fi == 0) && (fj == 2))
            value = (e_Mu * e_Normals[j].x * e_Normals[i].z
                  +  e_Lambda * e_Normals[j].z * e_Normals[i].x );
        else if ((fi == 1) && (fj == 0))
            value = (e_Mu * e_Normals[j].y * e_Normals[i].x
                  +  e_Lambda * e_Normals[j].x * e_Normals[i].y );
        else if ((fi == 1) && (fj == 2))
            value = (e_Mu * e_Normals[j].y * e_Normals[i].z
                  +  e_Lambda * e_Normals[j].z * e_Normals[i].y );
        else if ((fi == 2) && (fj == 0))
            value = (e_Mu * e_Normals[j].z * e_Normals[i].x
                  +  e_Lambda * e_Normals[j].x * e_Normals[i].z );
```

```
57          else if ((fi == 2) && (fj == 1))
58              value = (e_Mu * e_Normals[j].z * e_Normals[i].y
59                    +  e_Lambda * e_Normals[j].y * e_Normals[i].z );
60      }
61      return value/(9 * e_Volume);
62  };
```

Listing A.11: Elasticity 3D Tetrahedron Class Methods.

# Bibliography


[1] url=http://www.mono-project.com/.

[2] url=http://dotgnu.org/pnet.html.

[3] *Getfem++ an open source finite element library.* http://download.gna.org/getfem/html/homepage/index.html.

[4] *Intel Xeon Processor X5560.* http://ark.intel.com/products/37109/Intel-Xeon-Processor-X5560-8M-Cache-2_80-GHz-6_40-GTs-Intel-QPI.

[5] *Intel Xeon Processor X5650.* http://ark.intel.com/products/47922.

[6] *Object Oriented MPI (OOMPI):A class library for the Massage Passing Interface*, South Bend, IN, 1996. IEEE Computer Society Press.

[7] *Intel Math Kernel Library Reference Manual.* http://www.intel.com, January 2010.

[8] H. Adeli and G. Yu. An integrated computing environment for solution of complex engineering problems using the object-oriented programming paradigm and a blackboard architecture. *Computers & Structures*, 54(2):255 – 265, 1995.

[9] J.E. Akin. *Finite Elements for Analysis and Design.* Computational mathematics and applications. Academic Press, 1994.

[10] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[11] P. R. Amestoy, I. S. Duff, J. Koster, and J.Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[12] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. SIAM, Philadelphia, PA, USA, 1999.

[13] O. Axelsson and V.A. Barker. *Finite Element Solution of Boundary Value Problems Theory and Computation*. SIAM, Philadelphia, PA, USA, 2001.

[14] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. volume 1662 of *LNCS*, pages 760–760. University of Passau, D-94030, Passau, Germany, 1999.

[15] S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kuashik, M. Knepley, L.C. McInnes, B. Smith, and H. Zhang. *PETSc Users Manual*. MCS Division, Argonne National Laboratory, 3.2 edition, Sept 2011.

[16] W. Bangerth. Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in DEAL.II. Proceedings of the IMACS 2000 World Congres, August.

[17] W. Bangerth, R. Hartmann, and G. Kanschat. DEAL.II a general purpose object oriented finite element library. *ACM Transactions on Mathematical Software*, 33, August 2007.

[18] W. Bangerth and G. Kanschat. Concepts for object-oriented finite element software the DEAL.II. In *Preprint 43, SFB 359*, 1999.

[19] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. *Computing*, 82(2-3):103–119, 2008.

[20] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.W. Chin. Phipac: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. Technical report, University of Tennessee, 1996.

[21] G.H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *In Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC5)*, pages 243–252. Society Press, 1996.

[22] S.C. Brenner and L.R. Scott. *The mathematical theory of finite element methods*. Springer, 2008.

[23] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[24] A.N. Brook and T.J.R Hughes. Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 32:199–259, 1982.

[25] P. Ciechanowicz, M. Poldner, and H. Kuchen. The münster skeleton library Muesli - A comprehensive overview. ERCIS Working Paper No. 7, 2009.

[26] M. Cole. Algorithmic skeletons: structured management of parallel computation. *MIT press*, 1991.

[27] T. Davis and I. Duff. An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):140–158, 1997.

[28] E.G.D. doCarmo and A.C. Galeão. Feedback Petrov-Galerkin methods for convection-dominated problems. *Computer Methods in Applied Mechanics and Engineering*, 88:1–16, 1991.

[29] A.J. Dorta, J.A. González, C. Rodríguez, and F. Sande. llc: A parallel skeletal language. *Parallel Processing Letters*, 13(3):437–448, 2003.

[30] J. Duffy. *Concurrent Programming on Windows*. Addison Wesley Professional, 2008.

[31] A. Ern and J.L. Guermond. *Theory and Practice of Finite Elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2004.

[32] B.W.R. Forde, R.O. Foschi, and S.F. Stiemer. Object-oriented finite element analysis. *Computers & Structures*, 34(3):355 – 374, 1990.

[33] L.P. Franca, S.L. Frey, and T.J.R. Hughes. Stabilized finite element methods: I. Application to the advective-diffusive model. *Computer Methods in Applied Mechanics and Engineering 95*, pages 253–276, 1992.

[34] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] K. Gärtner. Mathematical topic "Solution of large sparse linear systems". http://www.wias-berlin.de/research/rts/GlSyst/index.jsp.

[37] L. Gil and G. Bugeda. A c++ object-oriented programming strategy for the implementation of the finite element sensitivity analysis for a non-linear structural material model. *Adv. Eng. Softw.*, 32(12):927–935, November 2001.

[38] V. Girault and P.A. Raviart. *Finite Element Methods for Navier-Stokes equations: Theory and Algorithm*. Springer Series in Computational Mathematics, Springer-Verlag, 1986.

[39] P. Gottschling and C. Steinhardt. Meta-Tuning in MTL4. In *ICNAAM 2010: International Conference of Numerical Analysis and Applied Mathematics*, volume 1281, pages 778–782. American Institute of Physics, 09 2010.

[40] N. I. M. Gould, Y. Hu, and J. A. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical report, Council for the Central Laboratory of the Research Councils, May 2005.

[41] A. Gupta. WSMP: Watson Sparse Matrix Package Part I - direct solution of symmetric sparse systems. Technical report, IBM T. J. Watson Research Center,, 1101 Kitchawan Road, Yorktown Heights, NY 10598, November 2000.

[42] A. Gupta. WSMP: Watson Sparse Matrix Package Part II - direct solution of general sparse systems. Technical report, IBM T. J. Watson Research Center,, 1101 Kitchawan Road, Yorktown Heights, NY 10598, November 2000.

[43] F. Hecht. New development in FreeFEM++. *Journal of Numerical Mathematics*, 20(3-4):251–265, 2012.

[44] B.C.P. Heng and R.I. Mackie. Using design patterns in object-oriented finite element programming. *Comput. Struct.*, 87(15-16):952–961, August 2009.

[45] B.C.P. Heng and R.I. Mackie. Parallel modal analysis with concurrent distributed objects. *Computers & Structures*, 88(2324):1444 – 1458, 2010. Special Issue: Association of Computational Mechanics United Kingdom.

[46] Intel Coorporation. *Intel Math Kernel Library Reference Manual*, 10.2 edition, 2010.

[47] Y. Jinyun. Symmetric gaussian quadrature formulae for tetrahedronal regions. *Computer Methods in Applied Mechanics and Engineering*, 43(3):349 – 353, 1984.

[48] V. John and P. Knobloch. On spurious oscillations at layers diminishing (SOLD) methods for convectiondiffusion equations: Part I - A review. *Computer Methods in Applied Mechanics and Engineering*, 196:2197–2215, November 2007.

[49] V. John and P. Knobloch. On spurious oscillations at layers diminishing (SOLD) methods for convection–diffusion equations: Part II – Analysis for P1 and Q1 finite elements. *Computer methods in applied mechanics and engineering*, 197(21-24):1997–2014, 2008.

[50] C. Johnson. *Numerical solution of partial differential equations by finite element method*. Cambridge University Press, May 1998.

[51] Y. Karasawa and H. Iwasaki. A Parallel Skeleton Library for Multi-core Clusters. In *Parallel Processing, 2009. ICPP '09. International Conference*, pages 84 – 91, Sept 2009.

[52] G. Karypis. *Metis A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* Department of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455, 5 edition, 8 2011. http://www.cs.umn.edu/karypis.

[53] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific computing*, 20(1):359–392, 1998.

[54] B.S. Kirk, J.W. Peterson, R.H. Stogner, and G.F. Carey. libMesh : A C++ library for parallel adaptive mesh refinement coarsen-

ing simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. urlhttp://dx.doi.org/10.1007/s00366-006-0049-3.

[55] X.-A. Kong. A data design approach for object-oriented {FEM} programs. *Computers & Structures*, 61(3):503 – 513, 1996.

[56] X.-A. Kong and D.P. Chen. An object-oriented design of {FEM} programs. *Computers & Structures*, 57(1):157 – 166, 1995.

[57] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[58] X.S. Li. An overview of SuperLU: Algorithms implementation and user interface. *ACM Transactions on Mathematical Software*, 31:302–325, Sept 2005.

[59] J.W.H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions of Mathematical Software*, 11(2):141–153, June 1985.

[60] A. Logg, K.-A. Mardal, and G.N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, http://dx.doi.org/10.1007/978-3-642-23099-8, 2012.

[61] J. Lu, D. White, and W.F. Chen. Applying object-oriented design to finite element programming. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, SAC '93, pages 424–429, New York, NY, USA, 1993. ACM.

[62] R.I. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35(2):425–436, 1992.

[63] R.I. Mackie. An object-oriented approach to fully interactive finite element software. *Advances in Engineering Software*, 29(2):139 – 149, 1998.

[64] R.I. Mackie. Object-oriented finite element programming-the importance of data modelling. *Advances in Engineering Software*, 30(911):775 – 782, 1999.

[65] R.I. Mackie. Implementation of sub-structuring within an object-oriented framework. *Advances in Engineering Software*, 32(1011):749 – 758, 2001.

[66] R.I. Mackie. Object oriented implementation of distributed finite element analysis in .net. *Advances in Engineering Software*, 38(1112):726 – 737, 2007. Engineering Computational Technology.

[67] R.I. Mackie. Object-oriented programming of distributed iterative equation solvers. *Computers & Structures*, 86(6):511 – 519, 2008. Civil-Comp Special Issue.

[68] R.I. Mackie. Advantages of object oriented finite element analysis. *Proceedings of the ICE - Engineering & Computational Mechanics*, 162:23–29, 2009.

[69] R.I. Mackie. Design and deployment of distributed numerical applications using .net and component oriented programming. *Advances in Engineering Software*, 40(8):665–674, 2009.

[70] S. Modak and E.D. Sotelino. An object-oriented programming framework for the parallel dynamic analysis of structures. *Computers & Structures*, 80(1):77 – 84, 2002.

[71] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide, second edition: C++ programming with the standard template library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[72] OpenMP Architecture Review Board, http://openmp.org/wp/. *OpenMP Application Program Interface*, version 3.1 edition, July 2011.

[73] P.S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, November 1996.

[74] B. Patzak and Z. Bittnar. Design of object-oriented finite element code. *Advances in Engineering Software*, 32(10-11):759–767, 2001.

[75] B. Patzak and D. Rypl. Object-oriented, parallel finite element framework with dynamic load balancing. *Advances in Engineering Software*, 47(1):35–50, 2012.

[76] J. Reider. *Intel Threading Building Blocks*. OŔeilly Media, 2007.

[77] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: A High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation, 2000.

[78] S.-P. Scholz. Elements of an object-oriented fem++ program in c++. *Computers & Structures*, 43(3):517 – 529, 1992.

[79] R. Touzani. An object oriented finite element toolkit. In *Proceedings of the Fifth World Congress on Computational Mechanics (WCCM V)*, pages 163–202. Vienna, Vienna University of Technology, Austria, ISBN 3-9501554-0-6, July 2002.

[80] R. Vuduc, J.W. Demmel, and K.A. Yelick. *The Optimized Sparse Kernel Interface (OSKI) Library*. BeBOP, University of California, Berkeley, http://bebop.cs.berkeley.edu/oski, 1.0.1h edition, June 2007.

[81] R.C. Whaley. *Software Automatic Tuning : From Concepts to State-of-the-Arts Results*, chapter Chapter 2 ATLAS Version 3.9: Overview and Status. Springer, 2010.

[82] G. Yu and H. Adeli. Object-oriented finite element analysis using eer model. *Journal of Structural Engineering*, 119(9):2763–2781, 1993.