DEVELOPMENT OF A PARALLEL-EXTENSIBLE GENERIC BOUNDARY ELEMENT METHOD APPLICATION FRAMEWORK

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

BY

HAKAN BAYINDIR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
SOFTWARE ENGINEERING

JUNE 2017

GCPS

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

_____

Prof. Dr. Ali Kara

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

_____

Prof. Dr. Ali Yazıcı

Head of Department

This is to certify that we have read the thesis "Development of a Parallel-Extensible Generic Boundary Element Method Application Framework" submitted by "Hakan BAYINDIR" and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy

_____	_____
Assist. Prof. Dr. Besim Baranoğlu	Prof. Dr. Ali Yazıcı
Co-Supervisor	Supervisor

Examining Committee Members

Prof. Dr. Hasan U. Akay	_____
Prof. Dr. Ali Yazıcı	_____
Assist. Prof. Dr. Ziya Karakaya	_____
Assoc. Prof. Dr. Murat Manguoğlu	_____
Assist. Prof. Dr. Barbaros Çetin	_____

Date: 29.06.2017

I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Hakan Bayındır

Signature:

ABSTRACT

DEVELOPMENT OF A PARALLEL-EXTENSIBLE GENERIC BOUNDARY ELEMENT METHOD APPLICATION FRAMEWORK

Bayındır, Hakan
Ph.D., Software Engineering
Supervisor : Prof. Dr. Ali Yazıcı
Co-Supervisor : Assist. Prof. Dr. Besim Baranoğlu

June 2017, 142 pages

A new framework for the solution of engineering problems using Boundary Element Method is presented. The developed framework presents a new programming model which provides complete problem solutions called *Flow*s, and ways to extend a scientific computing framework, the addition of new capabilities or tuning for special problems are easier by providing programming units to build customized *Flow*s. Presented framework makes implementation of high performance and parallel algorithms easier by introducing a layered and modularized structure. The framework is tested by implementing two *Flows* for elastostatic and Laplace problems, and the framework achieved its goals by solving the problems accurately, using parallel algorithms and with minimum effort.

Keywords: framework, boundary element method, parallel programming

ÖZ

PARALEL, GENİŞLETİLİR VE GENEL AMAÇLI BİR SINIR ELEMAN YÖNTEMİ UYGULAMA ÇERÇEVESİNİN GELİŞTİRİLMESİ

Bayındır, Hakan
Doktora, Yazılım Mühendisliği
Tez Yöneticisi : Prof. Dr. Ali Yazıcı
Ortak Tez Yöneticisi : Yrd. Doç. Dr. Besim Baranoğlu

Haziran 2017 , 142 sayfa

Bu çalışmada Sınır Eleman Yöntemi ile mühendislik problemlerini çözebilecek bir yazılım çerçevesi geliştirilmiştir. Bu çerçeve, problemlerin kolayca çözülmesini sağlayan, *Akış* adı verilen problem çözüm üniteleri ile yeni bir programlama modeli sunmaktadır. Ayrıca; yeteneklerin eklenmesi, özelliklerin probleme özel şekilde optimize edilmesi ve *Akış*ların kolayca oluşturulabilmesi ile ilgili, çözüm adımları sağlayan fonksiyonlar ile yeni yaklaşımlar getirmektedir. Bu yazılım çerçevesi, yüksek performanslı ve paralel yazılım geliştirme süreçlerini kolaylaştırıcı özellikler içermektedir. Çerçeve, lineer elastik ve Laplace problemlerin çözülmesi ile ilgili *Akış*ların programlanması ile test edilmiş, çerçeve, problemleri paralel algoritmalar kullanarak doğru şekilde ve az bir programlama uğraşı ile çözerek hedeflerine ulaşmıştır.

Anahtar Kelimeler: uygulama iskeleti, sınır eleman yöntemi, paralel programlama

to my Grandfather, Family, and Oceanus.

ACKNOWLEDGMENTS

I would like to thank to my supervisor Prof. Dr. Ali Yazıcı and co-supervisor Assist. Prof. Dr. Besim Baranoğlu for their great support.

I would also like thank to my thesis overseeing committee members Assoc. Prof. Dr. Murat Manguoğlu and Assist. Prof. Dr. Ziya Karakaya for their advice and guidance in helping to shape this dissertation.

Lastly, I would like to thank Starbucks Coffee Çayyolu for providing study space and caffeine till the closing time during the intense studying period required for writing this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
B3	Boundary Building Blocks
B3F	Boundary Building Blocks Framework
BC	Boundary Condition
BE	Boundary Element
BEA	Boundary Element Analysis
BEM	Boundary Element Method
BLAS	Basic Linear Algebra Subprograms
BLACS	Basic Linear Algebra Communication Subprograms
CG	Conjugate Gradient Method
CI	Continuous Integration
CPU	Central Processing Unit
DLA	Dense Linear Algebra
FDM	Finite Difference Method
FEM	Finite Element Method
FMM	Fast Multipole Method
FVM	Finite Volume Method
GE	Governing Equation
GPU	Graphics Processing Unit
IPC	Instructions Per Cycle
MLFMM	Multi Level Fast Multipole Method
MPI	Message Passing Interface
OpenBLAS	Optimized BLAS
PVM	Parallel Virtual Machine
RAM	Random Access Memory
ScaLAPACK	Scalable Liner Algebra Package
VCS	Version Control System

CHAPTER 1

INTRODUCTION

The Boundary Element Method (BEM) is a well-established numerical method for the solution of many engineering problems which can be expressed as a set of differential equations [1]. Due to its several advantages over other numerical methods, such as the Finite Element Method (FEM) [1] and the Finite Volume Method (FVM), the BEM drew considerable attention especially in the fields of acoustics, fluid mechanics, electromagnetics and linear elastostatics/elastodynamics. Yet, due to several disadvantages of the method [1], the BEM could not spread widely in common end-users, rather limited itself mostly to the academic research. Aside from a limited number of libraries and software packages [2, 3, 4, 5, 6, 7, 8] which solves a small set of problems or cases, the BEM does not have a widespread commercial and/or open source package which aims the wide-range solution of many engineering problems.

This lack of real interest has several reasons: First, the formulation of the BEM requires the determination of the so-called *fundamental solutions*, which require a mathematical background. Although, nowadays, many engineering problems have pre-determined fundamental solutions, the implementation of the method requires special treatment, especially when it comes to numerical integration of the singular

integrals which arise from these fundamental solutions.

Second, unlike more popular numerical methods of today, the BEM works on dense system matrices. And, third, the components of these dense system matrices are obtained through the evaluation of a large number of integrals. Both the dense matrices, and the evaluation of the integrals involved require more processing power and memory space than today's most popular methods. Processing power of this scale required multi-computer clusters and specialized programming until recent years, which made the BEM less useful in common single-board computers and/or workstations. However with the emergence of multi-core architectures and increasing core counts per Central Processing Unit (CPU) alongside improvements in processing power and memory density, the BEM found its way into working on single-board computers, hence can be used by a larger audience in near future.

Aforementioned computer hardware and the CPU improvements made two important High Performance Computing (HPC) cornerstones mainstream: parallelization and vectorization. With multi-core systems, more expensive multi-CPU systems miniaturized as multi-core systems and became the norm hence, most computers today can do more than one task simultaneously. While available since 2001 for double precision mathematics, vectorization (or vector processing) capabilities of the recent processors are greatly enhanced with AVX [9, 10] and AVX2 [11] instruction sets developed by Intel and available in both Intel and AMD processors. Also, multi-core architectures contain these advanced Floating Point Units (FPUs) in every core, so every core can make double precision vector computations at the same time. These features make computation of these dense matrices and evaluation of integrals much easier for the current computer architectures.

Another advancement in hardware front is the implementation of atomic operations as instructions in CPU, which enables replacement of computationally expensive mutexes, or locks, with much cheaper atomic operations which are handled at CPU instruction level. Translating atomic operations to direct CPU instructions makes thread synchronization and resource sharing more lightweight and effortless, paving way to performance gains in multi-threaded applications by utilizing the CPU more efficiently [12, 13, 14].

With the conversion of GPUs into massively parallel mathematical accelerators much harder mathematical operations can be executed on more compact computer systems. With the increased acceptance of the technology, libraries supporting GPUs are started to be developed more. This two way support accelerated the improvement of the technology and allowed even more problems to be solved on single board computers.

The BEM, because of its structure and solution model, is very well suited to use these advances in the hardware and software. The formulation and implementation of the BEM involves the evaluation of independent integrals that can be performed in parallel and, its large, dense matrices can be computed quickly both in today's CPUs and GPUs. Because of these reasons, it is a plausible to implement the BEM and start to utilize it in solving real application problems.

CHAPTER 2

GENERAL FORMULATION OF THE BOUNDARY ELEMENT METHOD FOR ENGINEERING PROBLEMS EXPRESSED IN LINEAR HOMOGENEOUS DIFFERENTIAL EQUATIONS

The Boundary Element Method (BEM) proves to be very efficient and accurate in most linear differential equations arising from different fields of engineering practice. This chapter expresses the common procedures of the BEM applied to linear homogeneous differential equations. For this, we first refer to Fig 2.1 for general definitions. Recall that, the presented text is completely original and has no direct references, but a vast literature exists in its theory [1, 15, 16, 17].

2.1 Definitions and General Formulation of the Boundary Element Method

Assume a finite, semi-finite or infinite domain defined in 3D (or in 2D) space. This text particularly deals with 3D problems in engineering. This domain will be represented by V and the boundary of the domain will be represented by S. The outward unit direction at any point on the boundary will be given by n and the unit normal in this direction will be denoted by $\mathbf{n}$.

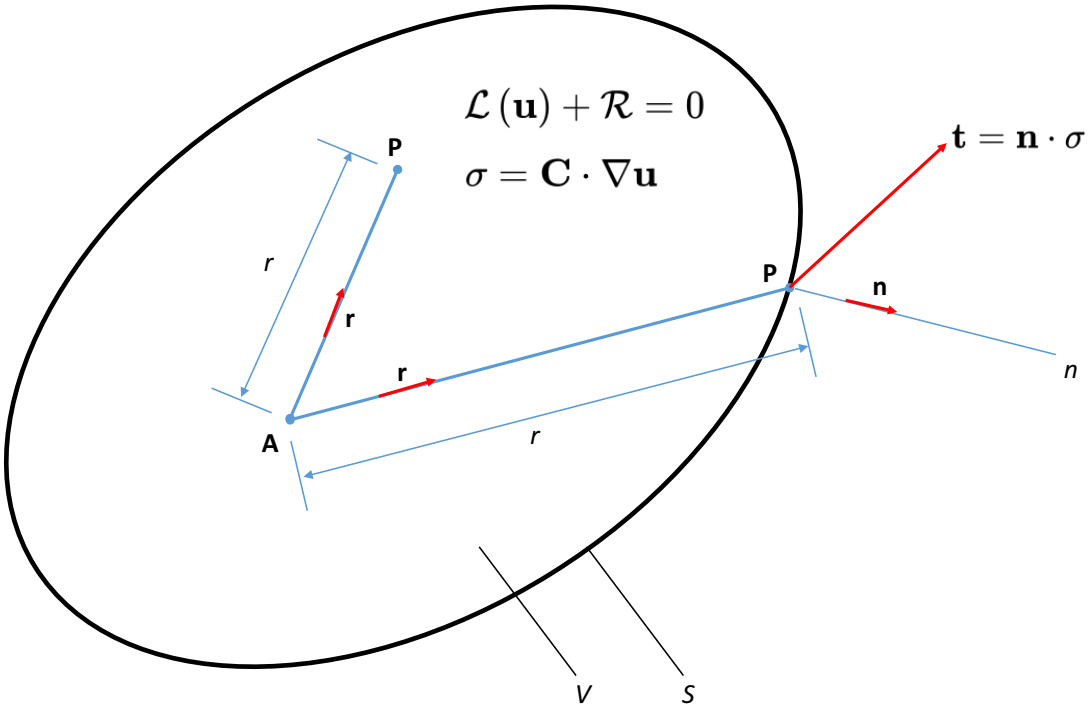


Figure 2.1: Definitions of the domain

In this domain, a field variable $\mathbf{u}$ is defined, which can be scalar, vector or tensor quantity. Examples are; temperature (scalar), light intensity (scalar), electric potential (scalar), displacement (vector), flow velocity (vector), etc. Along with this field variable, a flux can be associated with the gradient of the field variable. This flux can be defined directly as the gradient in defined coordinate system, as in electric potential,

$$\mathbf{q} = \varepsilon \nabla u \tag{2.1}$$

or its negative, as in heat transfer,

$$\mathbf{q} = -k \nabla u \tag{2.2}$$

or a more complicated form can be given as in static elasticity problems:

$$\sigma = \mathbf{C} \cdot \nabla \mathbf{u} \tag{2.3}$$

In these equations the operator ∇ is the gradient operator defined as:

$$\nabla = \partial_i = \frac{\partial}{\partial x_i} \tag{2.4}$$

where x_i represents the directions of the chosen coordinate system. Note that in Equations 2.1 and 2.2 the field variable is a scalar (therefore the symbol is not printed in bold), but in Equation 2.3 the field variable is a vector quantity. In these equations, ε, k and $\mathbf{C}$ are the material properties, namely the electric permissibility, the heat conduction coefficient and elastic properties. From this point on, the flux will be referred with the most general form, as in the form in Equation 2.3:

Further, the normal flux on the boundary of the domain will be defined by the directional derivative of the flux:

$$\mathbf{t} = \mathbf{n} \cdot \sigma \tag{2.5}$$

Note, that, if the field variable is scalar, the normal flux will be scalar. Similarly, if the field variable is a vector quantity, the normal flux will also be a vector quantity. In general, if the field variable is a tensor quantity of order s, the normal flux will be a tensor quantity with the same order.

Assume, now, that a harmonic or bi-harmonic linear differential operator $\mathcal{L}$ acting on the field variable $\mathbf{u}$ is given as:

$$\mathcal{L}(\mathbf{u}) + \mathcal{R} = 0 \tag{2.6}$$

where $\mathcal{R}$ represents the internal excitation (mostly named as the body forces or the internal generation), which may or may not be a function of the field variable. This equation, which is mostly named as the *governing equation* (GE) is defined over the solution region. In particular, for Laplace equations, the linear operator is simply

$$\mathcal{L} = k\nabla^2 \tag{2.7}$$

and for Helmholtz equations,

$$\mathcal{L} = k\nabla^2 + C \tag{2.8}$$

and finally for Navier Equations:

$$\mathcal{L} = \mu\nabla^2 + (\lambda + \mu)\,\nabla\,(\nabla\cdot) \tag{2.9}$$

where ∇^2 is the so-called Laplace operator defined as

$$\nabla^2 = \nabla \cdot \nabla \tag{2.10}$$

and $\nabla\,(\nabla\cdot)$ represents the gradient of the divergence operating on the field variable. In above equations k, C, μ and λ represents material properties associated with the problem.

Assume a fixed point (FP) within the solution domain, $\mathbf{A}$. In literature, this point is also named as the source point. This point represents the point at which the field variable is to be evaluated. Assume further another point $\mathbf{P}$ in the domain (or on the boundary) which will be named as the varied point (VP), or in some sources, the integration point. This point will be used to evaluate the field variable at the point $\mathbf{A}$ as an integrator over the solution domain.

The Euclidean distance between the points $\mathbf{A}$ and $\mathbf{P}$ will be denoted by r and the directional unit vector from the point $\mathbf{A}$ to the point $\mathbf{P}$ will be denoted by $\mathbf{r}$

There are mainly two methods to formulate the boundary element method for a given linear differential equation: The direct method and the indirect method. The indirect method involves the definition of pseudo variables. The direct method uses directly the physical quantities in an engineering system. Mainly, the direct method *assumes* an auxiliary system which encapsulates the actual system (Figure 2.2). The material

properties and the governing equations are the same in the auxiliary system, but the field variables are different, which is denoted by $\mathbf{u}^*$ and $\mathbf{t}^*$ in this text. Note, further, that the boundary of the actual system is a common boundary with the auxiliary system with having the same normal direction at each boundary point.

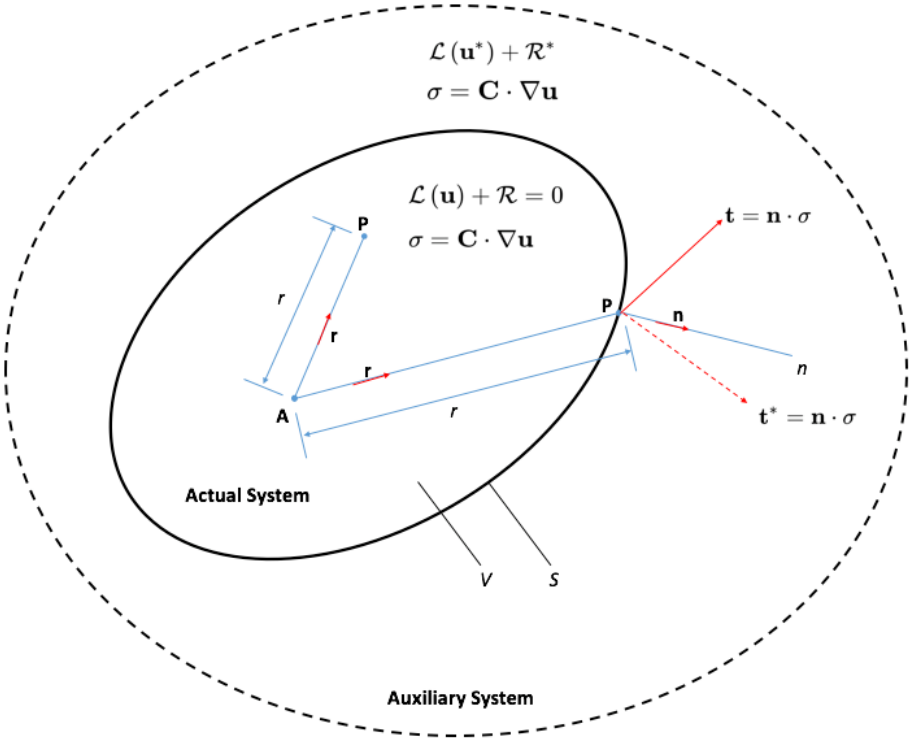


Figure 2.2: Definitions concerning the actual and auxiliary systems

At this point, for simplicity, but without losing generality, it is assumed that the linear differential equation is harmonic with the form

$$\mathcal{L} = \mathbf{C} \cdot \nabla^2 + \mathbf{S} \tag{2.11}$$

which contains spatial derivatives only. The below discussion can easily be extended to time dependent and non-harmonic or higher order linear operators.

The direct formulation starts with the Betti's Reciprocal theorem between the actual

and the auxiliary systems:

$$\int_V \sigma \cdot (\nabla \cdot \mathbf{u}^*)\, \mathrm{d}V = \int_V \sigma^* \cdot (\nabla \cdot \mathbf{u})\, \mathrm{d}V \tag{2.12}$$

Note that the chain rule requires:

$$\nabla \cdot (\sigma \cdot \mathbf{u}^*) = \sigma \cdot (\nabla \cdot \mathbf{u}^*) + \mathbf{u}^* \cdot (\nabla \cdot \sigma) \tag{2.13}$$

which, when inserted into Equation 2.12 leads to

$$\int_V \nabla \cdot (\sigma \cdot \mathbf{u}^*)\, \mathrm{d}V - \int_V \mathbf{u}^* \cdot (\nabla \cdot \sigma)\, \mathrm{d}V = \int_V \nabla \cdot (\sigma^* \cdot \mathbf{u})\, \mathrm{d}V - \int_V \mathbf{u} \cdot (\nabla \cdot \sigma^*)\, \mathrm{d}V \tag{2.14}$$

Equation 2.3 and 2.11 requires:

$$\nabla \cdot \sigma = \mathcal{L}(\mathbf{u}) - \mathbf{S} \cdot \mathbf{u} \tag{2.15}$$

and further note that, from Equation 2.6 we have

$$\mathcal{L}(\mathbf{u}) = -\mathcal{R} \tag{2.16}$$

Recalling that, the same equations are valid for σ^* and $\mathbf{u}^*$, Equation 2.14 takes the form:

$$\int_V \nabla \cdot (\sigma \cdot \mathbf{u}^*)\, \mathrm{d}V - \int_V \mathbf{u}^* \cdot \mathcal{R} \mathrm{d}V = \int_V \nabla \cdot (\sigma^* \cdot \mathbf{u})\, \mathrm{d}V - \int_V \mathbf{u} \cdot \mathcal{R}^* \mathrm{d}V \tag{2.17}$$

At this point, the Gauss Integral Theorem can be applied to the first terms on both sides leading to

$$\int_S \mathbf{n} \cdot (\sigma \cdot \mathbf{u}^*)\, \mathrm{d}A - \int_V \mathbf{u}^* \cdot \mathcal{R} \mathrm{d}V = \int_S \mathbf{n} \cdot (\sigma^* \cdot \mathbf{u})\, \mathrm{d}S - \int_V \mathbf{u} \cdot \mathcal{R}^* \mathrm{d}V \tag{2.18}$$

but in view of Equation 2.5 and assuming that the body force terms (or the internal generation) is zero (since the problem is assumed to be homogeneous), this equation turns out to the so-called Boundary Element Equation (BEE):

$$\int_V \mathbf{u} \cdot \mathcal{R}^* \mathrm{d}V + \int_S \mathbf{u} \cdot \mathbf{u}^* \mathrm{d}A = \int_S \mathbf{t}^* \cdot \mathbf{t}\ \mathrm{d}A \tag{2.19}$$

At this stage, assume $\mathcal{R}^*(\mathbf{A},\mathbf{P})$ is a Dirac-Delta tensor with the property

$$\int_V \mathbf{u}(\mathbf{P})\cdot\mathcal{R}^*(\mathbf{A},\mathbf{P})\mathrm{d}V = \left\{ \begin{array}{c} \mathbf{u}(\mathbf{A}) \text{ if } A\in V \\ 0 \text{ if } A\notin V \end{array} \right\} \tag{2.20}$$

Further, assume that the solution to the equation

$$\mathcal{L}\left(\mathbf{u}^*(\mathbf{A},\mathbf{P})\right)+\mathcal{R}^*(\mathbf{A},\mathbf{P}) = \mathbf{C}\cdot\nabla^2\mathbf{u}^*(\mathbf{A},\mathbf{P})+\mathbf{S}\mathbf{u}^*(\mathbf{A},\mathbf{P})+\mathcal{R}^*(\mathbf{A},\mathbf{P}) = 0 \tag{2.21}$$

can be found in the given Auxiliary system. This solution, $\mathbf{u}^*(\mathbf{A},\mathbf{P})$ is mostly called as the First Fundamental Solution of the problem. With this solution, it becomes possible to determine the so-called the second fundamental solution through

$$\mathbf{t}^*(\mathbf{A},\mathbf{P}) = \mathbf{n}\cdot\mathbf{C}\cdot\nabla\mathbf{u}^*(\mathbf{A},\mathbf{P}) \tag{2.22}$$

Through Equations 2.20, 2.21 and 2.22, Equation 2.19 can be re-written as

$$\mathbf{C}(\mathbf{A})\cdot\mathbf{u}(\mathbf{A}) + \int_S \mathbf{u}(\mathbf{P})\cdot\mathbf{u}^*(\mathbf{A},\mathbf{P})\,\mathrm{d}A = \int_S \mathbf{t}(\mathbf{P})\cdot\mathbf{t}^*(\mathbf{A},\mathbf{P})\,\mathrm{d}A \tag{2.23}$$

where

$$\mathbf{C}(\mathbf{A}) = \left\{ \begin{array}{c} \mathbf{I} \text{ if } A\in V \\ 0 \text{ if } A\notin V \\ \mathbf{I}/2 \text{ if } A\in \text{ smooth } S \end{array} \right\} \tag{2.24}$$

with $\mathbf{I}$ being the Kronecker's Delta Tensor.

It is at this point noted that Equation 2.23 is exact, in the sense that, with pre-determined first and second fundamental solutions, if this equation can be solved, it gives the exact solution for the field variable $\mathbf{u}$ at the point $\mathbf{A}$. In many engineering problems, though, the problem, the geometry and/or the boundary conditions are such complex that, obtaining an exact solution is either not possible or not simple. Thus, a numerical solution is attempted.

2.2 Numerical Solution

For the numerical solution, we first assume that Equation 2.23 is written at the fixed point $\mathbf{A}$ on the boundary. Thus, $\mathbf{C}(\mathbf{A}) = \mathbf{I}/2$. The boundary should be discretized to evaluate the area integrals defined on this boundary with Equation 2.23. In the context of this thesis study, for simplicity but without losing generality, we assume constant (single node) plane triangular elements are employed for boundary discretization. Note that, for Boundary Element Method, the polynomial order (constant, linear, quadratic, etc.) is independent from the element geometry (which can be planar triangular, planar quadrilateral, six-point defined quadratic triangular, etc.).

To visualize the discretization process, we assume a $h1 \times \phi 1$ quarter-cylinder solution domain as given in Figure 2.3. Assume all sides of this quarter cylinder is discretized using constant triangles (Figure 2.4). Each surface discretization along with the direction of normals at each element defined is given in Figure 2.5. It is to be noted that, being constant elements, each triangle has only one computational node; which, in the context of this study, is assumed to be located at the center of the triangle.

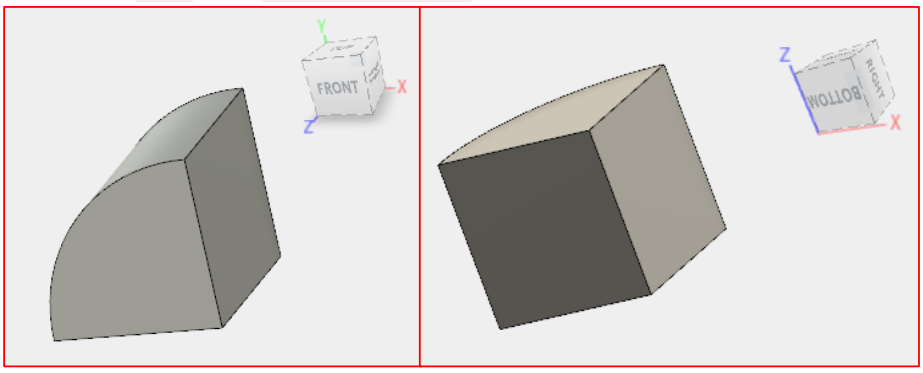


Figure 2.3: The solution domain: A quarter-cylinder with radius 1 and height 1 unit

Assume that the fixed point $\mathbf{A}$ is located at one of the nodes, i.e., the point is on the

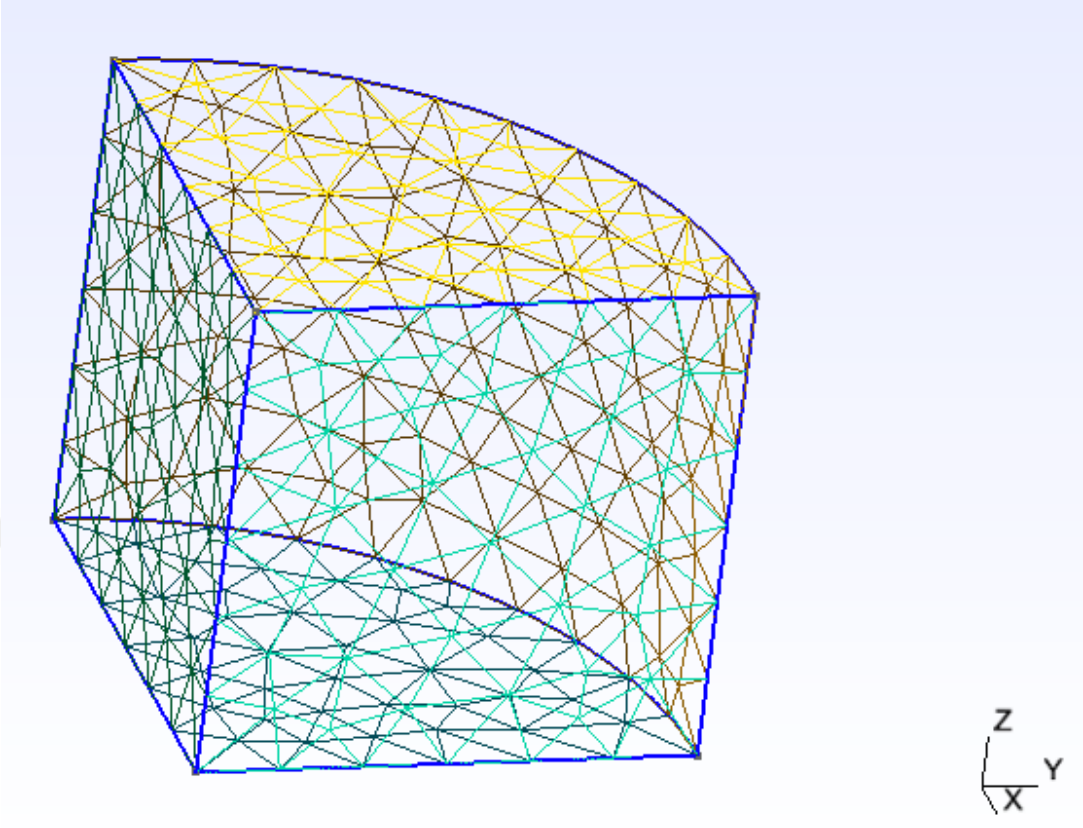


Figure 2.4: Discretized quarter cylinder

surface and located at the center of the triangle element, say element i. This point will be denoted as $\mathbf{A}_i$. The area integrals can be approximated as the sum of the area integrals obtained from each triangle on the boundary. Thus, Equation 2.23 can be written as:

$$\frac{\mathbf{I}}{2} \cdot \mathbf{u}(\mathbf{A}_i) + \sum_{j=1}^{N} \int_{S_j} \mathbf{u}(\mathbf{P}_j) \cdot \mathbf{u}^*(\mathbf{A}_i, \mathbf{P}_j) \, \mathrm{d}A = \sum_{j=1}^{N} \int_{S_j} \mathbf{t}(\mathbf{P}_i) \cdot \mathbf{t}^*(\mathbf{A}_i, \mathbf{P}_j) \, \mathrm{d}A \quad (2.25)$$

where S_j is the j^{th} triangle in the mesh and P_j is the integration point over that element. Note that the center point of this triangle is the j^{th} node, i.e., $\mathbf{A}_j$. The total number of elements used in discretization is denoted as N.

Constant elements are assumed, thus, the field variable $\mathbf{u}$ and its gradient $\mathbf{t}$ are both assumed to be constant over the element. It is for this reason, at each element, the

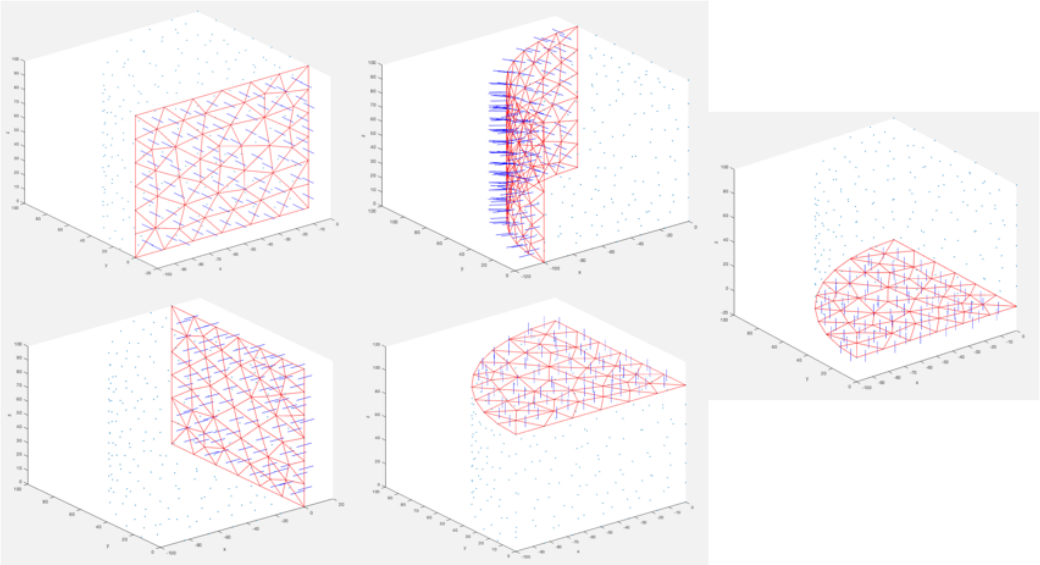


Figure 2.5: Discretized quarter cylinder - each surface along with normals

$\mathbf{u}(\mathbf{P}_i)$ and $\mathbf{t}(\mathbf{P}_i)$ can be assumed to have fixed values $\mathbf{u}(\mathbf{A}_i)$ and $\mathbf{t}(\mathbf{A}_i)$ which will be denoted as $\mathbf{u}_j$ and $\mathbf{t}_j$ respectively. Similarly, the value $\mathbf{u}(\mathbf{A}_i)$ will be denoted as $\mathbf{u}_i$. Also, the following integrals can be numerically evaluated:

$$\mathbf{G}_{ij} = \int_{S_j} \mathbf{u}^*(\mathbf{A}_i, \mathbf{P}_j)\, \mathrm{d}A \tag{2.26}$$

$$\mathbf{H}_{ij} = \int_{S_j} \mathbf{t}^*(\mathbf{A}_i, \mathbf{P}_j)\, \mathrm{d}A \tag{2.27}$$

The final form of the reduced boundary element equation written at the fixed boundary point $\mathbf{A}_i$ is given as

$$\frac{\mathbf{I}}{2} \cdot \mathbf{u}_i + \sum_{j=1}^{N} \mathbf{H}_{ij}\mathbf{u}_j = \sum_{j=1}^{N} \mathbf{G}_{ij}\mathbf{t}_j \tag{2.28}$$

Writing Equation 2.28 at each boundary node from $i = 1$ to N, a matrix equation in the form

$$\mathbf{H} \cdot \mathbf{u} = \mathbf{G} \cdot \mathbf{t} \tag{2.29}$$

is obtained. Here, $\mathbf{G}$ and $\mathbf{H}$ are the assembled two dimensional matrices of $\mathbf{G}_{ij}$ and

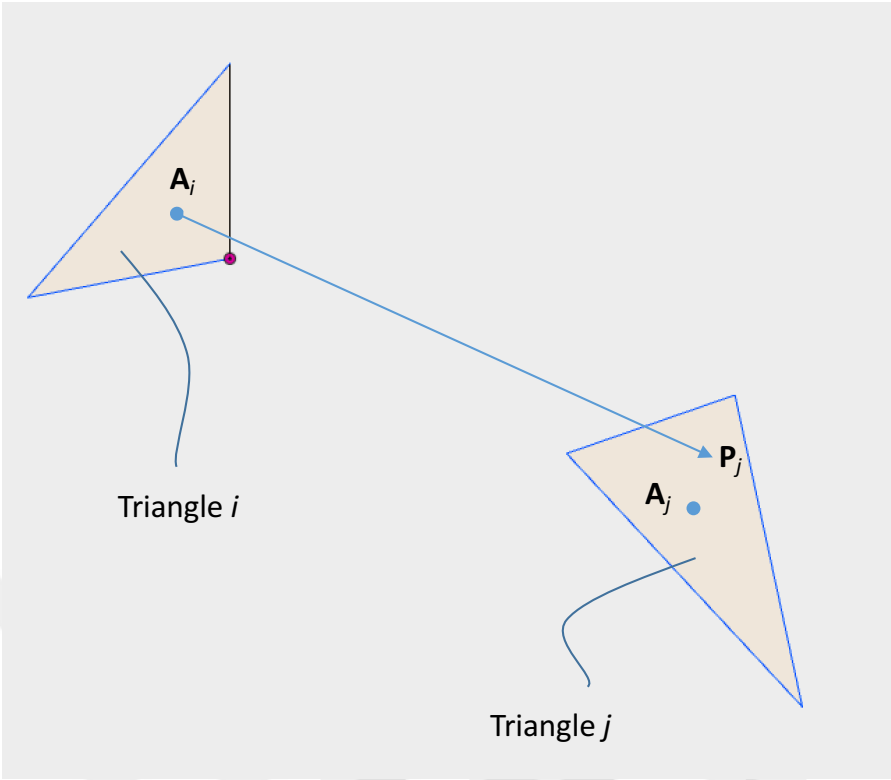


Figure 2.6: The fixed point and the integrated triangle

$\mathbf{H}_{ij}$ and similarly $\mathbf{u}$ and $\mathbf{t}$ are the assembled vectors of $\mathbf{u}_j$ and $\mathbf{t}_j$.

2.3 Imposition of the Boundary Conditions and Solution of the Problem

Assuming the field variable is a vector quantity of dimension $1 \times d$, Equation 2.29 would involve $2 \times d \times N$ unknowns. A proper unique solution would require $d \times N$ unknowns to be specified as boundary conditions. These boundary conditions may be in several forms; three most common types being:

- Dirichlet Boundary Conditions: The component of the field variable at a given point is specified, e.g., $u_j = \tilde{u}_j$
- Neumann Boundary Conditions: The component of the normal directional deriva-

tive of the field variable at a given point is specified, e.g., $t_j = \tilde{tmn}_j$

- Mixed Boundary Conditions: A function relation between the field variable and its normal directional derivative at a given point is specified, e.g., $u_j = a_{ij} \times t_i + b_j$

With the specified boundary conditions, through algebraic manipulations within the matrices (e.g. column swaps), Equation 2.29 reduces to

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.30}$$

where $\mathbf{A}$ is a $(d \cdot N) \times (d \cdot N)$ coefficient (system) matrix, $\mathbf{x}$ is a $d \cdot N$ column vector representing the unknown quantities and $\mathbf{b}$ is a $d \cdot N$ vector of known quantities. Solution of Equation 2.30 reveals the unknown boundary quantities.

It is at this stage, the Boundary Element Method completes the solution. After this, post-processing can be done for evaluating the field variable or its gradient within the solution domain using Equation 2.23 with $\mathbf{C} = \mathbf{I}$.

2.4 Advantages and Disadvantages of the BEM

The general procedure of the BEM depends on the determination of proper fundamental solutions (eg., the solutions in Equation 2.21) for the given problems. The fundamental solutions for linear and homogenous problems can be found easily and there is a very well established literature on this topic. When the problem is non-homogenous, though, the determination of the fundamental solutions is not an easy task. Further, when the problem is non-linear, the fundamental solutions may not exist or may be very hard to expressed explicitly as function forms. Therefore, in many

non-homogenous problems and almost on non-linear problems, the BEM cannot be applied directly - either is augmented with other approximate algorithms, like Dual Reciprocity or Multiple Reciporcity, etc., or combined with another method, like the FEM or FDM.

A second disadvantage of the BEM is that, the system matrices that arise in the procedure are dense matrices and they are not symmetric. Also, since the final form of the system matrix is obtained through column changes between two matrices, which is not compatible in dimensional order, the resulting system matrix is mostly ill conditioned and also is not diagonally dominant. This brings in difficulties in fast computation of the solution, and mostly makes many iterative solution methods not available.

Also, since the system matrices are dense, this brings the problem of storage. Also, evaluation of them requires more data transfer, especially for multi-core or multi-computer architectures. This was the major drawback of BEM until the emergence of larger memory storage and new computer architectures as well as compilers.

Aside from these disadvantages, the BEM has several advantages that makes the method a very potential alternative: First advantage is on the discretization of the solution domain. Unlike other methods like FEM and FVM, the BEM is a boundary-only method which requires the discretization of the boundary of the solution domain only. Noting that the integral equation is exact (not approximate as in FEM and FVM), the compatibility and connectivity of the elements are not important issues. Thus, any network of elements that span the boundary completely would suffice. This pays off well in case of new element types such as the isogeometric elements.

It is obvious that the boundary-only nature of the BEM also decreases the modeling

time, especially for those problems where the solution domain is 3D and complex. The number of unknowns would decrease drastically in such cases and more if small details are also to be meshed. It should also be noted that, the derivative quantities within the solution domain in the BEM are evaluated with no internal mesh; but in FEM and FVM, to obtain the derivative quantities, mesh should be refined in the areas of requirement - making the modeling more complex and number of unknowns increase.

Yet, placing this advantage of modeling and meshing, it should also be noted that, there is no boundary element mesher in the field that makes full use of these properties of the BEM. The method is stuck to the surface meshers that are mostly bounded by the requirements of the FEM and the BEM.

Since the reduced boundary element equation is exact, but solved numerically, the BEM is mostly considered as a *semi-analytical* method. This bring the advantage that the BEM is very accurate in the problems with which it is applied. Also, the resolution of solution within the solution domain is independent of elements; that means, once the boundary solution is obtained, the solution can be populated within the solution domain with no requirement of remeshing. This is also true for derivative quantities, which is very important for problems where the accuracy in derivative quantities is important. In such cases, mostly, the FEM and FVM uses an initial mesh to obtain the regions of interest where the variation of the derivative quantities are high, and then remeshes those regions and resolves the solution. This remeshing and resolving operations may also be repeated. In the BEM, there is no such requirement and the derivative quantities can easily obtained in every local point within the solution domain.

Also, boundary only property of the BEM makes it most valuable in problems with geometrical changes - such as the problems in plasticity, or particle flow in fluids. In the numerical methods that require domain discretization, the solution domain should be remeshed successively which often reduces the accuracy and sometimes renders the solution obsolete. In the BEM, however, this is not a critical issue, since only the boundary of the region is meshed and also the compatibility and continuity is not important. Local mesh refinements and/or relaxations can easily compensate the boundary changes.

Also, through proper fundamental solutions, the BEM can be applied easily to infinite and semi-infinite solution domains and can very accurately solve wave propagation problems in such media. This makes it very important in the fields of electromagnetics, acoustics and elastic wave propagation (eg., earthquakes).

This discussion ends with a note on the major disadvantage of boundary element method. In the last decades of 20^{th} Century, the popularity of the FEM and the FVM was so high that every commercial software focused on these methods. This made all researchers focus on the numerical procedures that ease the computation with these methods; both in terms of solution strategies and in terms of modeling. Thus, a complete framework that encompasses all advantages of the BEM is not studied on; mostly bits and pieces on research focused on different advantages, mostly in the academic field. For this reason, It is thought that the BEM requires a very well educated (mostly with doctoral degree) users to operate.

CHAPTER 3

RESEARCH QUESTION

3.1 Aim and Scope of the Thesis

The aim of this thesis is to develop a framework which can solve engineering problems with the BEM. This framework also aims to be extended and used for being a testbed for new problem solution strategies for solving problems with more accuracy and speed. The framework shall be able to be used as a part of a sophisticated application, which utilizes the BEM for solving problems.

The requirement for the development of the software arose from the fact that most of the available BEM software was unfit for the purpose for both testing new methods and solving realistic problems. The problems discussed before starting the design and implementation of the thesis can be listed as follows:

- Existing frameworks or libraries (a discussion and comparison of some of the libraries will be given in Section 4.3) are too limited in flexibility and cannot be extended easily.

- Existing frameworks or libraries require implementation of huge amount of

code for problem modeling.

- Most of the academic software end-up as single purpose applications and they are thrown away without any real world use.
- Most of the engineering software is built upon old file and storage standards and written with 3rd generation languages.
- Since BEM requires more hardware and computational resources, it is not yet optimized for single board computer usage, despite the advances in the single board computer segment (i.e. workstations, high end desktops, small servers).

As a result, aim of this thesis is to design and implement a modern BEM framework which is

- Easy to extend and modify
- Developed with modern technologies
- Easy to use, flexible and multi-purpose
- Allows users to solve problems without extensive programming
- Can work on desktop computers, but with high performance and better scalability

While the requirements seem extremely challenging from software engineering perspective, it's not impossible.

This thesis is primarily concerned with development of a BEM framework which can solve the problems with a new programming model, which is discussed in Section 4.6.

While performance of the developed framework is benchmarked in section 6.2, this performance comparison is not meant for a comparison with other software, which uses BEM or another numerical method to solve the same or similar problems. Aforementioned benchmarks are only for testing the framework in itself, and for finding its implementation's limits.

3.2 Research Methodology

This thesis' main aim is to develop a framework which can be used beyond this thesis period. This aim also creates requirements for maintainability and planned development of the framework. Because of these reasons, the development process closely resembles a real-world software development project.

Since the developed framework will test novel ideas and implementation methodologies in a not-so-well discovered application domain, project's experimental content will be high.

Software projects should be well planned for all stages of development, since both software and the development process itself is complex. However, research content adds substantial amount of uncertainty into this process, and makes it hard to plan and envision. To be able to succeed in such projects, application of some software project management strategies are beneficial. This can be summarized as follows.

- **Hardest part first:** Every software have some conventional parts, and these parts have well studied solutions, however the most time will be spent on *unknown* parts, and starting on them early is beneficial. This work applied this principle to materialize the novel parts first, and then the rest is built upon com-

pleted novel parts.

- **Agile methodologies:** As aforementioned, software research can be very unpredictable, but time is not infinite. To be able to minimize the uncertainties, working on small steps and being able to change development direction quickly is advantageous. Agile software development methods enable software teams to change directions fast and visualize the work done, but they are also extremely useful for teams of one to plan effectively and communicate their situation. As a result, this work employed agile methodologies extensively to plan and understand the work status. Detailed information about agile methodologies, such as Scrum, can be found in Schwaber's [18] or Martin's [19] books.

As a result, this thesis has combined an academic software research done for real world applications with a full fledged software development process to achieve results. The positivist and empirical approach is applied to the results obtained. The challenges of ordinary research is made visible with both software development tools and widely used, contemporary project management methodologies to make them tangible, hence easier to approach, understand and pass beyond.

CHAPTER 4

FRAMEWORK

4.1 Introduction

The Boundary Building Blocks Framework (B3 Framework or B3F), which is developed in this study, addresses the problems presented in previous chapters with a completely modular architecture and much simpler programming model which provides easy extendibility and ease of use. The B3F also utilizes the new features of C++11 and the modern processors which are launched in recent years.

To utilize aforementioned advancements and new features provided by the hardware, the B3F is developed using C++11. C++11 provides new features that enable effortless development of multi-threaded applications with high performance with its support of atomic variables and native threads [12, 13, 14, 20, 21].

The B3F uses *Eigen* [22] as its core matrix and algebra library. *Eigen* is a very big, yet very fast and easy to use library which supports dense and sparse matrices, operations on said matrices, contains its own linear solvers and matrix decomposers with various speed and accuracy properties. The library makes use of modern processor features like vectorization and new instruction sets. These vectorization and other op-

timizations are made explicitly for many CPU architectures and they are not decided by compilers' optimization processes. *Eigen* is also independent from other similar libraries in terms of both development and dependencies. This feature makes Eigen very easy to include and compile. *Eigen* compiles with the developed application and doesn't require a separate installation and linking step. *Eigen* is used by many high profile projects like ESA's Space Trajectory Analysis project, CERN's ATLAS experiment at LHC [23], Google's Tensor Flow and Ceres, Computational Geomety Algorithms Library (CGAL) and more. A complete list is available at *Eigen*'s Homepage [22], under title *Projects using Eigen*.

Developed framework does not contain any mesher or mesh manager. The B3F accepts a special, XML based, problem file which contains point cloud, point connectivity data, simulation details, boundary conditions and materials with required properties. This file acts as a single container for all problem data. The file has been designed for both easy translation to in-memory data structures and for extensiblity in mind. The library which handles this problem file is *RapidXML* [24], a very fast XML parser and writer which allows accessing XML Document Object Model (DOM) directly. Like *Eigen*, *RapidXML* is a production quality library which is used in products, like HTC mobile devices [24], used daily. *RapidXML* is also developed to be integrated to the project at the source level and compiles with the project, and does not require different installation and linking steps.

The B3F makes use of two other libraries, named *Easylogging++* [25] and *Catch* [26] which allows detailed logging and testing facilities for the B3F. *Easylogging++* is a single header, type and thread safe, macro based logging library targeting C++11. It is used for normal and verbose logging and simple performance tracking purposes. *Ea-*

sylogging++ provides very flexible logging options, allows for self-removal-during-compilation by using compiler flags, hence some log directives can be disabled to maximize performance during release compilation and has built-in high resolution timers, so that the code performance can be monitored during executions without using any other instrumentation. *Catch*, again a macro based library, allows development of unit tests of the codebase with ease. A C++ file written with *Catch*'s macros is compiled to another binary which runs the tests and gives results. Like *Easylogging++*, *Catch* has duration timers for test cases, which can be used to track performance of the code. *Catch* is actively used during development to both verify and regression-test the library. Both libraries are again drop in libraries compile with the application, in a single step.

Even though the B3F uses external libraries for various tasks, due to the nature of the libraries used, framework only depends on a C++11 compiler and STL library. This allows framework to be compiled on the target computer, with a single step, without any external dependencies. All aforementioned libraries come integrated to the B3F source tree.

4.2 Differences between Application, Library and Framework

In the world of software engineering and computer science, not all software is written from scratch, and not all software is written to fill the same role. Over time with the evolution of software development practices and software engineering itself, different types of software has emerged. These software can be put in three broad categories. Applications, libraries and frameworks. In this section small definitions of these con-

cepts will be given from a software engineering perspective.

An *application* can run by itself and perform its functions, hence it can be considered standalone. An application can use libraries and frameworks to perform its functions or to provide its other properties (parallel execution, system integration, communication, etc.).

Libraries are software packages which provide features or functions in a reusable form. Libraries are generally developed for a specific purpose (image processing, linear algebra, operations on a file or data type, etc.), and cannot run by or perform these functions by themselves, and shall be incorporated into a larger software entity like an application or a framework. While they cannot perform the function by themselves, they are the fundamental building blocks of software and code reuse.

A *framework*, on the other hand, can be considered as a partially finished software. It provides functions and subsystems, which eases and accelerates the development of types of applications that the framework targets. This means a framework is not a universal entity, but developed for a specific application domain or application type.

Applications, libraries and frameworks are utilized in different ways. Applications are typically standalone, and they are used by *running* them. When an application is run, its functions are used to obtain end-results, however the user has no control over the order of functions called, or other internal parts. Libraries are complete opposite of this use case, and they needs to be integrated into the application being developed by the application developer itself. Also, the data logistics, calling order and all other details should be handled by the developer.

On the other hand, frameworks needs to be instantiated, and customized according

to the needs of the application being developed. Their unique construction results in emergence of three features which are shared among all frameworks.

- **Inversion of Control (IoC):** Since frameworks are unfinished applications, after a certain point in their lifecycle, the control of the software flow is transferred to the framework. In other words, unlike libraries, frameworks are *run* or *launched* and all control is transferred to them for execution of the application which they're used to build.
- **Extensibility:** Frameworks are designed to be extended by the developer of the application during development. This can mean implementation of additional code, or extension of the framework by sub-classing its classes. This extension can result in addition of new functions or completion of parts which are intentionally left unfinished by the framework developer. Frameworks have designed in features for extension to make this process easier.
- **Non-modifiable core:** Frameworks' core parts, which most of the time implements the foundational data structures and application flow control sections are not meant to be modified and excluded from the extension mechanisms. This structure forms a two layer structure, which consists of an extensible outer layer and non-modifiable (or not meant to be modified) inner core.

The B3F is a framework and have all three features mentioned above. It is an unfinished BEM application, which requires transfer of control for problem solution, is designed to be extended for development of new problem solution methods and has an unmodifiable core which provides the basic data structures and functions.

For more information about frameworks, more detail can be found in Buchman's

book [27].

4.3 Comparison with Similar Libraries and Previous Work

Developed framework is neither the first, nor the only software which employs the BEM to solve various problems. Many pieces of software are written before the development of the B3F, and some of these are referenced in the introduction section. This section will briefly summarize the previous work done on software which employs BEM to solve the problems they attack and discuss the differences with the B3F in more detail.

Formerly developed BEM software falls into several categories. These are purpose build software, general purpose applications and libraries. As discussed in earlier sections, because of the resource intensive nature of the BEM, early work's main focus is to overcome the limitations of single computers when attacking the problems at hand. This can be either parallelization in various forms, or methods and formulations to reduce memory usage of the method to be able to solve problems with bigger and more complex geometries. Later work, with the increased capacities of computers and advances in software technology, work more on the accuracy and flexibility of the developed software.

Parallelization of BEM in various forms is a widely studied subject. Semeraro and Gray [28] has parallelized BEM with PVM [29, 30] using PBLAS [31], BLACS [32] and ScaLAPACK [33, 34]. They have solved problem of modeling a capacitive sensor with 6600 nodes, and in order to overcome memory limitations of a single computer, they have parallelized their code. In their approach, communication between nodes

are handled by the BLACS itself, hence the developers requested transferring blocks of matrices, instead of memory addresses. This work is also one of the earliest examples of parallel integration, called *Matrix Assembly* in the literature. They have opted to use *Block Cyclic* distribution of the matrix in the assembly process, which is simply dividing matrices to blocks and sending blocks one by one to computers which are sharing the load of the assembly work. They have opted to use this method, because it is both simple and the underlying linear libraries they use this method by default. The solution of linear systems are done sequentially, in a single node. Another idea implemented in their study is not to store the H matrix in the memory in order to reduce memory consumption with the trade-off of re-computation of the H matrix, however the increased computation is not seen as a disadvantage when compared to reduced memory usage.

Baltz and Ingber [35] also developed an application to analyze heat conduction in heterogeneous media using parallelized BEM. Their approach also used distributed *Matrix Assembly*. Unlike Semeraro and Gray, this study has access to more advanced tools. They were able to use a parallel LU solver, and an Intel Paragon, a much more powerful, massively parallel supercomputer. Beltz and Ingber also worked on different aspects of the parallelization of the BEM. First of all, they developed a method to distribute the matrices during assembly process so that the work done by every node is equal. This is necessary since evaluation of singular integrals require more processing power to complete. Similarly they have worked on multi-domain problems and another method to partition work on multi-domain matrices, so that the work is again equal across nodes. This is required since multi-domain problems produce more sparse systems than single domain systems.

When compared, the B3F has similar properties. The B3F is not currently capable of running on multiple systems, which can be counted as a disadvantage, but it can complete assembly of the matrices in parallel manner using all CPU cores on a single computer. Similarly, the consumer structure in the assembly process, which means every thread gets the next block for integration from the big matrices, results in equal work done by threads at the end. While a thread is busy with a singular or an integral which needs much work, other threads can consume and complete simpler integrals, hence no thread waits idle for work.

Another method of parallelizing BEM is to artificially partition the problem into multiple domains, which is done by Kamiya, et.al. [36, 37]. While there are BEM problems which should be modeled with multi domains because of multiple materials, a single domain problem can also be partitioned into multiple domains to accelerate its solution process. This method provides a simpler parallelization model, since the whole BEM code is essentially left untouched, but the domains are solved in parallel. However, this method is not without disadvantages. Solving multiple domains in parallel requires evaluation of cross-domain flux and potentials, and requires iteration of the solution until the flux and flow values converge, which requires 10 to 20 iterations which can be seen in referenced work. If the problem being solved is time-based (i.e. requires multiple time-steps), this approach results in multiple iterations per time-step. However, since the boundary condition renewal and re-calculation does not invalidate matrix assemblies, the time impact is acceptable in this situation.

Another similar approach is taken by Ingber, et al. [38], which proposes a tool to automatically decompose complex geometries into domains which can be both evaluated concurrently and with greater accuracy. Their novel idea is to take models in the

form of Exodus II files, which are FEM database files, and decompose it to Exodus II Parallel files. After dividing these FEM files, these parallel files are transformed to BEM compatible models by an analysis procedure. Their application can divide the problems into great number of domains. They have evaluated a problem with 1024 domains in their work. During solution and convergence calculations, more than $300,000$ nodes were present, and they successfully evaluated the problem with the BEM.

The B3F is aware of the domain concept of the BEM both in Problem file and solution functions layer. However, it currently does not support multi-domain solutions, since required inter-domain calculation routines are not implemented. However, by design it is perfectly capable of solving domains in parallel and performing required calculations without modifying the core concepts of the framework.

Another approach is to solve some of the problems with more optimized formulations, namely Fast Multipole Method (FMM) and Multi Level Fast Multipole Method (MLFMM). These methods are generally applied in the BEM applications on electromagnetic scattering problems. This method brings a new formulation, which accelerates the multiplication of the matrix-vector multiplication of the Conjugate Gradient (CG) method, when it is used to solve the system iteratively. Song, et al. has implemented a naive FMM and valiated its results to known, more resource intensive solution methods [39]. Idesbald van den Bosch has worked on the MLFMA variant and further optimized the formulation and implementation make the solution even faster [3]. Idesbald's method especially focuses on the optimization of the algorithm and re-ordering its operations to reduce memory usage. His work is also materialized as a general purpose application called Puma-EM [3]. However, while FMM

and MLFMM can accelerate the calculation, they are not parallelized yet according to references given.

The B3F uses solvers supplied by the *Eigen* library, and does not implement its custom solvers currently. *Eigen* also includes a family of iterativ solvers, including a *Conjugate Gradient* solvers, and have preconditioners. Currently, utilized non-iterative solvers of *Eigen* provides fast and accurate solutions. If required, the iterative solvers may be used as well, however currently implementation of custom solvers is a low probability because of the well performing and broadly available solvers and the effort required to implement a high quality solver.

With the improvement of software technology and the research on BEM, more general purpose BEM packages started to emerge. These are modeled as software libraries. During the survey, two of them encountered and analyzed in depth. These are BEM++ and BETL.

BEM++ [5] is a library written with C++ which can solve Laplace, Helmholtz and Maxwell equations. It is set apart with its object oriented structure and support for Python extensions developed in tandem with the library. BEM++ is designed to be scriptable with Python, hence its design is made with this requirement in mind. High performance C++ libraries are generally built with templating, which is instantiated during compilation, to remove runtime polymorphism that creates some runtime overhead. However, templating the code reduces the flexibility of the code, requiring recompilation to change some parameters. This doesn't fit well to a library which also provides scripting support. Reasons for this is detailed in the relevant research paper, and can be inquired for further details if desired. BEM++ also provides modern integrations like solvers from Trilinos project [40, 41] and AHMED library [42], which

implements hierarchical matrices for elliptic curve equations [43]. While BEM++ doesn't contain any solvers, it can handle matrix assembly in parallel manner with the built-in assembler.

BEM++ does not only integrate libraries from the computation landscape, also contains its own *Fast Integration Boundary Element Routines* (FIBER) library. This library provides matrix assembly and other basic steps required in solution of all BEM problems. Another feature of BEM++ is own unique programming model. While BEM++ can import GMSH [44] mesh files and work on them, BEM++ can mesh objects declared as functions via Dune-FoamGrid [45], hence can work on problems without generating mesh files by hand. To ease implementation of the said formulations, BEM++ provides functions which provides *parts* of the formulas which can be modeled. Users can use this function parts to model their problems without generating relevant mesh files. Boundary conditions and other constrains on the problems are defined in a similar manner. As aforementioned, BEM++ does not have any integrated solvers, so Trilinos or another supported solver library (which is detailed in the relevant research paper) is required.

Quadratures are also handled inside BEM++. BEM++ generates quadrature sets via internal functions according to given *accuracy options* defined. BEM++ have a simple distinction between singular and non-singular integrals. For singular integrals, unless overridden, a quadrature order of $nonsingular\ order + 5$ is used.

Programming and customization of the BEM++ is done via *traits* system, which involves implementation of classes and providing this classes as overriding classes to objects which manage the properties of a given part. This means, to change the behavior of quadrature selection and generation, a new class with the desired parameters

is implemented and given as the *traits* for the quadrature parameters, so the traits can be overridden. While traits provide a very low level access and a well defined interface to the library, it is not practical and straightforward to implement. Any small parameter change requires a complete class implementation and overriding procedure which is time consuming.

As a result, BEM++ is a nearly complete library built around the idea of modeling and solving the problem inside the code completely. With the integration of the Python bindings and basic visualization tools, the library also becomes usable in scripting environments or interactively. With the integration of AHMED and Trilinos, the BEM++ can potentially solve big problems, but it lacks the easy customization and, providing complete problems for solution with a fixed procedure is not simple.

When compared to B3F, B3F's *Eigen* integration provides flexible matrix classes and efficient solvers. Some of these solvers are also multi-threaded. Similarly B3F can do matrix assembly in parallel. B3F currently doesn't support any other linear algebra library, however *Eigen* is compatible with other well-respected libraries in terms of data storage. Since the B3F's data storage is not hidden behind libraries, these data structures can be directly implemented in the *Flow* level (Concepts of the B3F is discussed in future sections of this chapter).

Quadrature handling is completely different in the B3F. B3F statically stores the rules for $21\ point$ Gauss quadrature, however with its adaptive element subdivision integrator, integrations are continued until a given percent error target. This method removes the need for overriding and tuning the matrix assembly tolerances, since the algorithm is constant accuracy, only the desired accuracy need to be provided to the algorithm.

When compared from programming model perspective, the B3F is complete opposite of the BEM++. The B3F has its own file format and data structures, which is designed for portable problem data transfer between platforms and systems. The scenario designed for the B3F is to implement a problem solution process (which is called *Flow*) once, and solve many problems with this *Flow* by providing different problem files. To be able to tune *Flow*s, a much simpler customization method has been devised in the B3F. Values are directly given as parameters to the *Flow*s, hence and implemented flow can be tuned with a few lines of code, if desired, before starting problem solution. This scenario, once the flow is implemented, reduces the solution of the problems into a few lines of code.

Another library developed in recent years is the *A Generic Boundary Element Template Library* (BETL) [6]. BETL is also a C++ library, developed primarily with templating and with generic programming paradigms. From programming perspective, BETL is representing the other edge of the spectrum.

BETL is designed around extensibility and efficiency, and modeled after the process of BEM. In other words, BETL provides a fixed skeleton that outlines the BEM solution process, and provides modules written with generic programming paradigms, and allows any user to change or extend these processing units. Also designed as a header only software, so it can be easily integrated into other C++ code, presumably more sophisticated BEM software and act as a back-end library for its operations.

BETL can solve Helmholtz and Laplace equations, supports triangular and quadrilateral polynomial parametric surface meshing, can use GMSH and VTK [46] file formats for importing geometries. Like BEM++, BETL can integrate with the AHMED library. On the other hand BETL provides its own solvers, but doesn't support any

type of parallelization, except AHMED's internal multi-threaded structure (AHMED can work in concurrent mode when compiled with MPI). For dense linear algebra support, BETL uses Boost/uBlas [47] as the core linear algebra library.

BETL models its solution pipeline in two big *model*s. These are Discretization Model and BEM Model. While Discretization Model stores the basic details about the geometry (mesh, degree of freedom, elements), BEM Model stores details about the BEM parts (fundamental solution, kernel, quadrature, integrator). Another structure called BEM Operator oversees these two models. This two spaced approach separates the problem geometry from the BEM process, and provides a natural separation between the problem and solver.

Like the BEM++, BETL also generates its quadrature rules in-code, during the problem solution, however BETL has a location aware integrator. The integrator defines four levels of singularity: regular, vertex adjacent, edge adjacent, coincident, and uses different rule sets for every case. In some limited cases, semi-analytic integration is supported for exact inner integrations. However, even in this case outer integrations are evaluated with Gauss quadratures.

BETL works by parametric instantiation of its data structures. This means, BETL code is written for a specific problem or problem type, but with preselected parameters on all aspects of the library, like the problem type, quadrature details, fundamental solution, solver selection, integrations, and other details are fixed when the code is compiled for a problem type. This aspect, while provides higher performance, limits the runtime flexibility of the code.

When compared with the B3F, BETL is more similar to the B3F than other libraries.

with its separation of geometry from problem solvers, and by providing a more complete library with internal solvers and problem file importers, it provides a complete package which can solve a problem from start to finish. BETL is also capable of solving large problems by utilizing AHMED library. It also supports 6 types mesh elements, three triangular and three quadrilateral. BETL also has a more fine grained singular integration handling with four levels of singularity control and semi-analytic integrations in some limited cases. However, further efforts to increase performance, namely design built upon extensive generic programming, and modeling a skeleton after the mathematical structure of BEM limits the flexibility of the BETL in real world applications. Even simple steps like fundamental solution (i.e. problem type) and mesh geometry selection is done over template parameters, which requires code recompilation for every small change in the problem solution process. Since templates are instantiated in during code compilation, these parameters cannot be bound to options or configuration files. In fact, template based generic programming is not considered harmful, or unnecessary. Instead, its extensive usage is considered limiting. *Eigen* is also a template library however, only required parts are template based, and its flexibility is not hampered by its template based parts.

The B3F is on the other hand is built upon runtime instantiation, but designed with very light layers. This allows the B3F to be compiled once and its all supported features can be used without any re-compilation. In fact, the B3F is also a header-only framework and lacks orchestration logic which will instantiate the framework and start the evaluation of problems, but a sufficiently advanced application can use its all features, and alter its all problem solution parameters without code modification and re-compilation. B3F also provides a more adaptive and advanced quadrature

evaluation model when compared to the BETL. While BETL's four level singularity control is novel and clever, it is not fine grained and constant accuracy like the B3F's adaptive element subdivision integrator. Also, again with *Flow*s, the B3F provides a more flexible problem solution unit. When combined with the problem file which contains every aspect of the problem, the B3F can act as a batch solver for problems it supports.

When compared to all libraries, the B3F's compilation dependencies is much smaller. The B3F can be compiled with all of its features using a standard C++11 development environment. Also function overriding and extension in the B3F is much easier with layered functions designed in the B3F. This makes the B3F more user-friendly for problem solving and integration with other software. In the B3F, parallelization is regarded as a first-class citizen, so the design and structures inside the B3F is designed to accommodate and encourage concurrent operation. Also, B3F is the only library which is aware of multiple domains inside a BEM problem. Neither BEM++, nor BETL is aware of the BEM domain concept. Last, but not the least, the B3F is the only framework for BEM problem solution which handles all the data logistics and packages solutions into well defined units called *Flow*s as the author is aware of. All other software are either designed as applications, or libraries. All of these facilities and functionality is detailed throughout this chapter.

While B3F seems superior from many perspectives, it is not free of disadvantages. First of all, the B3F is not utilizing advanced BEM solution techniques like MLFMM or AHMED's hierarchical matrices or other features. Unlike other libraries, B3F has no well defined interfaces for Trilinos or other highly regarded solvers in the BEM landscape. This potentially limits the problem sizes which B3F can attack for now,

however the design of the B3F will not hamper integration of interfaces for these advanced libraries and features.

4.4 General Structure

The most important components of the B3F are six namespaces and two classes. These components enable the B3F to store the complete problem in-memory and solve it.

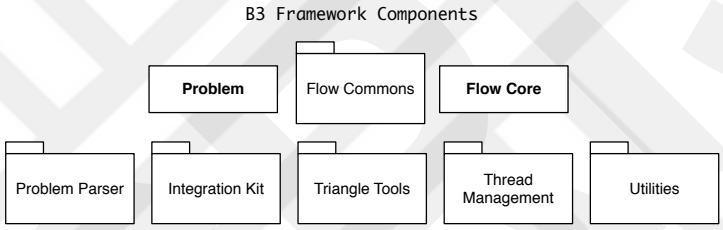


Figure 4.1: Framework's main components.

The *Problem* class stores geometry, domains, mesh, boundary conditions, materials, material properties, and other details required to solve a problem with relevant meta data inside a tightly integrated package, but without sacrificing the ease of access.

Flow Core is an abstract class (i.e. a class with both virtual and concrete functions, but declared virtual, hence cannot be instantiated without implementing the virtual functions and making the class completely concrete), which contains all the tools which enables to easily start to work on a problem and is extended while developing a solution method for a BEM problem type. Only `evaluate` function and class destructor needs to be implemented. Classes derived from this abstract class are called *Flow*s.

Flow Core makes use of the base utilities like problem file parsing, *Callback Subsystem* functions, *Voluntary Thread Management* subsystem and others, and acts as a execution engine. In this model, programmer or user implements the *Flow* by implementing the `evaluate ()` function and the destructor only. While *Flow* encapsulates the execution and provides helper functions, it does not interfere or oversee the execution process in any way. As a result, the implementer is free to implement the *Flow* in a way she sees fit.

The *Flow Commons* namespace contains functions which can work on domains of a problem or apply steps common accross many types of BEM problems. Most of the functions in this namespace does not have the core computational functions. Instead they require a pointer to a function with a specific function signature. The functions in this namespace are called *Toolbox Function*s. These functions are designed to be as efficient as possible and can use more than one CPU core, if the underlying computation function is embarrassingly parallel, and can be executed on many mesh elements concurrently without inter-dependencies.

The function signatures in *Toolbox Functions* only require computation related information and kept as simple as possible to ease external function development.

The *Integration Kit* and *Triangle Tools* namespaces contains core computational functions which are required by the functions in the *Flow Commons* namespace. The functions in this namespace are designed to work on a single or a couple of mesh elements and called *Low Level Function*s. Likewise the *Flow Commons* namespace, functions in these namespaces are designed and specially optimized for best performance and can use more than one CPU core if the computation can be parallelized.

The *Problem Parser* namespace contains the problem parser based on *RapidXML* and related exceptions. The parser and exceptions are directly used inside the *Flow* class to provide a quick way to import problems into the memory to solve.

The *Utilities* contains some functions to print memory structures to the screen and log files to make logs more understandable for both users and developers alike.

The *Thread Management* namespace contains the components and functions required to implement *Voluntary Thread Management* (*VTM*) system implemented throughout the B3F. Details of *VTM* is discussed in depth in Section 4.9.

Developed framework's layered structure provides flexibility and expressiveness whether user chooses to use framework provided or externally developed functions. The B3F is designed around usability and extensibility without compromising the B3F's performance.

The B3F's structure can be seen from the communication perspective in Figure 4.2. This perspective shows how modules in the B3F communicates during a problem solution process.

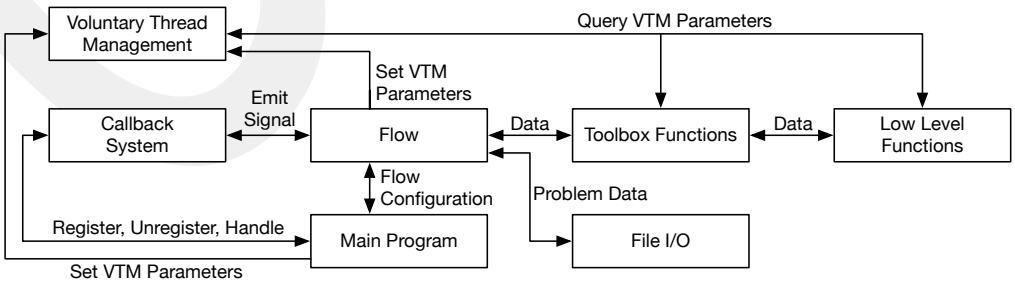


Figure 4.2: Inter-component communication in the B3F.

4.5 Extensibility and Flexibility

To enable the desired flexibility and extensibility model, namespaces are used instead of classes during the implementation of the B3F. Namespaces allows direct access to the functions, hence removes the programming and instantiation overhead required to use a single function. Second important decision was about the division of functions which provide problem solution sptes into two parts. Since BEM problem solution steps require application of small computations over large number of mesh elements, the part which handles the core computation (*Low Level Function*) and the part which applies this computation to mesh elements (*Toolbox Function*) are decoupled. This approach has resulted in many benefits during development and utilization of the B3F.

The performance penalty, which may incur because of large number of function calls resulting from this division, have been eliminated by using inline functions, which removes the function call overhead. Since the functions which do the actual computation (i.e. *Low Level Function*) and the function which iterates this computation over mesh elements (i.e. *Toolbox Function*) are decoupled, in most cases *Toolbox Function* part can be parallelized without modifying the *Low Level Function*. On the other hand, the *Low Level Function* can be optimized without modifying the *Toolbox Function*. This allows a user to experiment with new approaches and methods either in *Toolbox Functions* or *Low Level Functions* without focusing on unnecessary parts.

This capability has been implemented by removing the *Low Level Function* from the*Toolbox Function*, and adding a function pointer parameter to the *Toolbox Function*. This function pointer has a specific function signature and this function signature only carries information required for computation, hence the signature is as simple as

it can be. By implementing a function with the same function signature, and providing this new function's pointer during computation, user's own *Low Level Function* can be seamlessly integrated into the B3F. Similarly a new *Toolbox Function* implemented by user can call any *Low Level Function* already implemented allowing for different application strategies without implementation of a complex *Low Level Function* from ground up. Core framework code is not modified in either case, but easily extended.

Another flexibility feature implemented in the B3F is called *Signals*. *Signals* allow a problem solution process to be interrupted at any point and handed over to a user implemented function which can access the *Problem* object. When the function returns, the solution process continues. This method allows for midway verification, checkpointing, or conditional interventions to the problem and can be used in any way, which benefits the user during problem solution. As aforementioned, since the *Flow* doesn't oversee the execution of the `evaluate` function, *signals* are also not controlled by *Flow*. Hence, the vigilance of the implementer is required to ensure that the signal handling functions work as intended.

4.6 Programming Model

The B3F is designed to provide complete functions which can be used as computation steps for BEM problems. These functions are executed in a serial manner to create a problem solution recipe. These recipes are called *Flow*s in the B3F terminology. *Flows* are essential classes that includes all the utilities needed to solve a problem with minimal effort. These *Flow* classes are derived from specially designed abstract

base class, named *Flow Core*. Aforementioned abstract class contains utilities for problem file parsing, logging, signals, and provides the virtual functions which needs to be implemented to make the class concrete.

As mentioned in Section 4.4, *Flow* only contains the utilities and problem solution method and provides it as a single, easy to use class. The `evaluate` function is essentially free form. User or developer is free to use any function inside `evaluate` function in any way suitable. Beyond providing the necessary parts to ease and encapsulate the problem solution method, *Flow* does not oversee or manage the execution of `evaluate` function in any way. In other words *Flow* is a simple execution engine, not a complex manager, which controls every part of the execution.

When a *Flow* is implemented and is working, it becomes a well-defined box which can be used over and over with different problems of the same type. *Flows* are designed this way to promote implementation quality and reusability. Other advantages of *Flows* that, they can solve a complete problem in three lines of code. A simple problem can be solved on an implemented *Flow* as shown.

```
Elastostatic_flow elastostatic_flow;
elastostatic_flow.set_problem_file_path ("problem.xml");
elastostatic_flow.evaluate ();
```

Figure 4.3: Solution of a BEM problem using a Flow class.

To provide this ease of use and high level of flexibility, there are two classes of functions which are also implemented besides *Flows*. These are *Toolbox Functions* and *Low Level Functions* as aforementioned. *Flows* use the *Toolbox Functions* with references to suitable *Low Level Functions* during problem solution process. It is trivial

to add utilities, which allows overriding of *Low Level Functions* used in problem solution without modifying the implemented *Flow*.

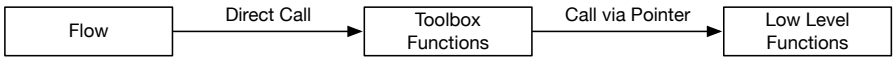


Figure 4.4: Framework function layering.

This layering of functions and classes creates a three layered structure. At the bottom *Low Level Functions* reside and handle the core computation work. The upper level functions, *Toolbox Functions*, apply the computation of low level functions to large parts of the solid geometry. At the top, *Flow* coordinates the problem solution process and directly manipulates the *Problem* objects to solve the problem. This layered approach both builds a complete, well integrated structure and allows for unfettered intervention and extension of the B3F by the user.

The process of supplying the problem information and getting results are also designed from ground up. Instead of using old file formats like STL [48], which can store only geometry data, a new file format which can store the whole problem with boundary conditions, materials and problem parameters have been designed. To complement this on-disk problem format, an equivalent in-memory problem structure is designed in tandem, so whole problem can be stored in a tightly coupled data structure, which can be conveniently accessed from a single point. As a result, the *Problem* object is the only data structure manipulated during the lifetime of the problem solution process. Both of these implementations will be discussed in their respective subsections, further in the chapter.

4.7 Data Model

The data model of the B3F aims to encapsulate all aspects of the problem in a set of tightly coupled, interconnected data structures. This data structure set is accessible via the *Problem* class, which acts as the root of this data structure set. The data structure set is modeled after the requirements of the problem and implemented in tandem with the on-disk file format detailed in the next section.

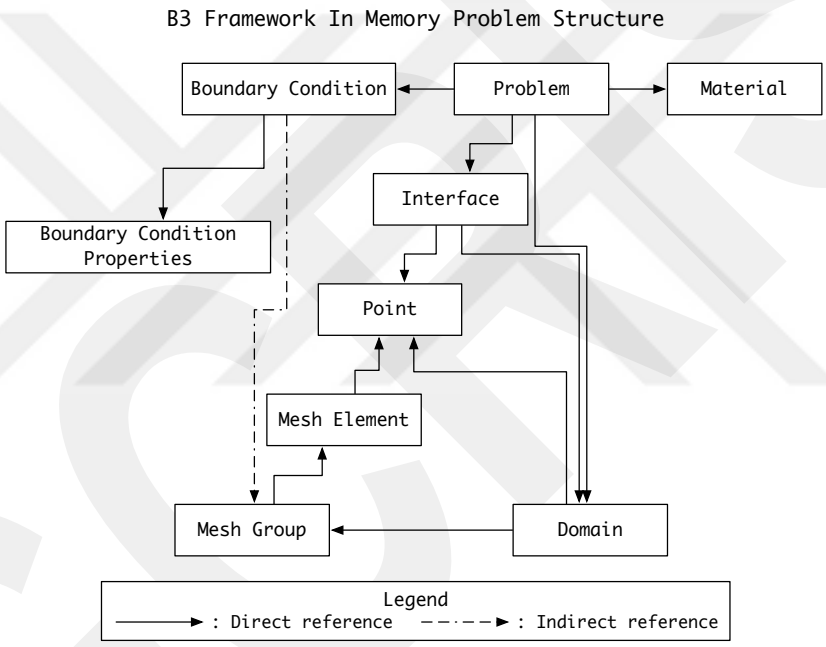


Figure 4.5: B3F in-memory data format.

As detailed in the Figure 4.5, the data structure is composed of many components, but all parts are accessible from *Problem* class, and this *Problem* class is exposed to users for interfacing with the whole data structure set. Some BEM problems contain object geometries which is composed of more than one part. The in-memory data model supports these object geometries with multiple parts. Every part of this multi-part object called a *Domain* in BEM terminology. These *Domains* store separate parts of

the solid. Every *Domain* has its own set of *Points*, *Mesh Groups* and *Material*. The *Mesh Groups* under each *Domain* are used to store different faces or surfaces of the solid part. Every *Mesh Group* contains *Mesh Elements*, which forms the respective face of the solid. Every *Mesh Element* references its *Points* from *Domain*'s domain-global *Point* set, hence when a *Mesh Element* deforms during simulation, deformation is automatically reflected to the *Domain*. This provides a very effective and efficient way of handling deformation during simulation.

If the object is composed of more than one *Domain*, these *Domains*' interactions can be defined in the *Interfaces* section of the problem data. *Interfaces* have an interface type, a list for storing *Domains* which interact in this Interface, a *Point* set and *Mesh* definition for the *Interface*. This way, more than one *Domain* in a solid can interface and interact. Data model can store infinite number of *Interfaces* and these interfaces can include any number of *Domains* for interfacing.

Data model does not only contain the object geometry, but other computational information such as *Materials* per *Domain*, and the properties of *Materials* in the *Domain*. *Material* properties are not fixed. *Problem* definition can have as many custom materials and properties as needed. Likewise, some *Domains* can be marked as non-deformable in the data model.

Data model also supports non-dimensionalization, time or frequency based computation for Fourier and Laplace, rigid body motion (rotation, translation) per *Domain*, and human readable labels for every *Material* and *Domain* for easy identification of parts of objects. The *Meshes* have freedom in vertex per element, element order and element continuity to enable modeling with great versatility.

Aside from object geometry, translation and materials; *Boundary Condition*s and related information is also stored in this data model. Model supports any number of *Boundary Conditions*. Every boundary condition can be applied to as many *Mesh Groups* as necessary and their type and 3D properties alongside their values are stored. Also, all the metadata found in the problem file also carried to the *Problem* object, hence this information can be utilized if needed to.

As a result the complete problem data, the object and every required information can be stored inside a single file on disk and as a single object in-memory. This single object in reality consists of many smaller sub-objects, but they are connected on top within a single object for easy access and modification.

4.8 Input and Output

A special file format for input and output has been developed and implemented to accompany the new data model. There are several reasons to do this. First, there is a need to be able to represent the problem on the disk for input and output. Since all software evolves over time, the problem file format should be able to adapt to change, and ideally the software should maintain backward compatibility to be able to use older files.

A complete problem contains the geometry, point space, materials, boundary conditions and other small details of the problem. Conventional data formats such as STL [48] are designed only to contain the geometry of the solid. Also, these file formats has no versioning information, hence they are not parsable reilably without some user intervention. The developed format is based on XML. XML is designed to be both

machine and human readable since its inception. Relevant metadata (file version, generating program and user data) are embedded into the problem file as mandatory fields, so file properties can be understood for optimum and effortless parsing. Also XML files' structural validity can be checked before parsing [49], hence errors in the file can be detected at an early stage. Since versioning is an integral part of the data file, parser can read the version and proceed according to given version number, hence XML based format is superior for automated processing.

On the problem data side, XML based data file contains much more information such as 3D point cloud of the mesh, connectivity of these points (hence mesh itself), domains if the solid is being worked on is multi-part, materials of these domains, any required property of the material without any limits, any number of boundary conditions required in the problem and their details, details about time or space based non-dimensionalization and time dependent solutions and simulations. As a result, the complete problem with its geometric data is stored in a well formed XML file. Moreover, this file can also be used to output the final state of the problem, hence this special file format can be used for output purposes. This feature enables the file to be used as an intermediate representation layer to couple the B3F with graphical applications, which can help both model a problem and visualize the results of the computation.

In the B3F, the input and output section is completely decoupled from the data structures, if any need to interface with other file formats or other applications arise, an implementation of importer or exporter which can understand the in memory data model will be sufficient.

The included metadata in the problem and result files also can help in cataloging the

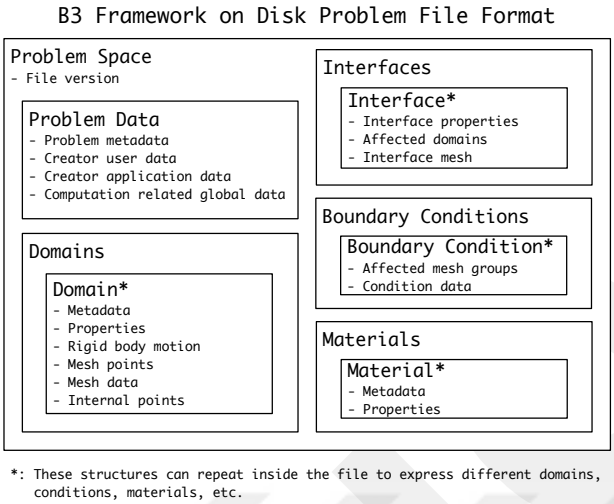


Figure 4.6: Framework on-disk problem file format.

problems and their solutions, allowing many ways to sort and search, hence saving a researcher from spending time to catalogue and organize problems and their solutions.

Currently the B3F can read the designed file format without complexity limits, however the result exporter has not been implemented yet.

4.9 Thread Management

With the multi-core processors becoming norm in computers of any size, multi-threaded programming became much more widespread, however utilizing more than one core efficiently is not straightforward in many cases. Multi-threaded algorithms frequently require synchronization and, in some cases require a specific number of cores for maximum efficiency. Similarly, development of multi-threaded algorithms is not easy. Debugging algorithms using more than one core gets exponentially harder as core count increases.

Secondly, with the inclusion of more cores in servers and clusters, the issue of resource sharing became an issue. Modern operating systems and virtualization software can limit access to the hardware resources, however not every user has access to these virtualized or managed single user environments. As a result, thread management can be beneficial in the following scenarios.

- Preventing excessive resource usage in multi user systems and servers which do not employ resource allocation management.
- Personal systems, which user wants to maintain responsiveness and ability to use system for other tasks while working on a problem.

Allowing users to run and utilize B3F on any system without rendering it unusable during problem solution process is a big advantage.

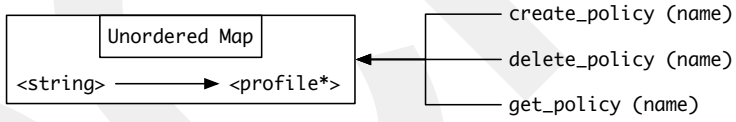


Figure 4.7: B3F VTM system overview.

To be able to make framework more usable in all of these cases, a small layer named *Voluntary Thread Management* (*VTM*) is implemented inside the B3F. *Voluntary Thread Management*, as its name implies has no power of forcing number of threads on an algorithm, hence it is not a wrapper. Instead, *VTM* maintains a list of profiles composed of free form names and an integer for allowed maximum thread count. Upon initialization, a profile called `default` is generated and its thread count is set to system's maximum thread count. Any function want to utilize this system can explicitly query *VTM* for thread count of a policy of its choice. As a result *VTM* returns the

allowed thread count for the policy. If the requested policy does not exist, `default` policy is returned as default, however this behavior can be overridden by the querying function. This allows two scenarios during the operation of the B3F.

- Framework user can limit all *VTM* utilizing functions' thread usage by directly modifying the `default` profile.
- Framework user can limit a single *VTM* utilizing function by directly creating and modifying the profile queried by the said function either for debugging or optimization (e.g. bottleneck mitigation, keeping number of threads to a specific count, etc.) purposes.

By combining these two use cases, every *VTM* utilizing function can be tuned in the B3F without much hassle. *VTM* layer has been implemented during development for debugging and scalability testing purposes at first, but left in place because of the utility it provides.

4.10 Parallelization Support

The B3F does not contain any automated parallelization support, however it is designed with parallelization in mind, and it contains a subsystem to control parallelization level of various functions which is detailed in previous section. As aforementioned in the thesis, BEM's computation process is well suited to parallelism, since the integrals evaluated during the solution of a problem is completely independent from each other.

Another reason of separating the core computation (i.e. *Low Level Function*) func-

tions from application (i.e. *Toolbox Function*) functions is to being able to exploit this parallelism in different modes. Depending on the computation being done, parallelization can be utilized in any of the three layers (*Low Level Function*, *Toolbox Function* or *Flow*).

Following examples can be given for parallelization at different layers:

- **Low Level:** If the computation function which works on the single or pair of meshes can be parallelized, paralellization at this level is beneficial.
- **Toolbox Level:** If the computation can be done on multiple mesh elements at the same time, paralellization at this level is beneficial.
- **Flow:** If the problem contains multiple parts (e.g. domains), parallelization at this level is beneficial.

While the scenarios for parallelization is not limited to the conditions given in the previous list, these scenarios are the most applicable ones.

In the B3F, this parallelization model has been applied in *Elastostatic Flow*, in two places. One is in *Toolbox Functions* level and the other one is at *Low Level Functions* level. Both parallelized operations are integration evaluation routines of the problem solution, and since the integral evaluations are embarrassingly parallel, the process of parallelization of the code was fairly easy.

In both integrations, there are two matrices, and large number of integrations are performed using these matrices (these integrations will be detailed in the Section 4.12). Since the matrices are already populated, and no integration is interdependent on each other, the problem can be modeled as a consumer style problem, where threads con-

sume an already-produced workload, which are the matrices themselves, greedily, until the work is exhausted. Since the employed integration method is not constant time and can change due to some variables (again will be detailed in the Section 4.12), assuming a static number of integrations to threads is suboptimal. Instead threads get the next integration from a global queue and work until the pool is consumed. This minimizes the idle time per thread, and threads automatically terminate when the integration pool is empty.

When there is dynamic load balancing between multiple threads, there arises the need of synchronization. This synchronization, while necessary, is the most performance sensitive part of a multi-threaded algorithm. Making synchronization lightweight and efficient will reduce the time spent for this step and will allow a thread to spend more time performing actual work. On the contrary, inefficiencies in the synchronization may reduce the performance of an algorithm in orders of magnitude.

To minimize synchronization overhead, a special variable type, called atomic, has been introduced in C++11. Atomics are special variables which allows simultaneous reads and managed updates, which are non-blocking, but with guaranteed granular operations. This means concurrent operations on the variable is applied one at a time, and with order, preventing value collisions and corruptions and guarantees retrieval of intended value by the thread using the shared atomic variable without explicitly locking. Another advantage of the atomic variables are the hardware support for atomic operations. This allows atomic operations to be converted into CPU instructions and handled in the CPU level, rather than large functions. This, in turn, makes atomic variables very lightweight and extremely fast, reducing thread synchronization overhead dramatically.

Parallel integrations in the B3F use atomic variables to synchronize and consume the correct part of the matrix. In the B3F's implementation, system's CPU thread count is automatically determined and same number of threads is created. A single shared atomic variable carrying information about which matrix block to be processed is shared among the threads. The threads read and increase the index in one atomic operation, and make the appropriate calculations on the matrix blocks they receive. Since this *read and increase* is done in a single atomic operation, other threads are not forced to wait at synchronization point, which is called *critical section*, hence the algorithm is called a *lockless* parallel algorithm.

The ease of implementing and presence of lockless algorithms, which maximizes the performance is one of the most important features of the B3F. By allowing framework users to implement these kind of functions with ease, it becomes easy to obtain maximum performance both from the B3F and the underlying computer hardware. As detailed in the results section, these lockless algorithms allows the B3F to scale very efficiently.

4.11 Callbacks

Solution of BEM problems involves many steps and being able to monitor the state of problem for various reasons (progress checking, obtanining intermediate values for various reasons, modifying problem data at various states to solve slightly different problems, etc.) may be necessary. To be able to enable these capabilities inside the B3F, a signal based interrupting and handing-off system is implemented and integrated inside the abstract *Flow Core* class. This system is completely available

anyone implementing a new *Flow* or using an existing one.

During execution of the `evaluate` function, *Flow* implementer may choose to emit various signals. These signals' names are free form and don't have any maximum number. Similarly, *Flow* developer can emit the same signal many times, or emit different signals. Emitting signals does not create any measurable overhead. System works as follows. *Flow* class maintains an `unordered_map` to store the relations between signal names and the functions they call. In the beginning the map is empty and populated as the *Flow* user registers signals to the *Flow* before solution of the problem. When solution computation starts and a signal is emitted, the emit function looks whether signal name is present in the map. If yes, the corresponding function is called, otherwise computation continues without interruption.

This system allows great flexibility since *Flow* implementer can emit signals without complex programming and similarly, signals can be registered and unregistered without complex programming. Also, registering an non-existing signal will not produce any errors and halt the computation. Currently system allows a single handler per signal.

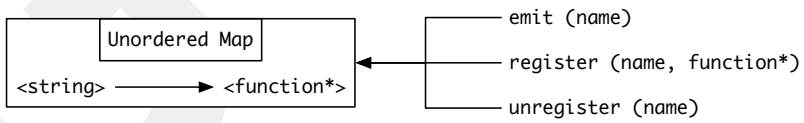


Figure 4.8: Framework callback system overview.

If a signal has a registered handler, solution computation stops until the handling function returns, hence the signal system is synchronous. Signals is implemented this way to protect *Problem* object's integrity and keep the programming constraints of the handling function simpler.

If a signal does not have a registered handler, the emit returns immediately to minimize the impact of the signal emit call. Algorithm and underlying data structures are chosen for their and low impact profile during emit, to keep the system performance high. The flow of callbacks can be seen in Figure 4.9.

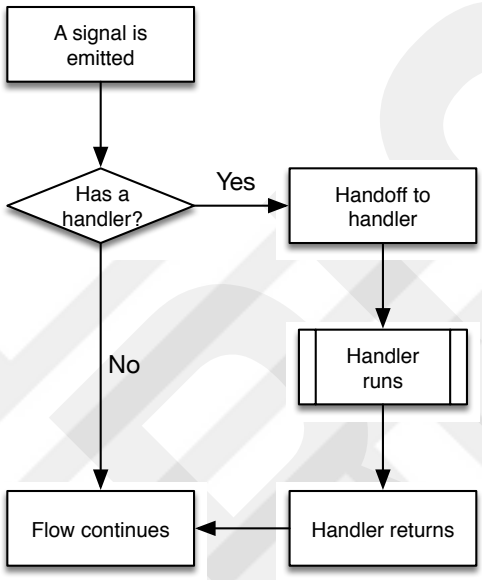


Figure 4.9: Framework callback process flow.

4.12 Integrals and Quadratures

Evaluation of integrals is a very important part of the BEM problem solution process, and its effect is two-fold. Since integral evaluation is the longest operation of the problem solution process, its duration directly affects the problem solution time. Also, its accuracy is directly related to the quality of the problem solution, so finding a way to obtain accurate integrations at shortest time possible is very important.

Gauss integration [50, 51] is a way to obtain integrations in a numerical way. In simple terms it needs a single *fixed point*, which is away from the *target object*, and a set

of *points* on the *target object* whose integration value will be obtained. This *set of points* are called *quadratures* and these points have a special pattern and associated *weights*, which are taken into consideration during the integration. These quadrature sets may have different number of points, and a suitable set is selected for the integration.

Gauss integration's accuracy is affected by two variables. One is the distance between target and fixed point and the other one is the number of points in the selected quadrature. As the distance increases, the accuracy which can be obtained with a particular quadrature increases. As the distance decreases, lost accuracy can be recovered by using a quadrature with more points, but with the expense of speed in the process.

Another disadvantage of the Gauss integration is the emergence of singular integrals. Singular integral can be defined,in numerical sense, as an integral which cannot be directly calculated by the numerical formula at hand. A singular integral appears when the distance between the fixed point and the target point becomes zero. This zero distance becomes a problem because in numerical formulations used in Gauss Integration, this distance is used as a divisor, and a 0 distance leads to *division by zero* condition. Unfortunately, emergence of singular integrals are unavoidable during problem solution process. They naturally emerge while taking the integrals to calculate the big G and H matrices, when the fixed point is on the *Mesh Element* which is being integrated at the moment. For a problem geometry containing N mesh elements, N singular integrals will emerge, since every *Mesh Element* is being integrated with each other, including themselves.

The B3F employs some special methods to keep the integration phase as quick as possible with greatest accuracy required by the problem at hand. To increase the ac-

curacy of the integrations, a novel algorithm, which keeps the percent error uniform, has been developed. This novel method also has some methods to handle singular integrals. To speed-up the process, the algorithm has been parallelized with a very high efficiency method, called lockless parallelization as aforementioned. These methods will be detailed in the following subsections.

4.12.1 Adaptive Element Subdivision Integration

As aforementioned, naive Gauss integration is a variable accuracy method, and accuracy of the problem solution is directly linked to the accuracy of the integrations. In order to increase accuracy of the integrations, a novel algorithm has been developed and implemented inside the B3F.

The algorithm is built upon the idea that taking an integration with more points is more accurate when taking a single integral over a large area, however instead of increasing the number of points in the used quadrature, triangles are divided into four *similar* triangles *in place* and their integrals are independently taken and summed. If the percent error between the single integral and the newly taken four integrals is higher than target percent error, every triangle is divided into the four in the same way, and process is repeated. If the percent error is smaller than the target error, the process is stopped. Time cost and performance of this process is discussed in Section 6.4.

This division process is called *Element Subdivision* and is not new. In literature, this process is mainly done by either dividing the mesh element into three triangles from its center, or by projecting the source point onto the plane of the integrated

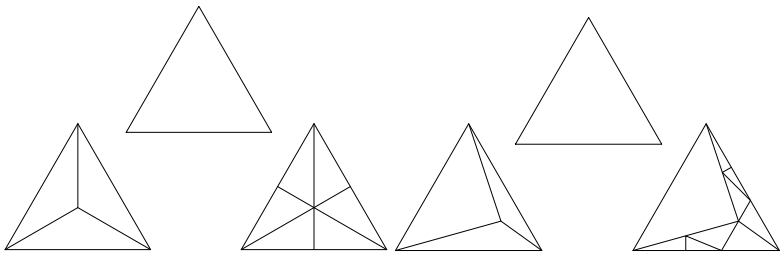


Figure 4.10: Two commonly applied subdivision technique for triangular shapes, (a) center point based divison, (b) projection based division. The left division is the first iteration, the right is the second iteration to obtain regular triangles

mesh element, and using this point as a pivot to subdivide the element [52]. In this methods, the triangles result from the division are not balanced. Either one of them become bigger than the others, or triangles with obtuse angles emerge as a result. To remedy these problems, another subdivision is generally required to balance the divided elements.

When these two methods are applied, it is important to note that, the divided triangles are not similar to the original triangle. In that case application of an adaptive method is also is not easy.

An exception to this subdivision technique appears in the study of Ali Yazıcı[53], where the triangular element is divided into similar triangles by *folding* the triangle - as given in Figure 4.11. The method can be considered to be adaptive, since it continues the subdivision process until a given convergence criteria is achieved. However, the major disadvantage is each subdivision process doubles the number of triangles, making $k^2/2$ sub-triangles for a process of k-fold, and the subdivision is not sensitive to the location of singularity or near singularity.

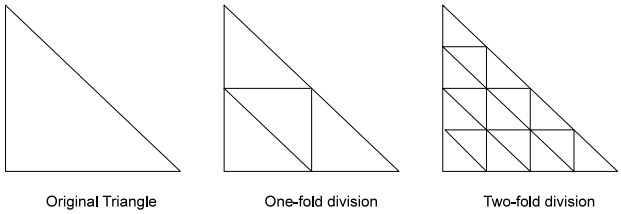


Figure 4.11: Subdivision by folding as presented in Ali Yazici's work, using Romberg integration scheme

Similarity of divided triangles plays an important role for this algorithm, since Gauss integral is more accurate when applied on regular triangles, rather than triangles with obtuse angles. Also, since the triangles are divided until the desired accuracy target is met, the algorithm is effectively converted from a constant time, variable accuracy algorithm to variable time, constant accuracy algorithm. This allows great improvements in solution quality. Also, since the integrations are independent from each other, this approach can be parallelized easily. For more details on parallelization; see Subsection 4.12.3.

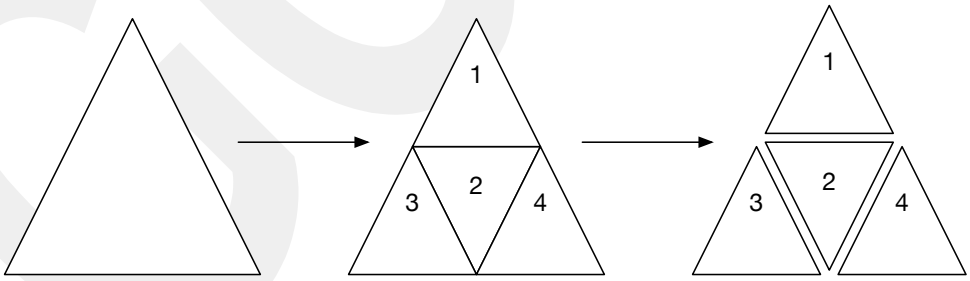


Figure 4.12: Triangle division process for adaptive integration.

Similar triangles are obtained by calculating a small triangle whose vertices are at the middle of edges of the parent triangle. When this calculated, the parent triangle is effectively divided into four as seen in the figure. These triangles are considered as

independent triangles and their integrals are taken independently. If the error percent is unsatisfactory, these triangles are considered as parents and divided to four again. This process continues until the requested percent error is met. The algorithm is summarized in the Figure 4.13.

The algorithm is designed to be blind to the depth of the process and only checks whether the required percent error is met or not, hence only control mechanism for completion of the process is the emptiness of the stack. If the error criteria is not met, divided triangles are pushed into the stack to be treated as parent triangles. If the required percent error is met, the triangles are discarded and the integration result is added to final result since the value is accurate enough. A visual representation of this process can be seen in Figure 4.14.

However, this method doesn't solve the Gauss integration's problem with singular integrals, so the algorithm deals with this issue with other methods, detailed in Section 4.12.2.

4.12.2 Handling Singular Integrals

As mentioned above, Gauss integration cannot calculate if the distance between the fixed and target object is zero. This zero distance problem arises due to distance calculation and selection of the fixed point. The fixed point is center of a mesh element, and the distance is calculated from this fixed point to the center point of the target mesh element. Since this integration is taken from all meshes to all meshes, there are instances where the fixed point and the target mesh element's center point is the same point, hence the difference between them is zero. To overcome this problem, during

integration, if the algorithm finds that the distance between the fixed and target points is *exactly* 0 in all three axes (i.e. x, y and z), the algorithm nudges the x, y and z coordinates of the fixed point by $1E-9$. While this value is very small in reality, this small distance enables Gauss integration to be able to calculate integrals, and since the integration algorithm detailed in previous subsection has uniform integration error regardless of distance, the result of the integration is not affected in a quantity which affects the accuracy of the problem solution.

This point nudging strategy is employed in both normal and internal point integration calculations with success. The accuracy and solution results for near-singular case, which point nudging strategy also creates, can be seen in Table 6.14.

4.12.3 Parallelization of Integration Algorithms

While adaptive element subdivision integration can take algorithms with great and uniform accuracy, this accuracy is not free in terms of time, especially with *near singular* and *singular* integrals. In fact, the developed algorithm can calculate more than *1.7 million integrations per second*[1] while calculating an integral for a single mesh element. To be able to decrease the time that integrals consume, with the exploitation of the fact that all integrations are independent from each other, a parallelization strategy has been developed for the integration routines.

The employed strategy is called *multiple consumer*. Since the jobs are predefined and produced long before consumers have started working, there is no *producer* in this scenario. These consumers communicate and synchronize over a single *atomic variable* and consume the integration jobs already produced as a by-product of the

[1] This rate test is done on the test system named *Rigel*. For details see Table 6.1.

process of problem solution. More details about atomic variables and their role in C++11 can be found in the previous section called *Parallelization Support* (Section 4.10).

The parallelization strategy works in cooperation with *Voluntary Thread Management* system, hence it can be tuned for both performance and resource preservation. After retrieving the number of allocated threads for itself, the algorithm creates same number of threads. These threads communicate over a single variable as aforementioned. This variable works as a *block identifier* (ID) and provides the correct matrix block's ID. The thread, by using operations special to atomic variables, retrieves and increases the variable in a single atomic operation, fetches the block pointed by the variable, makes the calculations, stores the result matrix block and starts over. Threads also know the maximum value for the block ID for the problem being solved and automatically terminates when all blocks are exhausted. For the details of the algorithm, please see Figure 4.15.

With this cooperation method, also known as blackboard collaboration, threads can consume the available jobs with great speed and efficiency. While will be detailed in the *Results* section, this parallelization strategy proved that the bottleneck in the performance of the algorithm is memory bandwidth, rather than CPU speed. This has shown that the CPU is used so efficiently that it can saturate its memory bandwidth during integrations. Considering *Eigen* is optimized for vectorized computation and has special optimizations regarding the locality of matrix data, making CPU to wait RAM only validates the efficiency of the algorithm.

4.13 Linear Algebra Operations

Solution of BEM problems require application of many linear algebra operations during the process. Since both the amount of operations done on the data and the required accuracy for this operations is high, using a well tested and respected external library rather than implementing required operations from scratch is a better choice.

In the B3F, this role is fulfilled by *Eigen* [22]. *Eigen* is a C++ oriented linear algebra library and provides all the features and facilities that the B3F needs in terms of linear algebra operations in an optimized and well packaged library. *Eigen* was the most attractive choice for various reasons. While *Eigen*'s performance is high and in the range of well respected libraries (BLAS, LAPACK, Intel MKL, ACML and others) [54, 55], *Eigen*'s integration to project and programming model is much simpler, well thought out and refined. These features allowed to develop the B3F faster, with much higher code quality and simpler programming techniques. Another advantage is the programming environment and inner workings of *Eigen* does not ignore presence of other significant libraries. *Eigen*'s internal data storage is directly compatible with other libraries aforementioned in this paragraph, so required data can be exchanged without any conversion in most cases if any need arises in the future.

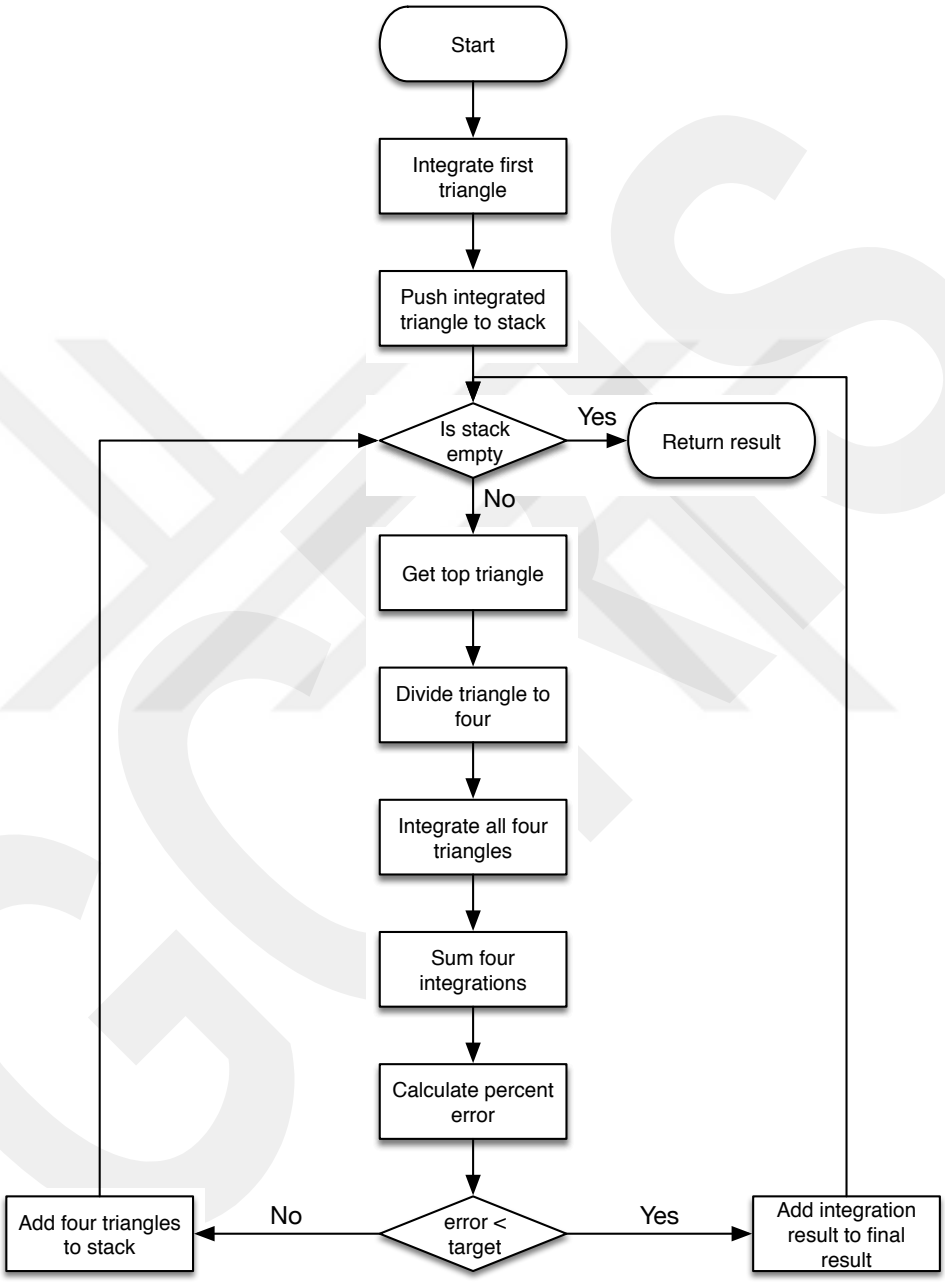


Figure 4.13: Adaptive integration algorithm flowchart.

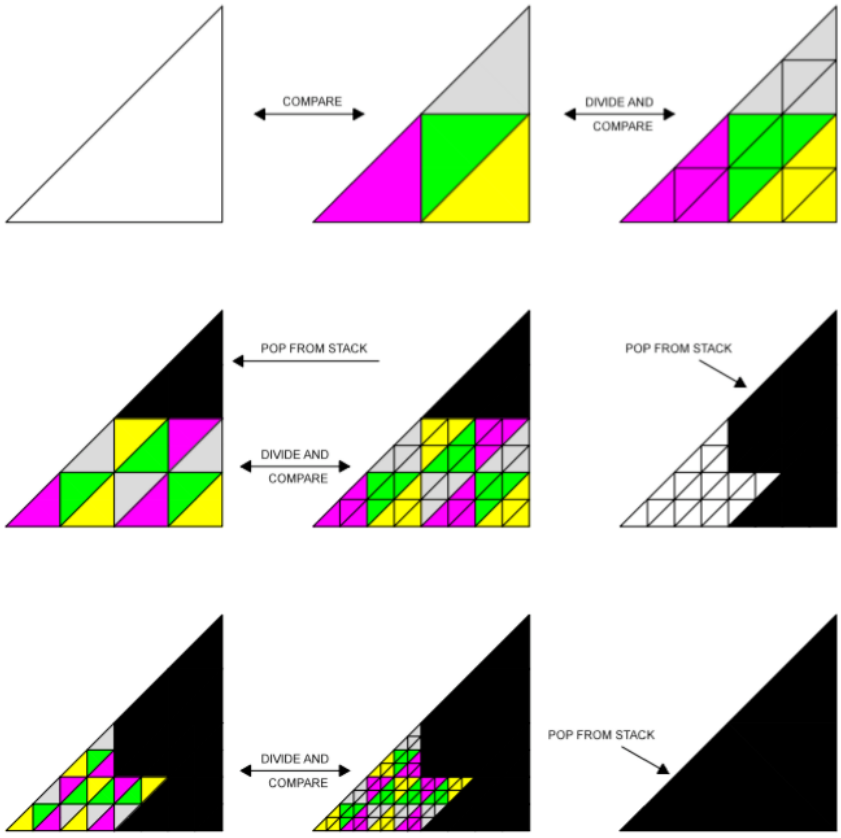


Figure 4.14: Evolution of an integral for a single triangle.

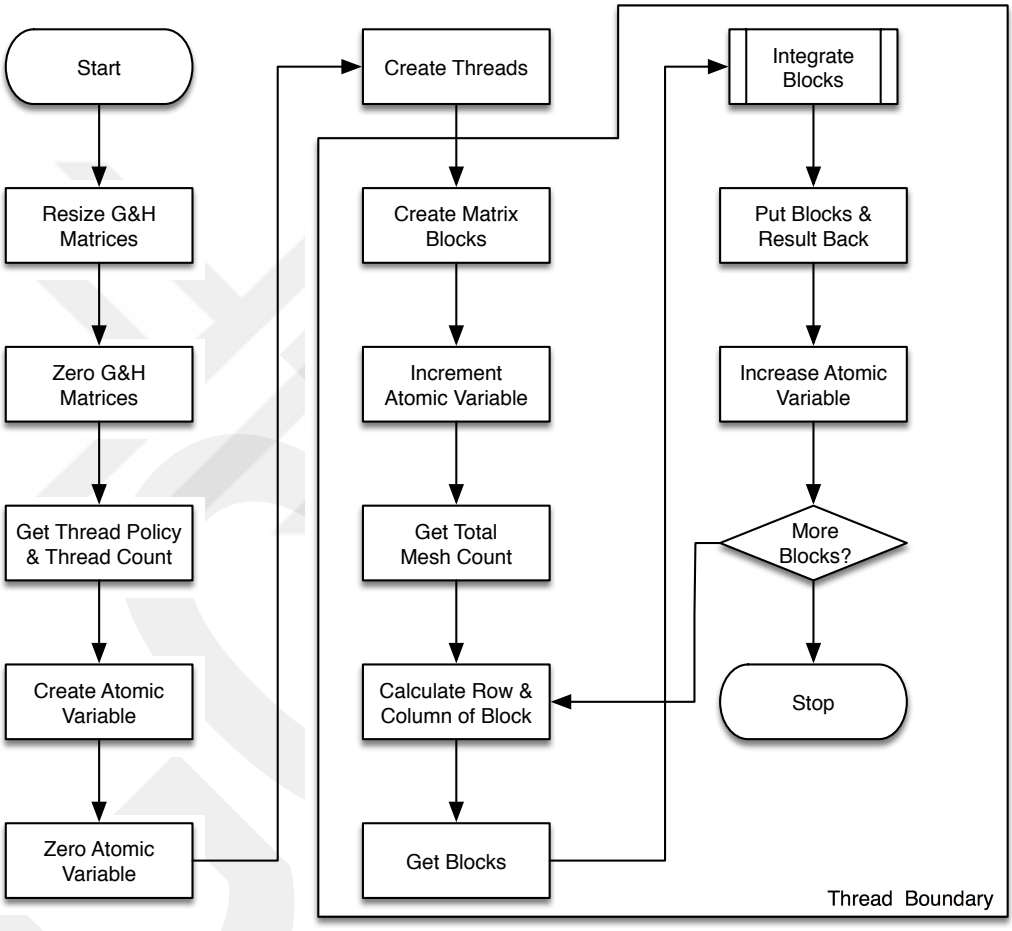


Figure 4.15: Parallel mesh element integration process.

CHAPTER 5

SOFTWARE ENGINEERING

Software development is a complex process, and this complexity increases exponentially as the developed software gets more sophisticated. Developing a scientific software, which can solve problems with a specific method and with high accuracy, requires monitoring of development process at every level. Because of these requirements, development of such software also creates its own challenges from the software engineering perspective.

To be able to make software development visible, verifiable and well managed, some processes have been implemented and executed during the framework development. This processes covered activities from planning and implementation to keeping track of source code and testing of implemented software. All the processes have been implemented using industry standard tools and with the state of the art software. All of the improvements in these practices are reflected back into the software development process to reap benefits from improvements of both processes and the tools used.

This section will detail the processes, used tools and the gained experience during development of the B3F.

5.1 Source Code Version Control

Source code versioning is the basic building block of software development work flow. It allows software development teams or individuals to work on features without affecting the stable version of the code, to return a known state back in time, or just to restart development from latest checkpoint easily.

Version control systems are not new, but its availability to individuals and small teams is relatively new. Currently the state of the art for software source code control is *Git* [56]. *Git* allows developers to work without connection to main repository via an active connection, and does not impose a centralized checkout or locking requirement like other VCSs like *CVS* and *Subversion*. Because of this reason, *Git* is considered as a decentralized VCS.

Being a decentralized VCS, *Git* creates an working scenario different than other VCS systems. Since *Git* does not limit which developers to work on which file or part of the source code, any two or more developer can make changes on the same part of the source code, and can cause a *conflict*. *Git* is aware of this by design, and has mechanisms to both sidestep the issue and solve it effectively if a conflict occurs.

Git encourages to work in *branch*es. *Branches* can be visualized as literal branches of a tree. They originate from a version of a code at a certain point in time. These versions are called *commit*s in *Git* terminology. These *branch*es are used for development of features in the codebase, and when the features are complete, they are *merge*d back to a special branch called `master`, which is generally is the main branch of the code. There are other more complex ways to mark the latest code and other ways to use branches, however this development model is called *feature branch*, and is one of

the most popular ways to manage feature development.

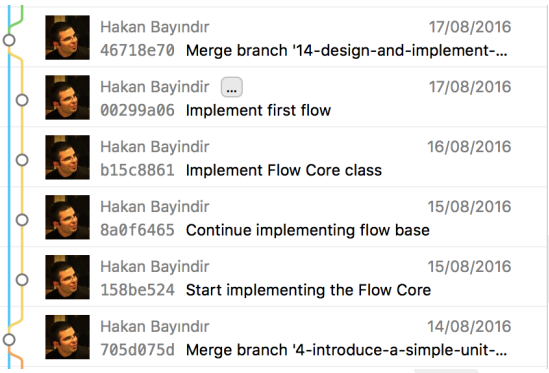


Figure 5.1: An example branching from framework development.

During code development, in some cases, a *merge conflict* can occur. This means, the target file which the changes will be applied is also changed, and changed in a way that *Git* cannot automatically figure out how to apply these changes. In other words, the changed lines in the file is also changed in the target file. In these cases, conflicts needs to be handled manually, however this is a rare occurrence in well managed teams.

Git's capabilities and tools allowed development of web services which not only visualize and allow remote access to the code, but integrate this workflow and allow it to be used by many people at once. *Git*'s decentralized structure, and very powerful branch and merge features have driven this development. Most prominent of these services are *GitHub* [57], *Bitbucket* [58], *GitLab* [59] and *Taiga* [60].

For hosting the B3F code in the university premises during the development, a local *GitLab* instance has been installed on a dedicated server. This *GitLab* instance has hosted the code, code documentation, code issues and the future plans. Effectively, *GitLab* is used as an all-around project management tool during development. As the name suggests, *GitLab* uses *Git* for underlying source code version control system

and allows code hosting and utilization of *Git*'s all features from the web interface. For feature development, *feature branch* method has been used. New features are branched out, and when development finished and code passed all tests, merged back to the `master` branch.

Using *GitLab* as a central repository brought many advantages. First, the source code has a dedicated hosting server, complete with backups, also the server allowed project management from a single, universally accessible point. Since there was a central repository, all development done on various computers are merged and exchanged from this single point, and allowed code to be developed and tested on various computers without big efforts. While detailed in later sections *GitLab*'s continuous integration features allowed the B3F to be higher quality with automated testing.

5.2 Software Project Management

Software consists of many parts, and these parts generally cannot be completed in one stint. Also, during development, bugs, regressions and design changes may happen. In short, software development have many facets. Within all this facets and complexity, the development has to be managed, and progress and tasks at hand should be kept visible and well organized. This is where software project management comes into play.

The B3F also consists of many parts and developed over a long period of time, and experienced all hardships and problems that a normal software project goes through, however through employment of some project management practices, all problems were manageable and kept visible.

Software project management is old as software development itself, and evolved as much as software development, if not more. Latest trends in software development is a set of ways called *Agile methodologies*. Agile methodologies have some variations, like *Scrum* [61], *Kanban* [62] and others, however most of them are oriented towards small to medium sized teams.

Since the development of the B3F has been done by a team of one; all planning, design and implementation is done by a single person, however this situation didn't prevent application of software project management practices.

Kanban is a method first employed by Japanese automotive manufacturer Toyota, and designed to visualize, and limit the work done simultaneously. *Kanban* boards contains cards and three states in most basic form. These parts are *To Do*, *Doing*, and *Done*. Every card contains a single task to be done and the card is moved from state to state during its lifetime. *Kanban* has revolutionized many project management processes since it allowed visualization of the work and the workload in a very expressive manner and made load balancing for teams easy. Since *Kanban* method only brings the board to the project and leaves the usage mostly liberal, it is much more suitable for teams of one. Because of this flexibility, Kanban has been successfully employed in the B3F development process.

The application of the methodology started since the planning of the B3F. Since *GitLab* didn't have an integrated board at that time, an external service, called *Trello* [63], is used for this *Kanban* board. The board used was the most basic variant, with three states as aforementioned. All work including design and feedback requirement is organized as cards. Trello's cards contain much more data than basic work like detailed description, comments, files, labels and responsible person. Leveraging these

features allowed to track the work, make it visible and collect all relevant information at one visible place. However while the board was hosting the work flow, the code and assets of the B3F was kept under the *GitLab* instance at the university.

For work tracking, *GitLab* has another facility called *issues*. These issues sport a simpler ticket based approach, which can contain a work description, a single responsible person, and an advanced commenting system. The advantage of this system is the ability to assign code branches in *Git* to these tickets. So, a work ticket has a branch attached to it, and all the work can be tracked over that code branch. In other words, the ticket system leverages *Git*'s liberal branch and merge workflow.

Because of this disconnection, all the work information was entered twice at first, however the returns in visibility, quality and speed was very high, and the net result was faster and more targeted development. As a result, double entry didn't cause any time loss. A *Kanban* board was implemented into *GitLab* in later versions, so board has been migrated into *GitLab* instance and the *Trello* board was closed.

In *GitLab* a four state board was implemented. These states were *To Do*, *Development*, *Testing*, *Done*. The state of the ticket was manipulated with the added tags to relevant ticket, so work state was also visible both in the integrated board and the issue list. Since the *GitLab*'s board was also integrated with source code via tickets, this method provided a tidier approach.

As a result, utilization of *Kanban* based work planning methods allowed to reduce mental load of work tracking and load balancing by providing a persistent place for visualizing, understanding and planning the work at hand, and allowed to keep the code quality higher and time to resume development shorter by acting as a state stor-

age for the developers. Another advantage that for anyone who wanted to supervise the work was able to see a detailed work report about all issues at a glance, keeping the transparency at highest level for the team.

5.3 Feature Planning and Implementation

The B3F has many intricate details and small features, and developed over a relatively long period of time. During this time period, planning and being able to follow the state of these plans allowed steady development of the source code.

Since the source code is relatively complex, nearly all parts of the B3F is planned and designed ahead of implementation. This is both good practice and a necessity when developing software beyond a certain scale.

Design of the software is done on various mediums. Most of them were so-called analog mediums like white boards and paper. These mediums were digitized later. Some of the more technical parts like data structures' smallest details are done on computer, with help of professional UML software, named *StarUML 2* [64]. All designs were verified before implementation either theoretically or with proof of concept implementation, hence "back to the drawing board" issues were kept minimal.

Planning and designing is not done on software implementation level only. Development of the B3F, implementation of the features and ordering of development tasks are also planned ahead for cleaner and more timely implementation. The implementation is done from bottom to top. When a layer or feature has completed, it enabled or built appropriate adapters or spaces for implementation even higher level functions and features. All this made possible with the help of *GitLab*'s project management

capabilities.

During development, framework had specific targets (e.g. Framework foundation implementation, enabling parallelization). These targets are modeled as *milestone*s inside *GitLab*. Then required tasks to complete these milestones are added as tickets to the issue tracker system. These tickets are implemented and closed with the most sensible order. If another issue has surfaced during milestone, its ticket is added to current milestone, and as the tickets are closed, the target is also achieved, step by step.

Using this approach also created a persistent state information about the project on the project management system, and since the progress on the milestone is updated automatically, it removed the burden and stress of following the state of the targets during development of the B3F. Hence, the only required focus point was the development of the feature described in the ticket itself, and since the design is done beforehand, nothing was ad-hoc in the process. Reducing the number of questions, and just implementing the issues reduced overall stress and improved both development quality and speed.

5.4 Continuous Integration and Testing

For ensuring the accuracy and correctness of any software during development, continuous testing is essential, and when the developed software starts to get bigger, more complicated and sophisticated, testing the software completely gets exponentially more difficult and time consuming.

Similarly, during development of the B3F, the code has to be tested periodically. Ide-

ally, before every commit. During early development phases, testing the code was simple. The B3F codebase had a binary, which contained some hard coded tests. This binary was compiled and ran. If the results were satisfactory, the code was committed to the repository.

When the code became more complex and started to compute parts of the problem solutions, testing for correctness involved a two step process. The problem parts were computed in mathematical software, and checked whether the results produced by the B3F were acceptable. When the results became acceptable, these results became the testing targets, the testing routines always expected to see these results every time.

After some point, since both the B3F developed even further, and the testing became time consuming; a complete testing solution to save time and increase the quality of the B3F was required. *Catch* [26] has been chosen to fulfill this role. *Catch* is a macro based library where the developer uses a simple macro syntax and C++ code to define tests to be executed, also with a special macro, this code is compiled into a fully functional binary, which supports individual or test set selection, performance tracking and other features. The binary can also produce colored, easy to read output to the console, so results are easily interpretable.

Introduction of *Catch* simplified testing significantly. After developing a feature and writing tests for the feature, the testing became completely integrated. The B3F had an integrated testing solution built into its source code tree, however running the test suite was still optional and manual. To improve testing further, and make it completely automatic, *GitLab*'s Continuous Integration (CI) subsystem has been introduced into the development process.

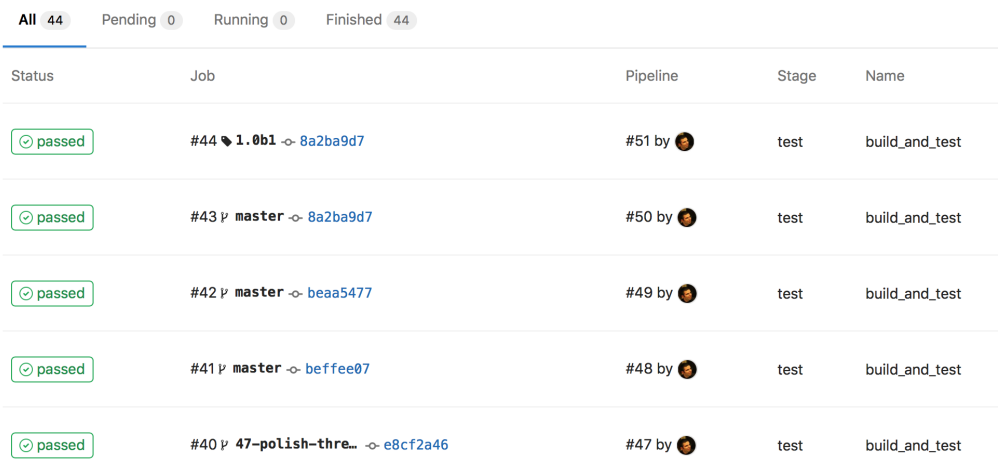


Figure 5.2: An excerpt of testing history from framework development.

Continuous Integration is a technology which is integrated into the development work flow, and means that the code is integrated (compiled, tested, and sometimes deployed to test system) in a continuous, automatic manner. CI is generally the step after the code is sent to the code repository. After the code is sent, CI automatically handles the code, and sends the result back to the responsible developer, or the team.

In the B3F development scenario, after the code is committed into the repository, regardless of the branch, *GitLab*'s CI client, which is installed another computer at the university starts to build a virtual environment inside a *Docker* [65] container. After container starts running, the C++ compiler is added, and the committed code is downloaded to that container. Next, the code is compiled, the test binary is run with all tests and the results are relayed back with an e-mail.

This process makes testing automated, mandatory and visible to everyone. Since all the testing history is retained, it is possible to take statistics, find regressions and performance improvements (since the B3F's test also contain time data in the logs).

As a result, automated testing enabled standardized and automated testing with result

preservation and removed human error from the equation.

CHAPTER 6

RESULTS

Development of a framework for solution of engineering problems using BEM with many novel ideas brings many results to discuss. Since these results are on different subjects and need different perspectives for correct evaluation, all of these results need to be evaluated individually. This individuality does not imply that these results are unconnected, but each of them are worthy of their own, independent discussion.

As a result, this section is divided into subsections, which highlight these aspects both in the context of the thesis and in their own light.

6.1 The Framework

The biggest outcome of the thesis is the development of the B3F for solution of engineering problems using BEM. The B3F, as discussed before has some requirements beyond simple problem solution like solution quality, accuracy, performance, extensibility and ease of use, among others.

The requirements of the B3F seemed challenging at first, since easy to use software platforms are seldom extensible and high performing, especially in the scientific com-

puting community. However, with the use of correct libraries, and modern programming techniques with new features, allowed B3F to have a very simplistic, yet expressive programming model.

The *Flow* concept is developed to pack long problem solution processes into compact and easy to use units. These units allow B3F to be converted into a simple, problem type specific standalone application if desired. These *Flows* can solve any number of problems of the same type, since all the required information is enclosed inside the problem file. Last but not the least, *Flows* can be used as the core of the application, or can be used as an integration point for more sophisticated applications (e.g. a visual application can solve problems via submitting required problem data to a *Flow* designed to solve a specific problem).

The data and in-memory models, which maps to each other perfectly allows great expressiveness while defining problems. The XML file format is well known and well matured and allows clear expression of problems to be used with B3F. Similarly, the interconnected in-memory data model allows easy access to *Problem* details while removing the burden of programming and geometry updating from the programmer and the user as much as possible.

The layered structure which contains the functions, which manipulates this data structure while computing the solution to the problem, both allows very high performance operation while allowing an high level of flexibility by decoupling computation from mesh application layer. By this way any user and developer can experiment with different strategies without modifying irrelevant parts to their work, hence both focusing on the work itself and not risking changing any unrelated code blocks and making changes bigger than intended.

All these features of B3F is extensively tested during the implementation of the *Elastostatic Flow*. After development of data structures, relevant parsers and *Flow Core*, the implementation of the problem was very straightforward. Since whole problem structure is available in the beginning of the problem solution process, without any additional efforts, all functions directly accessed to the *Problem* object, and manipulated the data structures instead of custom data structures developed to support the functions. This allowed much easier data logistics during the problem solution. As a result, the framework allowed to concentrate on the method of solving the problem, instead of thinking about the data structures and underlying features required to solve the problem effectively. This type of focused and accelerated development is the main purpose of a framework, and the B3F enabled the development of a solution with speed and focus.

Last, but not the least, since the aforementioned layering can effectively divide calculation algorithms from the application to the *Mesh Element*, different parallelization strategies can be effectively implemented into the B3F without much difficulty. Since the computation functions are layered as discussed in Section 4.6, any part can be independently parallelized. As shown in the Section 6.2, usage of this layering and lockless algorithms contributed highly to the performance of the B3F.

6.2 Problem Solution Computation Performance

During the development, the B3F has been continuously tested for performance. Since the aim was to develop a parallel framework from the beginning, the performance of the B3F only started to be recorded after the implementation of parallel

methods.

During the development of the B3F, a single problem has been utilized for correctness and performance tests. The problem was simple, but tested all parts of the B3F effectively. When the solution result is equal to the values at hand, this meant mathematical and algorithmic correctness, and speed and accuracy of the B3F can be correctly determined from the process completion time and the results.

The problem used in evaluation of the B3F is a simple elastostatic problem. The problem consists of a rectangular prism with base of 1 unit and height of 2 units. This results in a rectangular prism with $Width \times Height \times Depth$ dimensions of $1 \times 2 \times 1$ units. Prism contains 1000 mesh elements in total. These mesh elements are distributed as 100 elements per base and 200 elements per the side of the of the height of the prism. The prism of made of ST40 steel and compressed from its base with 1 unit of force. A single internal point is defined to measure stress and displacements inside. All systems and all tests used the same problem with same parameters for performance evaluation.

Since the B3F is developed with *hardest parts first* philosophy, the core mathematical functions are implemented first, and after stabilizing their results, the development on the mathematical functions are essentially stopped, as detailed in the previous parts of the thesis. This means the algorithms are optimized with best effort, only in the light of the simplest analysis, and with a time budget which allowed lowest hanging optimizations to be done. Paying heed to Donald Knuth's advice *Premature optimization is the root of all evil (or at least most of it) in programming* [66], this strategy paid itself with production of with well rounded, and well performing algorithms and code as the result. Since the parallelization layer is done at the *Toolbox Function* level

for integrations, process didn't require revisiting the mathematical foundation of the integrations.

The B3F is tested on three computer systems (will be shortly referred as systems). Two of them are servers, and one of them a desktop system, albeit a high end one for its time. The systems' crucial specifications are as follows. For clarity, the computers will be called with their names.

System	CPU Model	Speed[1]	Cores[2]	Memory Config	Total Memory
Orion	2x Xeon E5-2650	2000	2x8	24x16 GB	384 GB
Vega	2x Xeon E5-2630 v4	2200	2x10	4x16 GB	64 GB
Rigel	1x Core i7 3770K	3500	1x4	4x4 GB	16 GB

Table 6.1: Basic specifications of computers used for benchmarking.

Systems named Orion and Vega are two CPU socket systems. These systems have two physical CPUs, and total available RAM in these systems are divided among this two CPUs in equal manner. In other words Orion and Vega are systems with NUMA architecture. Rigel, on the other hand have a single physical CPU and this CPU have access to all RAM available to system, hence Rigel does not have NUMA architecture. Nevertheless, visual topology information of all systems can be found in the appendix.

All computers work under *GNU/Linux* based operating systems. Orion and Vega runs *CentOS 7.3* and *GCC 4.8.5*. Rigel runs *Debian Stretch* and *GCC 6.3.0*. Intentionally, compilers and operating systems' version are not frozen during the development. This was due to desire of developing a software which can be compiled without any warnings and perform well on a wide range of operating systems and compiler com-

[1] in MHz

[2] Physical

binations. During development a MacBook Pro laptop computer is also employed. This computer is running unmodified *macOS™* and *Apple LLVM/clang 8.0.0*. The B3F can also be compiled and run on this system without any modifications. While performance values of this computer is comparable to Rigel, the main purpose of development on this computer was to ensure portability and performance parity among *macOS™* and *GNU/Linux*.

Each computer has its own special feature. Orion has a very large RAM and lots of memory modules, Vega has the latest CPU extensions, and Rigel has a very high CPU frequency when compared to other systems. Since systems have different strengths and weaknesses, it imposes certain constraints on the B3F from different aspects.

Performance tests are done over four criteria. These are:

- **Integration time:** Time required to complete $1E6$ integrals.
- **Integration rate:** Number of complete integrations calculated in a second.
- **Integration speedup:** The speedup obtained by addition of more threads to integration process, compared to the single-threaded algorithm.
- **Integration efficiency:** The percentage of speed gain with addition of threads.

Formulas for calculation of the results are also given in their respective sections, further below.

The tests are scaled from single-threaded algorithm to the *maximum thread count* of the processor. Since all systems were supporting *Hyper-Threading Technology* [67, 68], this number is $2 \times physical\ core\ count$ of the CPU of the computer (please see 6.1). The thread counts in the tests are selected according to common thread

counts in servers used in HPC environments. For completeness, Pure single-threaded algorithm (Denoted by `1 Core`) and multi-threaded algorithm in single core more (Denoted by `1 Core MT`) are timed independently. The B3F is compiled using same optimization flag (`-O3`) in ll systems. This flag does not cause compiler to generate any CPU model specific instructions during compilation, thus the resulting binary is completely portable.

Next, details of the tests will be given as subsections, and meanings of these results will be analyzed independently at the end of each section.

6.2.1 Integration Time

Integration time is the most straightforward metric in the test suite, and measures the time required to complete calculation of $1E6$ integrals. This measurement is used as an input in all other calculations, and provides some interesting insight to the computer architecture and capabilities of the B3F. Integration times are given in the Table 6.2 for the three computer systems.

	Computers		
	Rigel	Orion	Vega
1 Core	286.0842	345.9372	249.1818
1 Core MT	257.13	441.213	256.6398
2 Cores MT	129.1998	222.0552	129.2208
4 Cores MT	68.5038	111.792	70.2252
8 Cores MT	50.728	61.7058	36.933
16 Cores MT	N/A	33.474	20.688
20 Cores MT	N/A	30.374	17.128
24 Cores MT	N/A	28.444	15.552
32 Cores MT	N/A	25.251	13.801
40 Cores MT	N/A	N/A	12.369

Table 6.2: Integration times (in $seconds$) for three computer systems.

In Figure 6.1, a plot of the integration times with respect to number of cores is given.

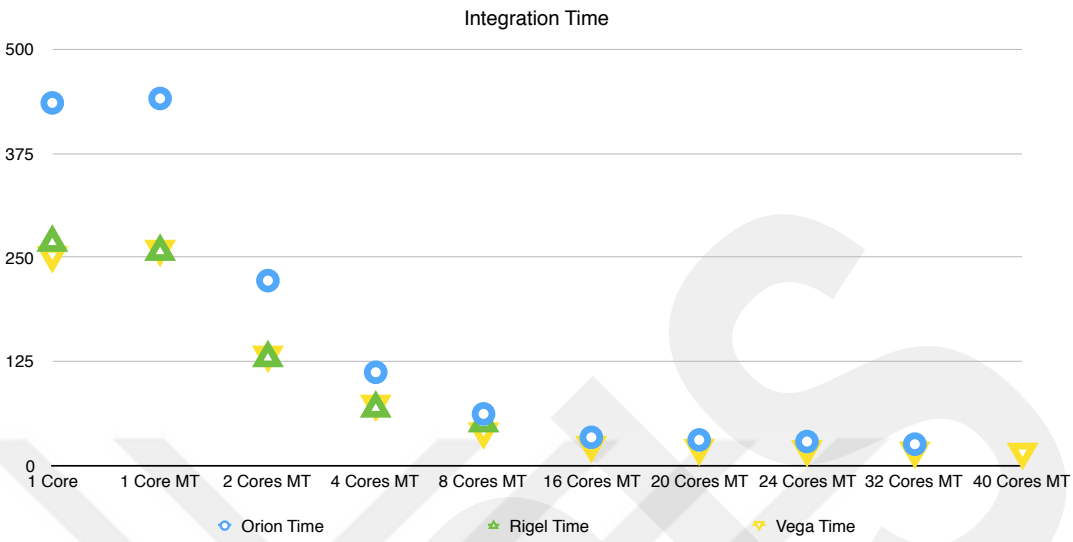


Figure 6.1: Integration times (in $seconds$) on three computer systems.

While results of this test is dependent on CPU speed, effects of architectural advances in CPU technology is clearly visible in the plot. In terms of CPU generation, Orion and Rigel have CPUs which are based on same architecture. Vega's CPU is much newer and has a more efficient architecture. This efficiency is visible on the plot. Despite lower frequency, Vega performs very similarly to Rigel, which has a CPU with much higher frequency.

Another interesting trend in the data is the time values after scaling beyond the physical core count (4 for Rigel, 16 for Orion and 20 for Vega). Since all CPUs have Hyper-Threading technology, the scaling tests are done at the maximum thread count the CPUs support, not the physical core count. This is done intentionally to measure the behavior of the CPUs and Hyper-Threading.

Since Hyper-Threading tries to keep a CPU core busier by running two threads on the same core, these threads need to do different jobs, or need to use diferrent parts of a

CPU core to be able to utilize the core to its fullest potential [69]. The B3F is, on the contrary, runs the same operations and thread functions, so Hyper-Threading cannot benefit the B3F notably in this scenario.

This lack of notable acceleration when the thread count increases beyond the physical core count also shows that the cores are already saturated with work. More details on this subject is thoroughly discussed in Section 6.2.2.

6.2.2 Integration Rate

Integration rate measures the number of *complete integration*s done in a second. Complete integration means a single round trip from the *adaptive integration* algorithm. Since a single adaptive integration call can calculate much higher number of integrals to calculate a single result, hence these numbers are not for a single Gauss integral, but for a single adaptive quadrature. The integration rate is calculated with the following formula.

$$\text{Integration Rate} = \frac{\text{Total number of integrals}}{\text{Integration time}} \tag{6.1}$$

The measured integration rates are given in Table 6.3.

When the data in Table 6.3 is plotted, the differences and highlights become more visible. Please see Figure 6.2

Again, this is a CPU speed dependent test, however the points where the slopes change are significant. Rigel's memory bandwidth is well balanced to its 4 cores, and it runs out of physical cores before memory congestion occurs, so the effect is

	Computers		
	Rigel	Orion	Vega
1 Core	3730	2293	4013
1 Core MT	3889	2266	3896
2 Cores MT	7739	4503	7738
4 Cores MT	14597	8945	14239
8 Cores MT	19712	16205	27076
16 Cores MT	N/A	29873	48337
20 Cores MT	N/A	32922	58383
24 Cores MT	N/A	35156	64300
32 Cores MT	N/A	39602	72458
40 Cores MT	N/A	N/A	80847

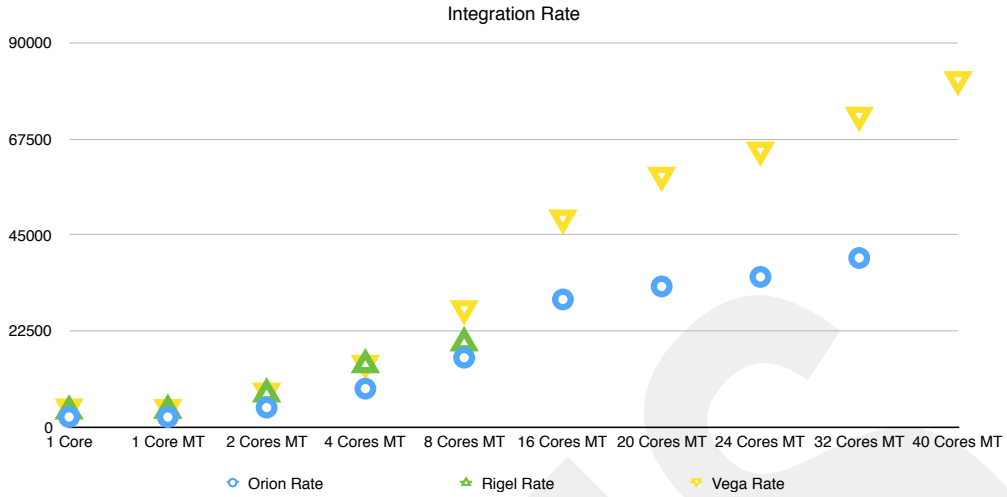
Note: Rate is adaptive integrations per second.

Table 6.3: Integration rates for computer systems.

not visible in the graph. However for Orion and Vega, the speedup slope breaks its usual curve at `16 Cores MT` point and turns to a linear speedup. This is because of possible memory bandwidth saturation.

To further understand the phenomenon visible at 16 Cores MT point and beyond, the issue is further investigated using profiling tools on the system named Orion. To profile the system, three tools are used. First is *STREAM Benchmark* [70, 71], which can determine usable memory bandwidth of the system, and used in the industry for benchmarking. Second is *SysBench* [72] tool, which is a benchmark geared towards general system benchmarking and database benchmarking. Last one is the *Perf* tool [73], which can record and allow analysis of events which occur at kernel and hardware level to better understand features and behavior of an application.

The mesh integration step, which evaluates the integrals to calculate and populates the G and H matrices requires a lot of random memory access with very small block sizes during the calculations. In the *Elastostatic Flow*, the integration algorithm obtains two 3×3 sub matrices, works on these matrices, calculates the results and puts them



Note: Integration rate is adaptive integrations per second, higher is better.

Figure 6.2: Integration rates on computer systems.

back to relevant matrices. For compatibility and easy data exchange with other linear algebra libraries, *Eigen* stores all matrices column-major as default [74] (please see the paragraph after the table in coefficient accessors section). On the other hand, the matrices in the B3F are declared `dynamic` when their size is unknown, and this results the matrices to be dispersed in the memory as vectors in the said column-major order [74] (please see the section called Fixed vs. Dynamic). This means, during every iteration of the integration, for every mesh pair being integrated, 6 random memory accesses are made to obtain requested sub matrices. These matrices' are very small. Since the B3F uses `double` for matrix data storage, which is 8 bytes, requesting 18 elements will result in fetching of 144 bytes every iteration.

Requesting a lot of small data fragments from RAM hurts memory performance of a system in a considerable manner. To showcase this phenomena, two benchmarks are conducted with *STREAM Benchmark* and *SysBench*. The main difference is *STREAM Benchmark* is for showcasing the potential of a system and *SysBench* is to understand

behavior of a system under certain type of load.

To objectively show the potential of Orion, *STREAM Benchmark* is tuned to use $80\ Million$ cell arrays, repeat its test 100 times, compiled with *OpenMP* support and optimized with *-O3* flag of the *GNU C Compiler* (GCC). *STREAM Benchmark* has four tests, called *Copy*, *Scale*, *Add* and *Triad*. Results are as follows.

	Tests			
	Copy	Scale	Add	Triad
1 Thread	6490	11156.3	11737.1	11841.5
2 Threads	14732.5	23286.2	24917.9	25062
4 Threads	27828	39000.3	42774	42967
8 Threads	47994	46256.1	50438.8	50457.2
16 Threads	61299.9	45520.7	49817	49527.8
20 Threads	51996.7	43463.6	47109.6	47497.8
24 Threads	58094.7	44342.7	48864.8	49048.4
32 Threads	58830.5	43974.1	48606.4	48400.5

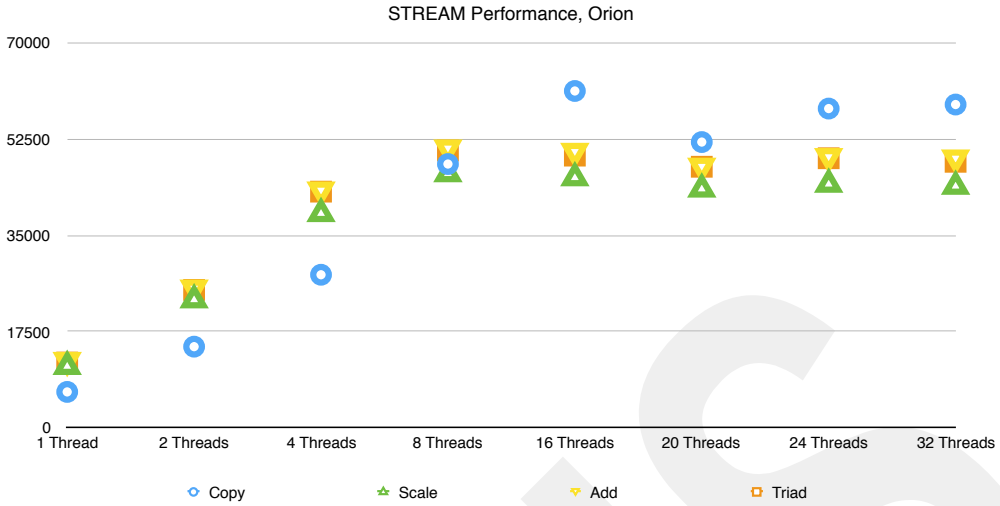
Note: All values are MB/s. $1MB$ is 1024×1024 bytes.

Table 6.4: STREAM Benchmark Performance for Orion.

The performance numbers are given on Table 6.4 is plotted on Figure 6.3, and shows the behavior of the test system clearly.

Orion has 4 memory channels per CPU socket, which makes 8 memory channels total, hence the memory transfer speeds leveling for *Scale*, *Add* and *Triad* after 8 threads is an expected result. *Copy* test can scale up to 16 threads. When the nature of the *Copy* test, which has no arithmetic operations, and low latency of the memory subsystem is combined, these results are not extraordinary.

When *SysBench* is run on the same system, the results are completely different. *SysBench* is run with $1\ Kilobyte$ blocks, to copy $180\ Megabytes$ in a *random* pattern. Tests are executed 5 times for each thread count, results are averaged. The benchmark



Note: Values are MB/s. Higher is better.

Figure 6.3: STREAM Benchmark performance of system named Orion.

results are as follows.

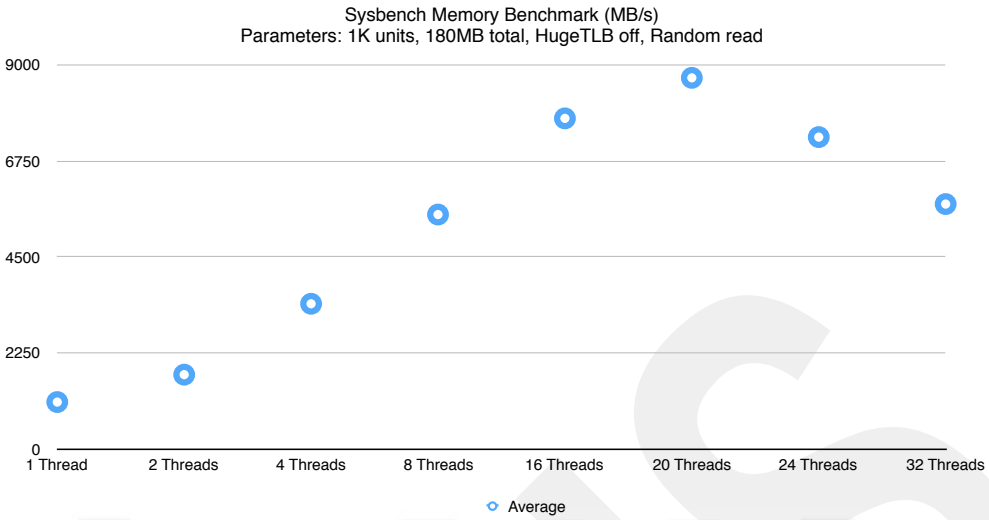
	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1 Thread	1171.03	1073.5	1117.96	1036.22	1135.94	1106.93
2 Threads	2106.44	2075.59	1473.05	1325.94	1762.06	1748.62
4 Threads	3780.81	3827.56	2967.44	2591.12	3877.93	3408.72
8 Threads	7081.1	4371.2	4506, 47	4427.28	7105.84	5498.38
16 Threads	6735.32	8204.21	7522.61	8038.09	8249.99	7750.04
20 Threads	7919.37	8812.08	9300.89	8815.51	8648.02	8699.17
24 Threads	7965.36	7267.98	7071.83	7414.93	6837.91	7311.60
32 Threads	6262.01	5867.65	5502.31	5603.96	5480.83	5743.35

Note: All values are MB/s. $1MB$ is 1024×1024 bytes.

Table 6.5: SysBench Performance for Orion.

The data in Table 6.5 is plotted on Figure 6.4. The plot shows the differences in a much clearer way.

As seen in the 6.4, Orion is able to scale up to 20 threads, however the numbers are much lower than *STREAM Benchmark*. This scaling capacity comes from the low latency of the memory subsystem. After 20 threads, Orion cannot keep up with the



Note: Values are MB/s. Higher is better.

Figure 6.4: SysBench Benchmark performance of system named Orion.

large number of memory requests and transfer rate starts to drop.

Another way to monitor these bottlenecks is to monitor CPU events with *Perf*. Modern CPUs from both Intel and AMD have event counters which detect and count various events happening in the CPU during execution. Linux Kernel's *Perf* subsystem can access this information with minimum overhead, and profile a system near real-time. B3F is also profiled with *Perf* to understand the situation during execution. Following events and performance metrics are obtained directly from CPU.

- **Cache Misses (CM):** Amount of calls to cache, in terms of percent, is missed and required data load from higher levels of memory (in this case RAM).

- **Stalled Cycles, Frontend (SCF):** In Intel CPUs, frontend means the instruction decode and dispatch section. This counter provides information, in terms of percent, about how much of the instruction decode and dispatch cycles were spent idle, waiting for information from memory, or waiting a free pipeline to

dispatch the uOP for execution.

- **Stalled Cycles, Backend (SCB):** In Intel CPUs, backend means uOP execution pipeline, where the instruction are executed. This counter provides information, in terms of percent, about how much of the instruction execution pipelines were idle, waiting for information from memory, or waiting a resource, such as FPU or divider, to be free.

- **Instructions Per Cycle (IPC):** This value is a calculation, however provides valuable information about throughput of a CPU. Every CPU has a theoretical upper instruction per cycle limit, and this upper limit is dictated by its design. Approaching to this number shows that the CPU is used very efficiently during the run. Smaller number means stalls, inefficiencies, and other problems in code. These problems can be remediable or not, depending on the structure and nature of the code.

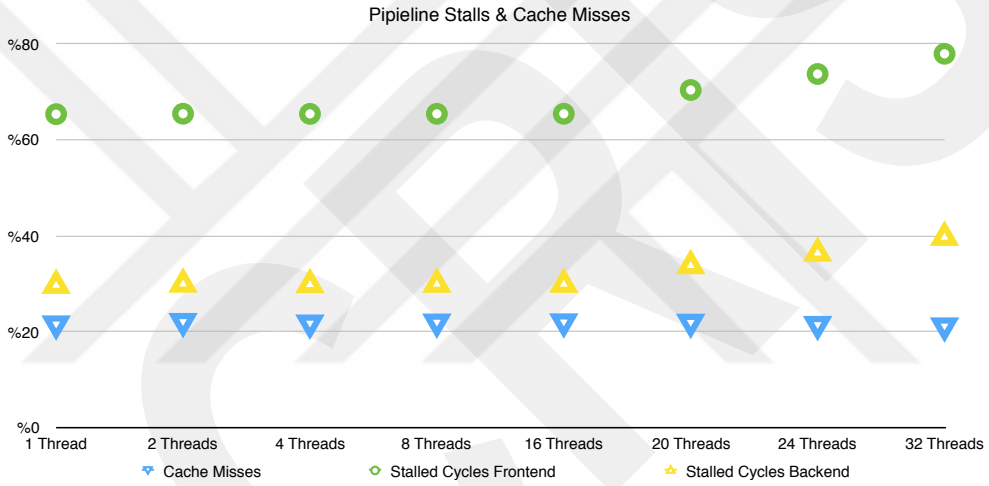
- **Stalled Instruction Per Cycle (SIPC):** Similar to previous calculation, but giving the number of stalled instructions per cycle. Higher numbers means the CPU is waiting for memory or an internal resource more often. Again these problems can be remediable or not, depending on the structure and nature of the code.

As can be seen in the Figure 6.5, the cache misses are steady, meaning the code causes invalidation of 20% of the cache data continuously, and uniformly regardless of thread count. This is due to small, random memory accesses which brings in both usable and unusable data, however it shows that since the cache is invalidated in the same amount in a thread-independent way, cache miss is not a bottleneck here.

	Cache Miss	SCF	SCB	IPC	SIPC
1 Thread	21.08%	65.39%	29.72%	1.16	0.56
2 Threads	21.594%	65.48%	29.97%	1.16	0.56
4 Threads	21.268%	65.45%	29.86%	1.16	0.56
8 Threads	21.468%	65.46%	29.94%	1.16	0.56
16 Threads	21.512%	65.47%	29.93%	1.16	0.56
20 Threads	21.428%	70.43%	33.82%	1.16	0.70
24 Threads	20.972%	73.77%	36.43%	0.89	0.83
32 Threads	20.665%	77.98%	39.74%	0.76	1.03

Note: IPC and SIPC values have no units. Others are percent.

Table 6.6: Event profiling results for Orion.



Note: Values are percent. Lower is better.

Figure 6.5: Cache miss and pipeline stall values for Orion.

As seen in Figure 6.5 again, stall related numbers start to climb after 16 threads, while instruction per cycle starts to drop as seen if Figure 6.6. As aforementioned, Orion has 8 memory channels but relatively low latency RAM allows the B3F to scale up 16 threads. In other words, memory subsystem can hold up against the flood of small, random requests with the help of latency performance. After 16 treads, the requests start to be too fast for memory subsystem to keep up, so performance starts drop.

For stall numbers in threads lower than 20 (1-16), since the back-end stalls are much

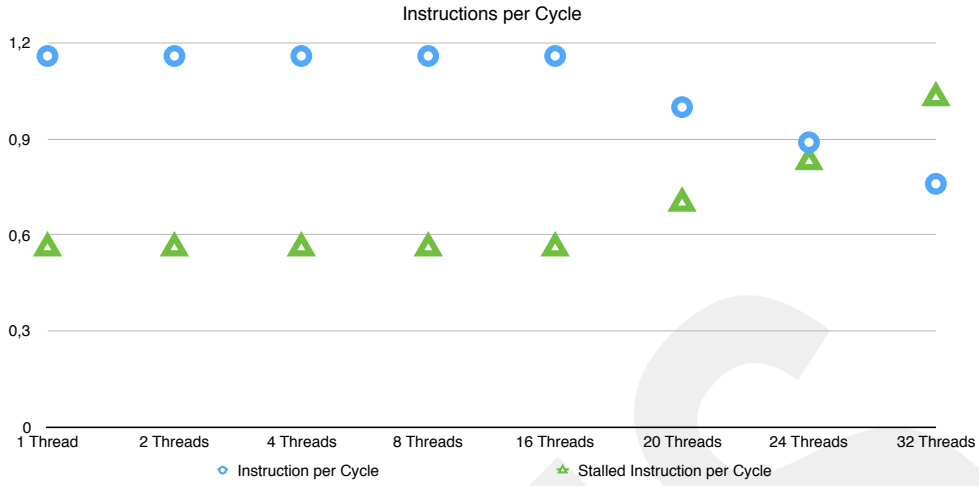


Figure 6.6: Instruction per cycle performance of system named Orion.

lower than front end stalls, it can be said that the CPU is relatively busy working with the integrals, and the front end (dispatcher) of the CPU is stalling to wait for the back end (execution pipeline), however this stall numbers may mean imbalanced utilization of resources in the CPU. Since the code is not compiled with any CPU specific extensions, regardless of Intel and AMD, vectorization units and other SIMD subsystems may not used by the integration codes developed. When compiled with -O3, *Eigen* uses explicit vectoring and other optimizations to utilize the CPU fully.

As a result, while the B3F is saturating the memory subsystem's capacity with small and random accesses, this does not mean that the B3F is exhausting all possible potential. With more targeted and intricate optimizations, the B3F may be accelerated. It is important to reiterate that the optimizations done in the B3F are best effort at this point and possibly have room for improvement.

6.2.3 Integration Speedup

Integration speedup test tries to highlight affects of adding threads to the integration algorithm. If the speedup numbers starts to decrease with the addition of more threads, this means the algorithm has reached its saturation point. The integration speedup is calculated as follows.

$$\text{Integration Speedup} = \frac{\text{Single core integration rate}}{\text{n core integration rate}} \tag{6.2}$$

The speedup values for all three systems is as follows.

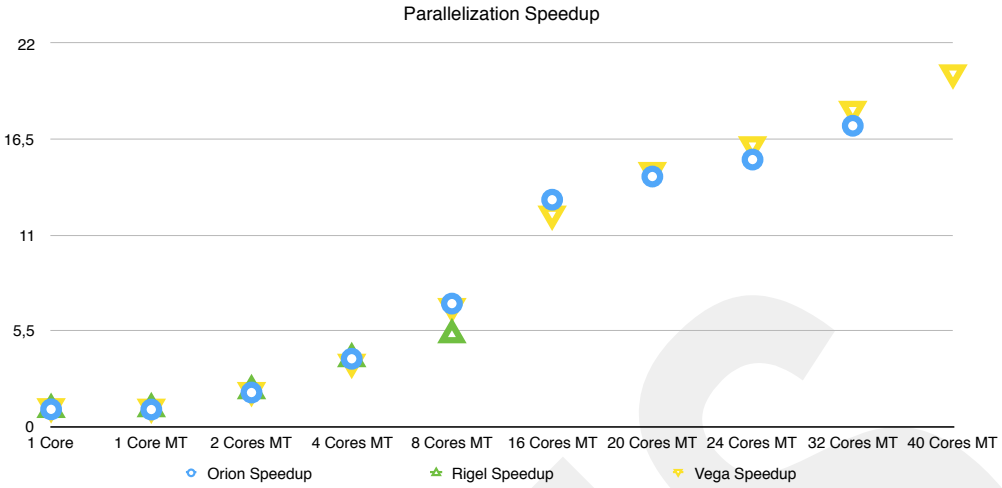
	Computers		
	Rigel	Orion	Vega
1 Core	1.0000	1.0000	1.0000
1 Core MT	1.0426	0.9880	0.9709
2 Cores MT	2.0749	1.9631	1.9283
4 Cores MT	3.9134	3.8995	3.5483
8 Cores MT	5.2847	7.0647	6.7469
16 Cores MT	N/A	13.0231	12.0448
20 Cores MT	N/A	14.3523	14.5482
24 Cores MT	N/A	15.3261	16.0225
32 Cores MT	N/A	17.2641	18.0553
40 Cores MT	N/A	N/A	20.1457

Note: Speedup is in *times*, relative to *1 core*.

Table 6.7: Integration speedups for computer systems.

Figure 6.7 provides a plotted view of the data given in Table 6.7, and illustrates the behavior of the code in a more visible way.

This is one of the two tests which remove CPU speed difference from the results, by taking CPUs baseline performance as their reference value. Results of this tests mirror the results of integration rate tests. However since the lines are now overlapping, the memory bandwidth saturation issue on Vega is much clearer. Since the lines overlap



Note: Speedup is in *times*, higher is better.

Figure 6.7: Integration speedup values on computer systems.

nearly perfectly for unsaturated part, and very close for the saturated part, it can be said that the algorithm can scale well on different CPU architecture revisions, and the code is generic enough that it can scale well on different generations of the same CPU. If the code contained special optimizations for a particular CPU family, the graph would be much different.

6.2.4 Parallelization Efficiency

Parallelization efficiency tests highlights contributions of each thread into the integration speed. In an ideal case, every time the thread count is doubled, the rate should double, and the time should halve, however this ideal is not always the case due to limitations both hardware and software. Efficiency tests remove the effects of CPU frequency and makes scalability differences between different CPU generations or types visible. Because of this merit, efficiency test is a good measurement method to

emphasize scaling characteristics of a CPU. The efficiency is calculated as follows

$$\text{Parallelization Efficiency} = \left[\frac{\text{n core int. rate}}{\text{1 core int. rate} \times n}\right] \tag{6.3}$$

The formula actually removes the speedup and shows the percent of speedup, relative to the single-threaded algorithm.

Algorithm's parallelization efficiency tests are as follows.

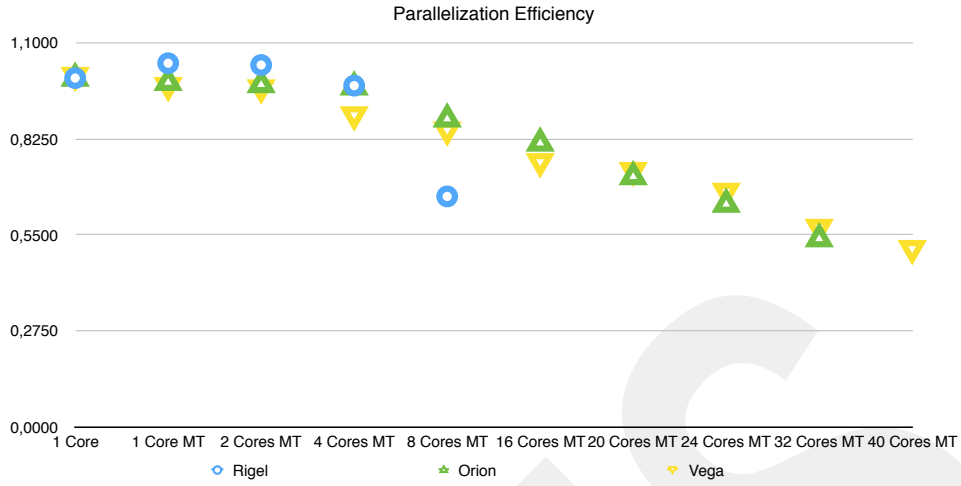
	Computers		
	Rigel	Orion	Vega
1 Core	1.0000	1.0000	1.0000
1 Core MT	1.0426	0.9880	0.9709
2 Cores MT	1.0374	0.9815	0.9641
4 Cores MT	0.9784	0.9748	0.8880
8 Cores MT	0.6606	0.8830	0.8433
16 Cores MT	N/A	0.8139	0.7527
20 Cores MT	N/A	0.7176	0.7274
24 Cores MT	N/A	0.6385	0.6676
32 Cores MT	N/A	0.5395	0.5642
40 Cores MT	N/A	N/A	0.5036

Note: Efficiency has no units, relative to *1 Core*.

Table 6.8: Integration efficiency values for computer systems.

The plot of the Table 6.8 can be found in Figure 6.8, which visualizes the results for clarity.

This test also removes the CPU's speed from the comparison and highlights the effects of addition of threads to the integration algorithms scaling capability. While the test simply calculates the difference between expected and real values, the plot reveals some interesting trends. First, Rigel scales extremely well until its physical cores are saturated. This can be due to two factors. The newer GCC used on Rigel tests, or lack of inter-CPU communication due to being the only single socket system



Note: Higher is better.

Figure 6.8: Integration efficiency values on computer systems.

in the test. The scaling efficiency of Orion is very high until `4 Cores MT` point, due to large number of RAM modules and higher number of available memory channels. Vega's scaling struggles become much evident here due to fewer number of memory channels available. On the other hand, the linear decrease in efficiency is understandable. While the integration algorithm is embarrassingly parallel, the queue is shared and next object's ID stored in a very fast, albeit shared variable. Using this variable at an increasing rate will add some unavoidable overhead to the algorithm. However the linearity of the decrease shows a well-scaling algorithm.

When all results combined, the parallel integration algorithm shows good performance. This results does not only cover the integration formulas, but matrix operations and other mathematical steps conducted by *Eigen*, hence these results and scaling performance shows that *Eigen* is also suitable for this kind of high performance programming tasks. As a result, it can be said that the parallel algorithm achieved its goals by being high performance and highly scalable.

6.3 Memory Consumption Performance

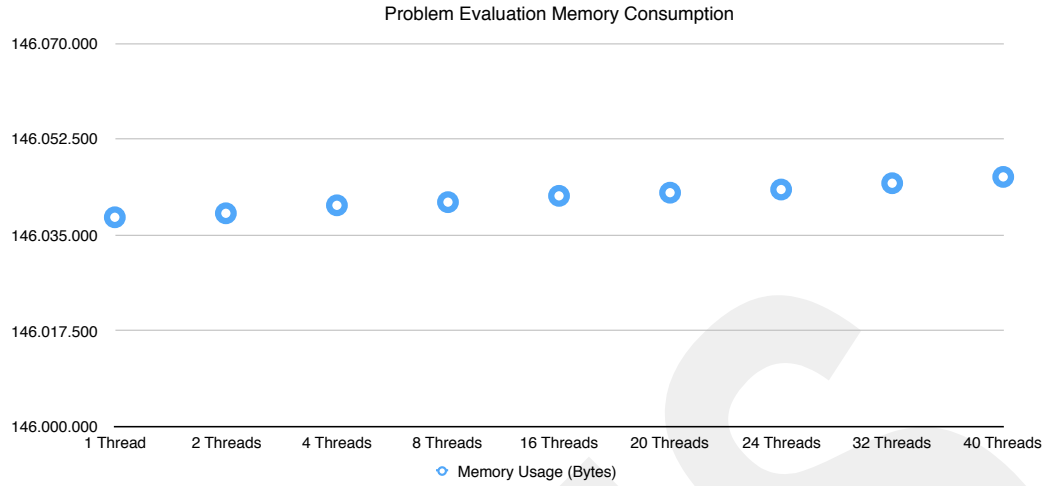
The B3F is also benchmarked for its memory consumption using the reference problem aforementioned in the previous section. The profiling is done with an instrumentation framework called *Valgrind* [75, 76, 77, 78]. *Valgrind* provides various tools to check or supervise the behavior of the applications, and for memory consumption, a tool called *Massif* is used. *Massif* can run the application inside its own environment, monitor and record memory allocation and deallocation event periodically, building a histogram of the memory consumption. In standard configuration, *Massif* takes a snapshot of the memory after a certain count of instructions executed. This mode is used in *Massif* runs made for benchmarking.

The memory consumption tests of the B3F framework is done on Vega, since it is the system with most cores and can provide information with more data points. Other systems are not profiled for memory consumption since they have the same CPU architecture, and memory consumption is independent of the system which the B3F runs. The memory consumption of the B3F is as follows:

	Memory Consumption (Bytes)	Memory Consumption (Kilobytes)
1 Thread	$146,038,232$	$142,615$
2 Threads	$146,038,960$	$142,616$
4 Threads	$146,040,416$	$142,617$
8 Threads	$146,040,992$	$142,618$
16 Threads	$146,042,144$	$142,619$
20 Threads	$146,042,720$	$142,619$
24 Threads	$146,043,296$	$142,620$
32 Threads	$146,044,448$	$142,621$
40 Threads	$146,045,600$	$142,622$

Note: $1MB$ is 1024×1024 bytes.

Table 6.9: B3F's memory consumption on test system Vega.



Note: Vales are bytes.

Figure 6.9: Memory consumption while solving the reference problem on Vega.

As the Figure 6.9 clearly shows, increasing thread counts increase memory consumption, but in a negligible amount. This consumption increase is due to stacks present in every thread which are used to store mesh elements which will be integrated for adaptive Gauss integration.

6.4 Adaptive Integration

The adaptive integration algorithm is developed before the framework itself, and it is subject to another set of benchmarks. In this section, the performance of the algorithm will be detailed with this specific set of benchmarks.

The first benchmark is an example from the work of Niu, *et al.* [79]. The integral

$$I = \int_T \frac{1}{r^n} dA \tag{6.4}$$

will be evaluated and be compared with exact values. In equation 6.4, n represents

the order of singularity. The triangle, which is evaluated is given in the Figure 6.10, below.

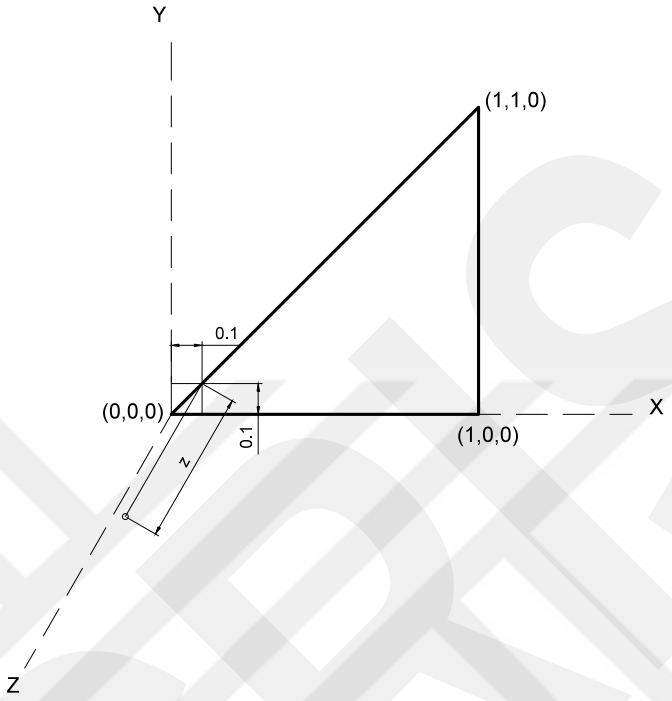


Figure 6.10: The location of the source point and the integrated triangular element for the benchmark problem

For the error measure, the same formula present in the [79] is used.

$$\text{Error} = \left\| \frac{I_{exact} - I_{evaluated}}{I_{exact}} \right\| \times 100 \tag{6.5}$$

For the evaluation, 1, 3, 5, and 7 point Gauss quadratures are used in the adaptive algorithm. Used quadrature is uniform for the master and divided triangles. i.e. the algorithms uses the same quadrature for entire process.

In order to compare the algorithm with the work of Niu, *et al.* [79], seven-point Gauss quadrature results are presented alongside the results in the aforementioned

work. Unless stated, the percent error target is given as $e = 10^{-4}$ for the newly developed algorithm.

As can be seen in Table 6.10, which compares the new algorithm with the previous work using the aforementioned error formula, the new algorithm provides results with constant accuracy independently from the location of the source point.

		% Error		
z	Exact	45 point GQ	Reference	New Algorighm
0.1	8.734547e+02	0.120e+00	0.207e-04	3.317e-10
0.01	1.046784e+06	9.108e+01	0.476e-11	1.089e-10
0.001	1.047197e+09	9.999e+01	0.342e-13	2.922e-10

Table 6.10: Comparison of the 7pt Gauss quadrature results from the current study with the 45 pt. Gauss quadrature in single triangle and the results presented in Niu *et al.*'s work.

Further comparisons are done with different number of Gauss quadrature points, which are 1, 3, 5 and 7. 1 point Gauss quadrature is the simplest quadrature and uses only the center of the target triangle which is being evaluated. Weights and details for other Gauss points can be found in [80]. The comparison, given in the Tables 6.11, 6.12 and 6.13 below where the error, number of triangles used in evaluating the integral and average solution time is tabulated respectively. All tests are executed on an Apple MacBook Pro (Mid 2014 model, with 2.8GHz Intel Core i7 processor and 16GB RAM), which had very similar performance to Rigel system. Timing runs are averaged over 100 runs.

It can bee seen that optimal performance (in terms of time, triangle number and accuracy) is obtained with the utilization of 7pt. Gauss quadrature. It can be further observed that the algorithm is giving constant accuracy despite the number of Gauss points or position of the source point. Furthermore, this accuracy target can be se-

		% Error			
z	Exact	1 Point GQ	3 Points GQ	5 Points GQ	7 Points GQ
5	1.525908e-04	2.37957e-07	2.36516e-09	1.08807e-08	4.97802e-09
1	2.309147e-01	2.26356e-08	4.94124e-10	2.94222e-09	1.16493e-09
0.5	3.069750e+00	9.39295e-09	1.96110e-10	5.56401e-10	8.31990e-10
0.1	8.734547e+02	8.99166e-09	4.37916e-10	1.38061e-09	3.31681e-10
0.01	1.046784e+06	1.78155e-08	1.11768e-10	3.80499e-09	1.08910e-10
0.001	1.047197e+09	2.63925e-07	2.97944e-10	3.26394e-09	2.92218e-10

Table 6.11: Comparison of GQ order concerning error.

	# of Triangles			
z	1 Point GQ	3 Points GQ	5 Points GQ	7 Points GQ
5	1398101	877	317	5
1	28957509	60621	4561	137
0.5	112538229	280757	11361	389
0.1	786885809	1662197	62213	2009
0.01	2635604145	6467601	213969	6649
0.001	4738866477	11631221	394749	12125

Table 6.12: Comparison of GQ order concerning number of triangles used in evaluating the integral.

lected by the developer or user of the application to further tune the framework to optimize the whole system for the requirements of the problem at hand.

For completeness, one last analysis can be presented for evaluating the performance of the proposed algorithm with different orders of singularity. For this, n is changed from 2 to 5. In the presented evaluations, 7-pt GQ is used in triangular integration. It can be seen, from Table 6.14, for all singularity orders n, the proposed method gives a good approximation with the exact value.

One last analysis for is done for evaluating the performance of the adaptive algorithm with different orders of singularity. For this, n is changed on a range from 2 to 5. 7 point Gauss quadrature is used for the tests, since it is the best performing quadrature.

	Solution Time			
z	1 Point GQ	3 Points GQ	5 Points GQ	7 Points GQ
5	0.276154	0 .000508	0.000215	0.000015
1	5.673585	0.030468	0.003018	0.000184
0.5	21.908165	0.141797	0.007989	0.00041
0.1	150.5679	0.795135	0.043747	0.002424
0.01	501.344437	3.015818	0.141509	0.007422
0.001	898.436216	5.273177	0.260369	0.013778

Table 6.13: Comparison of GQ order concerning average time to obtain solution.

					% Error	
z	Order (n)	Exact	# of Triangles	Sol'n Time	Adaptive Alg.	45 pt. GQ
0.1	2	3.4098	761	0.277	3.86E-11	0.35
0.01	2	10.1173	2509	0.928	4.79E-10	16.32
0.001	2	17.3440	4449	1.694	3.305E-10	49.04
0.1	3	17.4833	1065	1.115	1.226E-08	0.67
0.01	3	296.3034	3621	3.947	3.804E-10	50.86
0.001	3	3123.6750	6417	7.240	3.624E-10	94.84
0.1	4	116.1290	1477	1.764	1.76E-10	0.57
0.01	4	15636.6782	4945	5.535	1.36E-10	77.67
0.001	4	1.570E+06	8813	10.066	6.54E-10	99.74

Note: Time is in μsecs.

Table 6.14: Comparison for different singularity orders.

As can be seen from Table 6.14, for all singularity orders n, the adaptive algorithm's accuracy provides the capability to compute results with a good approximation to the exact value.

Elastostatic fundamental solutions are evaluated for the case of the source point is on the integrated triangular element as another benchmark. The exact solution for this scenario is given by [17] and later by [15]. In the benchmark, the integrated element is selected as the same element in Figure 6.10, with the change of the source point which is now located at the center point of the triangle, e.g., at $(\frac{2}{3}, \frac{1}{3}, 0)$. Analytical solution is obtained through the formulation presented by [15] and presented in Table 6.15

with unitless material properties selected as; shear modulus, $\mu = 1.0$ and Poisson's ratio, $\nu = 0.2$.

G Matrix		
0.161629822	0.003846090	0
0.003846090	0.161629822	0
0	0	0.131698374

H Matrix		
0.5	0	0.007360794
0	0.5	-0.007360794
-0.007360794	0.007360794	0.5

Table 6.15: Exact values of the G and H matrices for the given benchmark triangle.

Since the G matrix is symmetric, H matrix is anti-symmetric, and both contain zero values in the off-diagonal, only nonzero independent components are compared, which are; G_{11}, G_{12}, G_{22}, G_{33} and H_{11}, H_{13} and H_{23}.

As can be seen in Table 6.16, even a very low error bound of 10^{-3} gives very accurate results for the G matrix when compared to prescribed error bound. Similarly, the percent error in the H matrix, presented in Table 6.17, very accurate results are obtanied. Note that, the number of triangles used in the evaluation is $951,700$ for $e = 10^{-3}$, $3,875,716$ for $e = 10^{-4}$ and $28,317,812$ for $e = 10^{-5}$ giving the solution times of approximately $0.37s$, $1.42s$ and $10.86s$ respectively.

	Prescribed error bound (e)		
	10^{-3}	10^{-4}	10^{-5}
G_{11}	0.000290912	8.09257E-05	1.30545E-05
G_{12}	0.00034539	0.000121786	2.30104E-05
G_{22}	0.000294995	8.07401E-05	1.28689E-05
G_{33}	0.000292714	8.05629E-05	1.27564E-05
max	0.00034539	0.000121786	2.30104E-05

Table 6.16: Comparison of % error in independent components of G for different prescribed error bounds

	Prescribed error bound (e)		
	10^{-3}	10^{-4}	10^{-5}
H_{11}	2.46E-05	8.8E-07	5.6E-07
H_{13}	0.001794535	2.36659E-05	7.3253E-05
H_{23}	0.001201677	0.000434233	4.70737E-05
max	0.001794535	0.000434233	7.3253E-05

Table 6.17: Comparison of % error in independent components of H for different prescribed error bounds

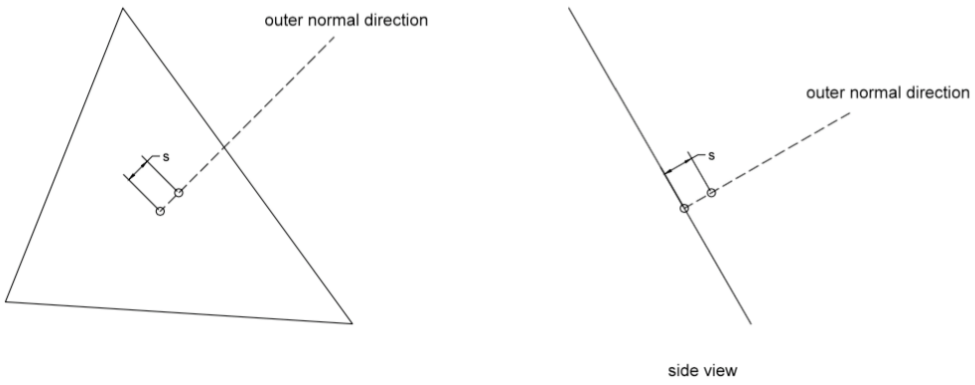


Figure 6.11: The change of position of the singular point on a triangular element

A last benchmark, using the same triangular geometry, but the location of the source point is carried out-of-plane of the triangle in the direction of its outer unit direction is done. The new location of the source point is varied, through a positive scalar value, s (see Figure 6.11). It is obvious that as $s \rightarrow 0$, the near-singularity increases and the components of the G and H matrices will approach to the singular exact values presented in the Table 6.15. For comparison, in Figure 6.12, the values of G_{11} and H_{11} are presented with respect to changing s. As expected, these values approach to the exact singular values as the value of s decreases.

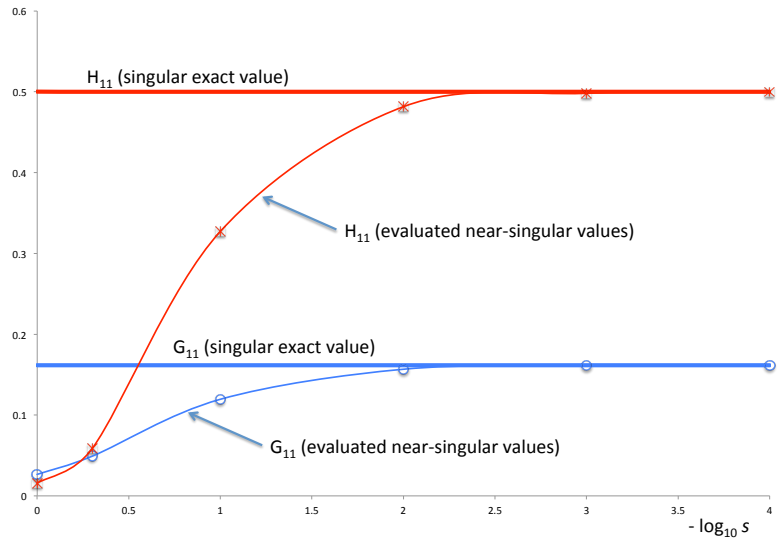


Figure 6.12: The change of position of the singular point on a triangular element

CHAPTER 7

CASE STUDIES

While performance of the B3F is studied in Chapter 6, analyzing the performance and behavior of the B3F by solving problems is not enough to analyze it completely. B3F is designed around some challenging non-functional requirements in order to enable easy extension and adaptability to wide range of engineering problems which can be solved with BEM.

In this chapter, the performance of the B3F will be analyzed by stressing these non-functional requirements in order to analyze the framework completely. To stress these features, a new problem type, Laplace problems, has been implemented and solved. Also, a more complex geometry is solved with the newly implemented *Laplace Flow* to test the accuracy of the Adaptive Gauss quadratures and the new *Flow* more thoroughly.

7.1 Implementation of Laplace Flow

To test the extensibility features of the B3F, another *Flow* has been implemented. Laplace problem solution has been selected for this task, because it has similar for-

mulations and solution process, however it is different enough to require acceptable amount of work for the implementation.

Implementation of *Laplace Flow* started with a copy of the *Elastostatic Flow*, since their solution process is essentially the same. Besides renaming of variables, the *Flow* class is essentially unchanged at the end of the implementation process. The *Low-Level Functions*, which contains the mathematical foundation of the problem solution process had to be modified. This was expected, since these functions are designed to carry the core computations and should be modified to allow different problems to be solved.

The implementation of the *Laplace Flow* did not go without small problems. These problems did not arise because of problems in the design or function layering of the B3F, but due to both inexperience in BEM itself, and some poor decisions made in data logistics (i.e. parameter passing from *Toolbox Functions* to *Low-Level Functions* and sizes of transferred matrices). All these problems can be solved without modifying the core and architecture of the B3F. These problems have been noted alongside their solutions to remedy the problem permanently, and a rather limited discussion of these matters can be found in the Future Work (Section 8.1). The gained experience is very valuable for developing truly universal *Toolbox Functions* which can be used more widely and with less effort. For the implementation of the *Laplace Flow*, these parameter passing systems are adapted without implementing the actual remedies, and this change is added as a short term future work.

On the other hand, the rest of the B3F has performed well. Besides the data logistics problems, all of the toolbox functions and logic of the algorithms needed no further modification. Adaptive Gauss quadrature, boundary condition handling, *Flow Core*

and other parts of the B3F worked as planned. As a result, the new *Flow* has been completely implemented and tested in three days, which can be considered remarkable.

During implementation of the *Laplace Flow*, a simple tool for converting multiple *STL* files to B3F's native on-disk data format has been implemented. The tool has been written with *Python* due to time constraints, but proved that the *STL* files can be converted with relative ease. It is important to remember that, since *STL* files contain only problem geometry; boundary conditions, materials and possibly other details also needed to be added to the XML file. A more capable tool for more complex conversion and containing features for easing problem specific information addition will be implemented in the future.

After implementation of the *Flow* and the *STL* importer tool, a test problem with 1290 elements is converted and solved to test solution quality and speed. The model in the problem is a quarter-cylinder, which is heated with $1\ unit$ of heat from above, and the bottom is held at reference temperature. During the solution of the problem, the Adaptive Quadrature algorithm performed very well. Actually, the percent error formula, which is used in Adaptive Quadrature had to be relaxed from Equation 7.1 to Equation 7.2 to prevent the algorithm from exceeding the requested accuracy. Even with very high error tolerances (e.g. 0.1 and 0.05), it produced very satisfactory results. With a lower tolerance of $1E-3$, the B3F can compute the solution precisely. Error tolerance of $1E-3$ is used for benchmark.

$$err_{percent} = \sqrt{\left|\frac{result_{curr}^2 - result_{prev}^2}{result_{prev}^2}\right|} \times 100 \tag{7.1}$$

$$err_{percent} = \frac{result_{curr} - result_{prev}}{result_{prev}} \times 100 \tag{7.2}$$

The results are compared with the exact solution of the problem and another BEM implementation which is done in *MATLAB*. The method is a naive implementation of Laplace solution, and uses a 42 point Gauss quadrature. None of the adaptive algorithms or similar methods are implemented inside the *MATLAB* code. Same problem is solved in both implementations and results (i.e. *temperature* and *flux*) are compared in tables 7.1 and 7.2.

$Soln_{exact}$	$Soln_{B3F}$	Err_{B3F}	$Soln_{MATLAB}$	Err_{MATLAB}
0.25	0.249774	0.090482%	0.249774	0.090482%
0.50	0.499992	0.001600%	0.499993	0.001400%
0.75	0.750197	0.026260%	0.750198	0.026393%

Table 7.1: Temperature calculation and percent error comparison between Exact, B3F and MATLAB solutions.

$Soln_{exact}$	$Soln_{B3F}$	Err_{B3F}	$Soln_{MATLAB}$	Err_{MATLAB}
0	-0.000300	-100%	-0.000300	-100%
0	-0.000415	-100%	-0.000416	-100%
1	1.000170	0.016997%	1.000175	0.017497%
0	0.000117	-100%	0.000117	-100%
0	0.000005	-100%	0.000003	-100%
1	1.001160	0.115866%	1.001164	0.116265%
0	0.000289	-100%	0.000292	-100%
0	0.000257	-100%	0.000256	-100%
0	1.000230	0.022995%	1.000224	0.022395%

Table 7.2: Flux calculation and percent error comparison between Exact, B3F and MATLAB solutions.

As can be seen from the solutions, the values between both implementations are extremely close to each other and to the exact solution. Some of the percent errors in Table 7.2 may seem high, however this is due to exact solution's value (i.e. 0).

When the arithmetic difference is considered between exact and the numeric solutions, the solution is perfectly acceptable. Results show that the Apative 7th order Gauss quadrature with 21 points can match the precision of the 42 point quadrature used by the *MATLAB* implementation. During a limited benchmark run using the system called *Orion* with 16 cores, the complete problem solution is obtained within 6.3 seconds. It is also worth noting that, the precision of the B3F can be increased, with the trade-off of computation speed, without modifying the code.

As a result, using the B3F, a new and completely reusable *Laplace Flow* is implemented in three days including a *STL* to B3F problem file converter. With the planned future work for better *Toolbox Functions* and other improvements targeted for easier development, the aim is to shorten the time required to implement new *Flows* or to experiment with existing ones.

7.2 Solving Problems with More Complex Geometries

In order to push B3F further, a harder problem with more complex geometry has been solved using the B3F. The problem geometry consists of a quarter disk with a radius of 200 units. The disk has a hole with a radius of 100 units. Whole model is composed of 1552 elements, and since it has two curved surfaces and is thin, this geometry generates many near-singular integrals. These many near-singular integrals create a hard job for the Adaptive Gauss Quadrature. Since the algorithm tries to reduce the percent error to the prescribed target, it ends up calculating much higher number of integrals at the end. The geometry of the solid, which is used in the problem can be seen in Figure 7.1.

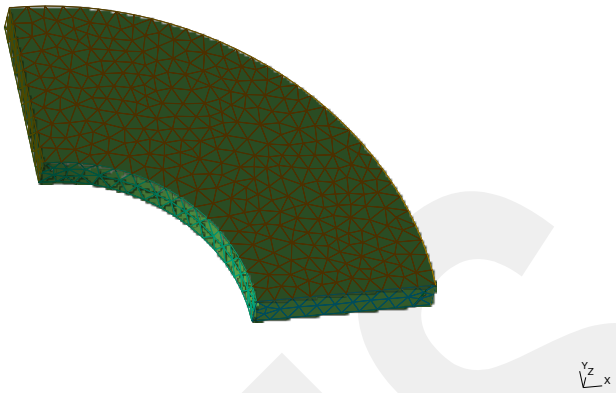


Figure 7.1: The solid for the complex geometry problem.

For the problem solution, the heat inside the disk is kept at $1\ unit$ and the outer arc surface of the disk is held at reference temperature. The k constant of the material is given as 1.0. The problem is solved with 11 internal points from inner to outer, including points in both extremes. The points, which are very close to boundaries are intentionally selected to stress the algorithms and the B3F even further.

$$x = r \times cos(45^\circ) \tag{7.3}$$

$$y = r \times cos(45^\circ) \tag{7.4}$$

The internal points' x and y coordinates are selected according to Equation 7.3 and 7.4 respectively. The z coordinate is fixed at 10 since this value is the middle of the disk, in z axis. However, due to hardness of the problem, and the time it takes to solve with default precision, the tolerance has been increased to $1E-2$ during solution of this problem. The selected points, with the respective results, and comparison with

the analytical solution calculated in *MATLAB* can be seen in Table 7.3.

Radius (r)	$Point_{internal}$	$Soln_{B3F}$	$Soln_{MATLAB}$	$Err_{Percent}$
101	(71.417, 71.417, 10)	0.984912	0.985645	0.074368%
110	(77.781, 77.781, 10)	0.863411	0.862496	0.106087%
120	(84.852, 84.852, 10)	0.743449	0.736966	0.879688%
130	(91.923, 91.923, 10)	0.622494	0.621488	0.161870%
140	(98.994, 98.994, 10)	0.515305	0.514573	0.142254%
150	(106.066, 106.066, 10)	0.415232	0.415037	0.046984%
160	(113.137, 113.137, 10)	0.321655	0.321928	0.084802%
170	(120.208, 120.208, 10)	0.233905	0.234465	0.238842%
180	(127.279, 127.279, 10)	0.151052	0.152003	0.625646%
190	(134.350, 134.350, 10)	0.073111	0.074001	1.202686%
199	(140.714, 140.714, 10)	0.006864	0.007232	5.088496%

Table 7.3: Temperature calculation error comparison between MATLAB (Analytic) and B3F calculations.

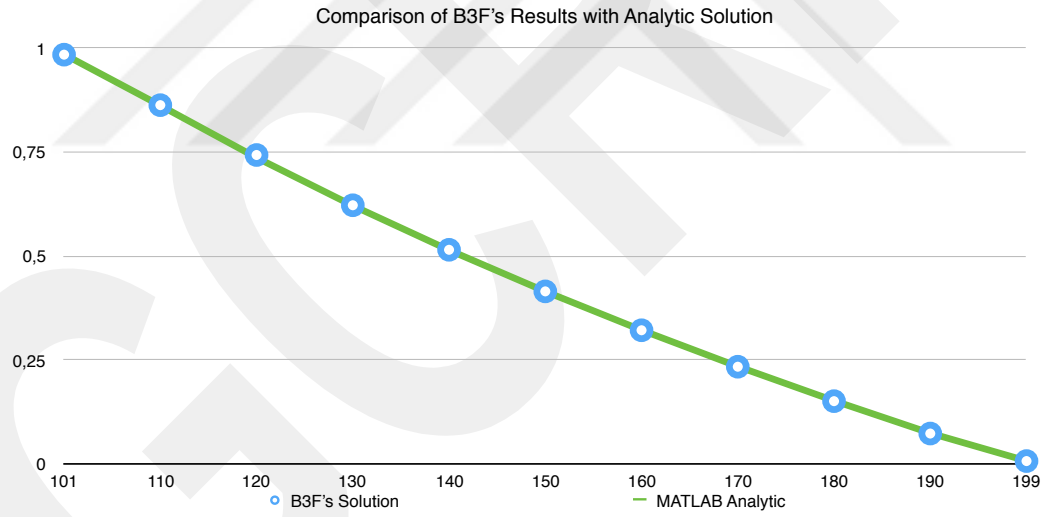


Figure 7.2: Comparison of B3F's results with Analytic solution.

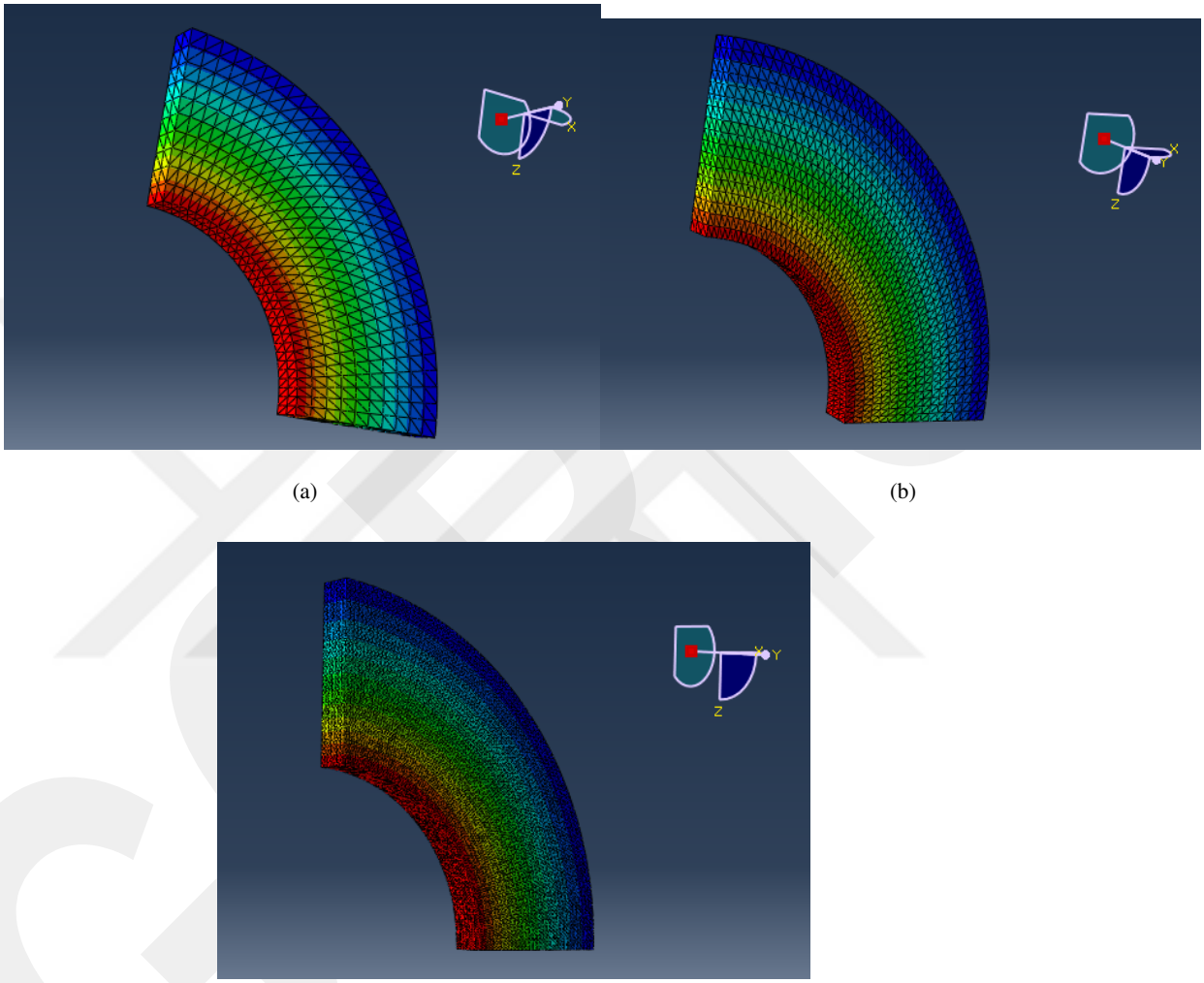
As seen in the results (Table 7.3), percent error values are higher when compared to error values in previous chapter. This situation has two reasons. First is the relaxation of error tolerance to $1E-2$, allowing the B3F to calculate faster, but with less precision.The second reason is the magnitude of the values. When values are compared with a normal difference formula ($|Soln_{B3F} - Soln_{MATLAB}|$), the difference

between values are not significant. Also it's worth noting that first two significant digits are completely same for both solutions except $r = 120$, which shows that B3F has satisfied the prescribed accuracy in 10 of 11 internal points. As a result, it can be confidently said that B3F can solve problems with the prescribed accuracy without any problems.

Also, as can be seen clearly in the Figure 7.2, the B3F's obtained results are perfectly in line with the analytic solution. Hence, it can be confidently stated that the B3F and its Adaptive Gauss Quadrature can solve a problem in a very accurate manner.

Another comparison is made between MATLAB's analytic solution, B3F, and FEM solution of the same problem. For FEM, *Abaqus* application is used. The problem has been modeled and solved using three different meshes, contaning 8952, 32364 and 217642 nodes. The MATLAB's analytic and B3F solutions are compared with the results obtained with the finest mesh. The results of the comparison can be seen in 7.4. Please note that, due to higher number of columns, only radii is given in the results table. The coordinates of the points are the same, and can be checked from Table 7.3 if desired. However, due to limitations of FEM, edge case radii are changed to 100 and 200 respectively.

As can be seen in the Table 7.4 and Figure 7.4, FEM's percent error values are higher when compared to B3F. When compared, it can be clearly seen that B3F can solve the same problem with much higher accuracy using much lower number of mesh elements.



(a) (b)

(c)

Figure 7.3: Meshes applied to the problem geometry in Abaqus, for FEM analysis. (a) Mesh with 8952 nodes. (b) Mesh with 32364 nodes. (c) Mesh with 217642 nodes. Results of finest mesh is used for comparison.

Radius (r)	$Soln_{FEM}$	$Soln_{B3F}$	$Soln_{MATLAB}$	$Err_{FEM,B3F}$	$Err_{FEM,MATLAB}$
100	1.000000	1.000000	1.000000	0.000000%	0.000000%
110	0.860470	0.863411	0.862496	0.341790%	0.234900%
120	0.733173	0.743449	0.736966	1.401579%	0.514678%
130	0.616239	0.622494	0.621488	1.015028%	0.844586%
140	0.508077	0.515305	0.514573	1.422619%	1.262406%
150	0.407461	0.415232	0.415037	1.907176%	1.825379%
160	0.321848	0.321655	0.321928	0.059966%	0.024850%
170	0.238492	0.233905	0.234465	1.923335%	1.717527%
180	0.154537	0.151052	0.152003	2.255123%	1.667072%
190	0.075200	0.073111	0.074001	2.778313%	1.620654%
200	1.23E-38	0.000000	0.000000	100.000000%	100.000000%

Table 7.4: Temperature calculation error comparison between FEM, MATLAB (Analytic) and B3F calculations.

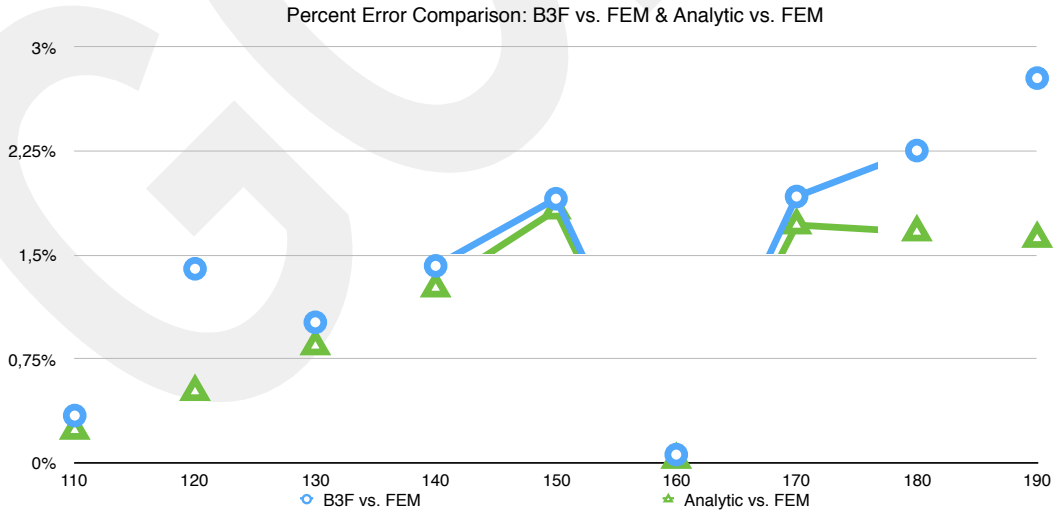


Figure 7.4: Percent error comparison of FEM with B3F and MATLAB.

CHAPTER 8

CONCLUSION

A new framework has been developed to serve as a basis for implementation of solution procedures of engineering problems using BEM, and an example procedure is implemented to evaluate Elastostatic problems.

The B3F explored implementation of high performance frameworks with C++11, and with utilization of the new features of the language, achieved high scalability numbers and high performance, without using any exotic or non-standard programming techniques or compiler extensions. The resulted code is completely ISO C++ standards compliant and can be ported among systems and compilers without any effort.

Again with utilization of the features of the language, framework explored sophisticated ways of implementing flexibility and extensibility into its structure. Instead of using programming models used by similar libraries, framework is developed with a novel idea of building an solution computation process out of completely functional blocks.

During the implementation of the problem solution *Flow*, a new adaptive integration method is introduced to minimize the effects of the variable accuracy property of the

Gauss integration method. The new method, albeit more expensive than unmodified Gaussian integration, allows integrations to be computed with a constant percent error, hence improving the precision of the solution vastly. With the integrated handling of singular integrations within the same algorithm, special treatment of singular integrals is not a necessity. The algorithm provides the following benefits.

- Subdivision of the elements continues until prescribed accuracy is achieved. Thus, the accuracy of the algorithm is secured.
- Subdivision continues in the direction of singularity, clustering the evaluation points through the projection of the singular (or near singular) point.
- The proposed method does not require a pre-determination of the distance of the point to the element, nor its projection on the element plane. It progresses independently from the position of the point with respect to the element.
- All sub-triangles originating from the original triangle being integrated are similar triangles to it. It is also possible to use this property to obtain the positions of the Gauss points within each triangle by simple translations and rotations instead of using the coordinate transformation, reducing the computational effort.
- Also, with the similarity property of each sub-triangle, it is not needed to evaluate the area of the newly formed sub-elements - each child will have an area that is exactly one fourth of their parent. This, also, is an important property which decreases the computational effort.

The parallelization support has been implemented again using C++11 features, allows native performance across different platforms, and have an *Voluntary Thread*

Management system which allows framework to run on wide range of platforms from personal computers to high end servers. Similarly, the libraries used with the B3F, while high performance, are completely integrated with the B3F, freeing users of the B3F from lengthy installation and compilation procedures, allowing framework to be used as a basis for a more advanced, single or multi-purpose BEM application.

To test the features of the B3F, a problem solution computation procedure (*Flow*) for elastostatic problems is implemented. The solution quality, with the help of adaptive integration, is quite high and evaluation speed is quite high. Also, with the help of the problem model, solution of problems are very simple and practical.

It is hoped that the B3F will evolve further in terms of both performance and features, and will be used in more problem types and real world applications, since the first trials shown that it is completely capable to handle real workloads and provide required solution quality.

8.1 Future Work

The B3F contains all the features and foundation it needs to have as a framework, however its other features are in infancy and is open to development and improvement. This small section summarizes these frontiers and opportunities for development and improvement of the B3F.

In the short term, a file writer for the problem results file shall be implemented to be able to save the problem results in a portable and standardized way. The *Problem* structure can be re-converted to a file similar to *Problem File* that the B3F parses for input. The included XML library *RapidXML* is capable of writing XML files

and selected with this requirement in mind. Another requirement is implementation of more problem types to evolve the B3F into a general purpose framework. While implementing these functions, it is possible that the function naming and some code re-organization is required. This will not invalidate any concepts and programming model. On the contrary, it will make the programming model easier and more clear. This was also known during development, so programming model is designed with these possible changes in mind. Also small utilities and tools, which will ease development and utilization of the B3F will be implemented and added to the source of the B3F. These utilities will be generally for improving interoperability with other file formats and applications.

Another required short term improvement is the further enhancement of data transfer interfaces (i.e. matrices and material properties) between *Toolbox Functions* and *Low-Level Functions*. These interfaces are implemented with some limitations due to inexperience in BEM space, and shall be improved with more flexible, yet compact data structures to make *Toolbox Functions* more universal. Without these modifications, some of the *Toolbox Functions* needs to be copied for adaptations, and this causes code duplication and bloat. Another improvement which can be applied to *Toolbox Functions* is awareness about problem parameters. Making these functions more aware about the problem-type independent parameters of the problem being solved can make *Toolbox Functions* much more flexible and lighten the load of developers and users during development.

In the medium term, optimization of the algorithms developed inside the B3F is considered. Since these optimizations are not considered "low-hanging" (i.e. trivial, easy), they will consume some considerable time, and requires experience. Also in-

cluding GPU support to the B3F is being considered. Since *Eigen* can work nVidia's *CUDA* [81, 82], The B3F shall not be modified extensively.

In the long term, converting B3F to a cluster-capable framework is being considered. This work involves incorporation of *MPI* [83] [84, 85] capabilities inside the B3F, and maybe optimizing the data structures for *MPI* usage will be required. Neither requirements, nor the effects of incorporation of *MPI* inside the framework, hence this work is left for long term development plans.

REFERENCES

[1] L. Gaul, M. Kögl, and M. Wagner, *Boundary element methods for engineers and scientists: an introductory course with advanced topics*. Springer Science & Business Media, 2013.

[2] U. Lemma and V. Marchese, "ACOUSTO - Acoustic Simulation Tool." `http://acousto.sourceforge.net`, 2008.

[3] I. van den Bosch, "Open Source Multi Level Fast Multipole Algorithm." `http://puma-em.sourceforge.net`, 2008.

[4] A. Babarit and G. Delhommeau, "Theoretical and numerical aspects of the open source BEM solver NEMOH," in *11th European Wave and Tidal Energy Conference (EWTEC2015)*, 2015.

[5] W. Śmigaj, T. Betcke, S. Arridge, J. Phillips, and M. Schweiger, "Solving boundary integral problems with BEM++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 2, p. 6, 2015.

[6] R. Hiptmair and L. Kielhorn, "BETL-a generic boundary element template library," *ETH Zürich*, 2012.

[7] U. Hohenester and A. Trügler, "MNPBEM–A Matlab toolbox for the simulation of plasmonic nanoparticles," *Computer Physics Communications*, vol. 183, no. 2, pp. 370–381, 2012.

[8] Y. Liu, *Fast multipole boundary element method: theory and applications in engineering*. Cambridge university press, 2009.

[9] J. Francés, S. Bleda, A. Márquez, C. Neipp, S. Gallego, B. Otero, and A. Beléndez, "Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 514–526, 2014.

[10] P. Gepner, V. Gamayunov, and D. L. Fraser, "Early performance evaluation of AVX for HPC," *Procedia Computer Science*, vol. 4, pp. 452–460, 2011.

[11] J. Brodman, D. Babokin, I. Filippov, and P. Tu, "Writing Scalable SIMD Programs with ISPC," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, (New York, NY, USA), pp. 25–32, ACM, 2014.

[12] H. Sutter, "C++ and Beyond 2012: Herb Sutter - atomic<> Weapons, 1 of 2." https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2, 2012. [Online; accessed: 12-Jul-2017].

[13] H. Sutter, "C++ and Beyond 2012: Herb Sutter - atomic<> Weapons, 2 of 2." https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2, 2012. [Online; accessed: 12-Jul-2017].

[14] H. Sutter, "C++ and Beyond 2012: Herb Sutter - atomic<> Weapons, Slides." https://1drv.ms/b/s!Aq0V7yDPsIZOgcI0y2P8R-VifbnTtw, 2012. [Online; accessed: 12-Jul-2017].

[15] P. K. Banerjee, *The boundary element methods in engineering*. McGraw-Hill, 2 ed., 1981.

[16] W. Hackbusch, *Numerical Techniques for Boundary Element Methods: Proceedings of the Seventh GAMM-Seminar Kiel, January 25–27, 1991*, vol. 33. Springer-Verlag, 2013.

[17] T. Cruse, "Numerical solutions in three dimensional elastostatics," *International Journal of Solids and Structures*, vol. 5, no. 12, pp. 1259–1274, 1969.

[18] K. Schwaber and M. Beedle, *Agile software development with Scrum*, vol. 1. Prentice Hall Upper Saddle River, 2002.

[19] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[20] "C++11 Standard Library Extensions — Concurrency." https://isocpp.org/wiki/faq/cpp11-library-concurrency, 2017. [Online; accessed 12-Jul-2017].

[21] H. E. Hinnant, B. Dawes, L. Crowl, J. Garland, and A. Williams, "Multithreading Library for Standard C++." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2320.html, 06 2007. [Online; accessed 12-Jul-2017].

[22] Eigen, "Eigen — A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.." http://eigen.tuxfamily.org/index.php?title=Main_Page&oldid=2335, 2017. [Online; accessed 02-May-2017].

[23] N. Styles, M. Bellomo, A. Salzburger, A. collaboration, *et al.*, "Developments in the ATLAS Tracking Software ahead of LHC Run 2," in *Journal of Physics: Conference Series*, vol. 608, p. 012047, IOP Publishing, 2015.

[24] M. Kalicinsk, "RapidXML — A very fast in-situ XML Parser." `http://rapidxml.sourceforge.net`, 2017. [Online; accessed 2-May-2017].

[25] M. Labs, "Easylogging++— Cross platform logging made easier for C++ applications." `https://muflihun.github.io/easyloggingpp/`, 2017. [Online; accessed 02-May-2017].

[26] P. Nash, "Catch — C++ Automated Test Cases in Headers." `https://github.com/philsquared/Catch`, 2017. [Online; accessed 02-May-2017].

[27] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns*. Wiley John + Sons, 1996.

[28] B. Semeraro and L. Gray, "PVM implementation of the symmetric-Galerkin method," *Engineering Analysis with Boundary Elements*, vol. 19, no. 1, pp. 67–72, 1997.

[29] G. Geist and V. S. Sunderam, "Network-based concurrent computing on the PVM system," *Concurrency and Computation: Practice and Experience*, vol. 4, no. 4, pp. 293–311, 1992.

[30] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency and Computation: Practice and Experience*, vol. 2, no. 4, pp. 315–339, 1990.

[31] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, *A proposal for a set of parallel basic linear algebra subprograms*, pp. 107–114. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[32] J. J. Dongarra and R. C. Whaley, "LAPACK Working Note 94 A User's Guide to the BLACS v1.," *Tech.£ eport*, 1997.

[33] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[34] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pp. 120–127, IEEE, 1992.

[35] B. A. Baltz and M. S. Ingber, "A parallel implementation of the boundary element method for heat conduction analysis in heterogeneous media," *Engineering analysis with boundary elements*, vol. 19, no. 1, pp. 3–11, 1997.

[36] N. Kamiya, H. Iwase, and E. Kita, "Parallel implementation of boundary element method with domain decomposition," *Engineering Analysis with Boundary Elements*, vol. 18, no. 3, pp. 209–216, 1996.

[37] N. Kamiya, H. Iwase, and E. Kita, "Performance evaluation of parallel boundary element analysis by domain decomposition method," *Engineering analysis with boundary elements*, vol. 18, no. 3, pp. 217–222, 1996.

[38] M. S. Ingber, J. A. Tanski, and P. Alsing, "A domain decomposition tool for boundary element methods," *Engineering Analysis with Boundary Elements*, vol. 31, no. 11, pp. 890–896, 2007.

[39] J. Song and W. Chew, "Fast multipole method solution of three dimensional integral equation," in *Antennas and Propagation Society International Symposium, 1995. AP-S. Digest*, vol. 3, pp. 1528–1531, IEEE, 1995.

[40] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. B. Williams, and K. S. Stanley, "An overview of the Trilinos project.," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[41] M. A. Heroux and J. M. Willenbring, "A new overview of the Trilinos project.," *Scientific Programming*, vol. 20, no. 2, pp. 83–88, 2012.

[42] M. Bebendorf, "Another software library on hierarchical matrices for elliptic differential equations (AHMED)," *Universität Leipzig, Fakultät für Mathematik und Informatik*, 2005.

[43] T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, and D. Roose, "Lecture Notes in Computational Science and Engineering," 2008.

[44] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, pp. 1309 – 1331, 2009.

[45] O. Sander, T. Koch, N. Schröder, and B. Flemisch, "The Dune FoamGrid implementation for surface and network grids," *CoRR*, vol. abs/1511.03415, 2015.

[46] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit 4th Edition*. Kitware Inc., 2006.

[47] Boost, "Boost uBLAS - Basic Linear Algebra Library." http://www.boost.org/doc/libs/1_45_0/libs/numeric/ublas/doc/index.htm, 2012. [Online; accessed 17-Jul-2017].

[48] 3D Systems Inc., "The StL Format." `http://www.fabbers.com/tech/STL_Format`, 1989. [Online; accessed 08-Jul-2017].

[49] T. Bray, J. Paoli, C. M. S.-M. E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation." `https://www.w3.org/TR/REC-xml/#dt-doctype`, 2008. [Online; accessed 08-Jul-2017].

[50] N. Bakhvalov and V. Motornyi, "Gauss Quadrature Formula." http://www.encyclopediaofmath.org/index.php?title=Gauss_quadrature_formula&oldid=34105.

[51] M. Abramowitz and I. Stegun, eds., *Handbook of Mathematical Functions*. New York: Dover, 1965.

[52] Y. Miao, W. Li, J. Lv, and X. Long, "Distance transformation for the numerical evaluation of nearly singular integrals on triangular elements," *Engineering Analysis with Boundary Elements*, vol. 37, no. 10, pp. 1311–1317, 2013.

[53] A. Yazici, "On the subdivision sequences of the extrapolation method of quadrature over a triangular region," *METU Journal of Pure and Applied Sciences*, vol. 23, no. 1, pp. 35–51, 1990.

[54] Eigen, "Eigen Benchmarks Main Page." http://eigen.tuxfamily.org/index.php?title=Benchmark&oldid=2152, 2011. [Online; accessed 13-Jul-2017].

[55] Eigen, "Eigen Performance Monitoring." http://eigen.tuxfamily.org/index.php?title=Performance_monitoring&oldid=2161, 2017. [Online; accessed 13-Jul-2017].

[56] J. H. e. a. Linus Torvalds, "Git - A distributed version control system.." https://git-scm.com, 2017. [Online; accessed 02-Jun-2017].

[57] "GitHub - The word's leading software development platform.." https://github.com, 2017. [Online; accessed 20-Jun-2017].

[58] "Bitbucket - The Git solution for professional teams.." https://bitbucket.org, 2017. [Online; accessed 20-Jun-2017].

[59] "GitLab - The platform for modern developers.." https://gitlab.com, 2017. [Online; accessed 20-Jun-2017].

[60] "Taiga - Love your project.." https://taiga.io, 2017. [Online; accessed 20-Jun-2017].

[61] I. Nonaka and H. Takeuchi, *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, 1995.

[62] J. M. Gross and K. R. McInnis, *Kanban Made Simple: Demystifying and Applying Toyota's Legendary Manufacturing Process*. AMACOM, 2003.

[63] "Trello - Work collaboratively and get more done.." https://www.trello.com, 2017. [Online; accessed 20-Jun-2017].

[64] "StarUML 2 - A sophisticated software modeler.." https://staruml.io, 2017. [Online; accessed 20-Jun-2017].

[65] "Docker - Build, ship and run any application, anywhere.." https://www.docker.com, 2017. [Online; accessed 20-Jun-2017].

[66] D. E. Knuth, "Computer Programming As an Art," *Commun. ACM*, vol. 17, pp. 667–673, Dec. 1974.

[67] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, *et al.*, "Hyper-threading technology in the netburst® microarchitecture," *14th Hot Chips*, 2002.

[68] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. Q1, 2002.

[69] Intel, "Using Intel® Hyper-Threading Technology." https://software.intel.com/en-us/mkl-linux-developer-guide-using-intel-hyper-threading-technology, 2017.

[70] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[71] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[72] "SysBench - Scriptable database and system performance benchmark.." https://github.com/akopytov/sysbench, 2017. [Online; accessed 03-Jul-2017].

[73] "Perf - Linux profiling with performance counters.." https://perf.wiki.kernel.org/index.php/Main_Page, 2017. [Online; accessed 03-Jul-2017].

[74] Eigen, "Eigen — A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms - Documentation about Matrix Class.." http://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html, 2017. [Online; accessed 02-May-2017].

[75] "Valgrind - an instrumentation framework for building dynamic analysis tools.." http://valgrind.org, 2017. [Online; accessed 03-Jul-2017].

[76] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley, "Tracking bad apples: reporting the origin of null and undefined value errors.," in *OOPSLA* (R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., eds.), pp. 405–422, ACM, 2007.

[77] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation.," in *PLDI* (J. Ferrante and K. S. McKinley, eds.), pp. 89–100, ACM, 2007.

[78] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building Workload Characterization Tools with Valgrind.," in *IISWC*, p. 2, IEEE Computer Society, 2006.

[79] Z. Niu, W. Wendland, X. Wang, and H. Zhou, "A semi-analytical algorithm for the evaluation of the nearly singular integrals in three-dimensional boundary element methods," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 9-11, pp. 1057–1074, 2005.

[80] D. Dunavant, "High Degree Efficient Symmetrical Gaussian Quadrature Rules for the Triangle," *International Journal For Numerical Methods in Engineering*, vol. 21, no. 6, pp. 1129–1148, 1985.

[81] nvidia, "nVidia CUDA Zone." `https://developer.nvidia.com/cuda-zone`, 2017. [Online; accessed 09-Jul-2017].

[82] nvidia, "nVidia CUDA Home Page." `http://www.nvidia.com/object/cuda_home_new.html`, 2017. [Online; accessed 09-Jul-2017].

[83] J. M. Squyres and A. Lumsdaine, "The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications* (V. Getov and T. Kielmann, eds.), (St. Malo, France), pp. 167–185, Springer, July 2004.

[84] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A Flexible High Performance MPI," in *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, (Poznan, Poland), September 2005.

[85] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

APPENDIX A

APPENDIX

A.1 UML Diagrams of Framework Classes

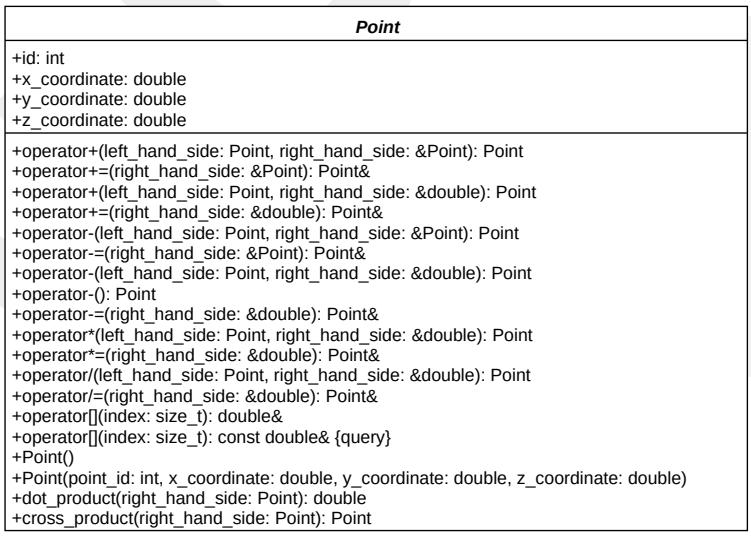


Figure A.1: Detailed view of Point class.

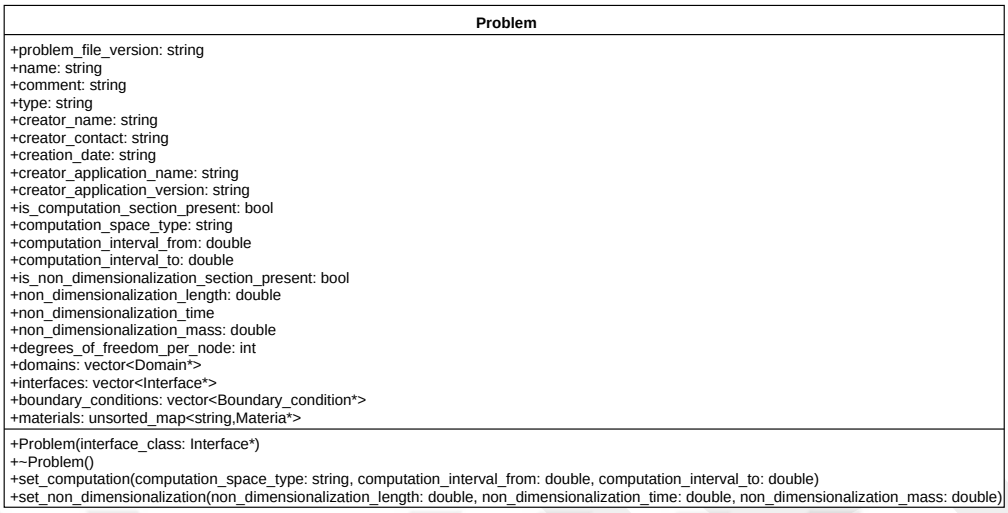


Figure A.2: Detailed view of Problem class.

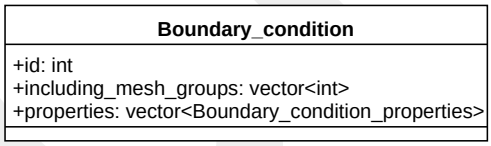


Figure A.3: Detailed view of Boundary Condition class.

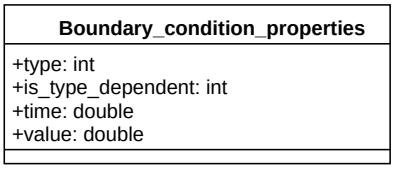


Figure A.4: Detailed view of Boundary Condition Properties class.

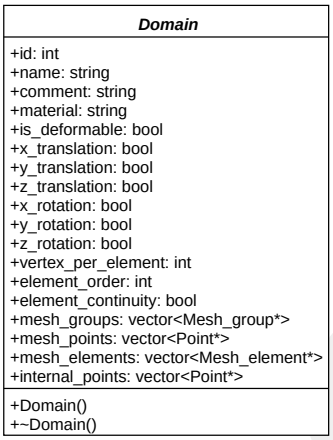


Figure A.5: Detailed view of Domain class.

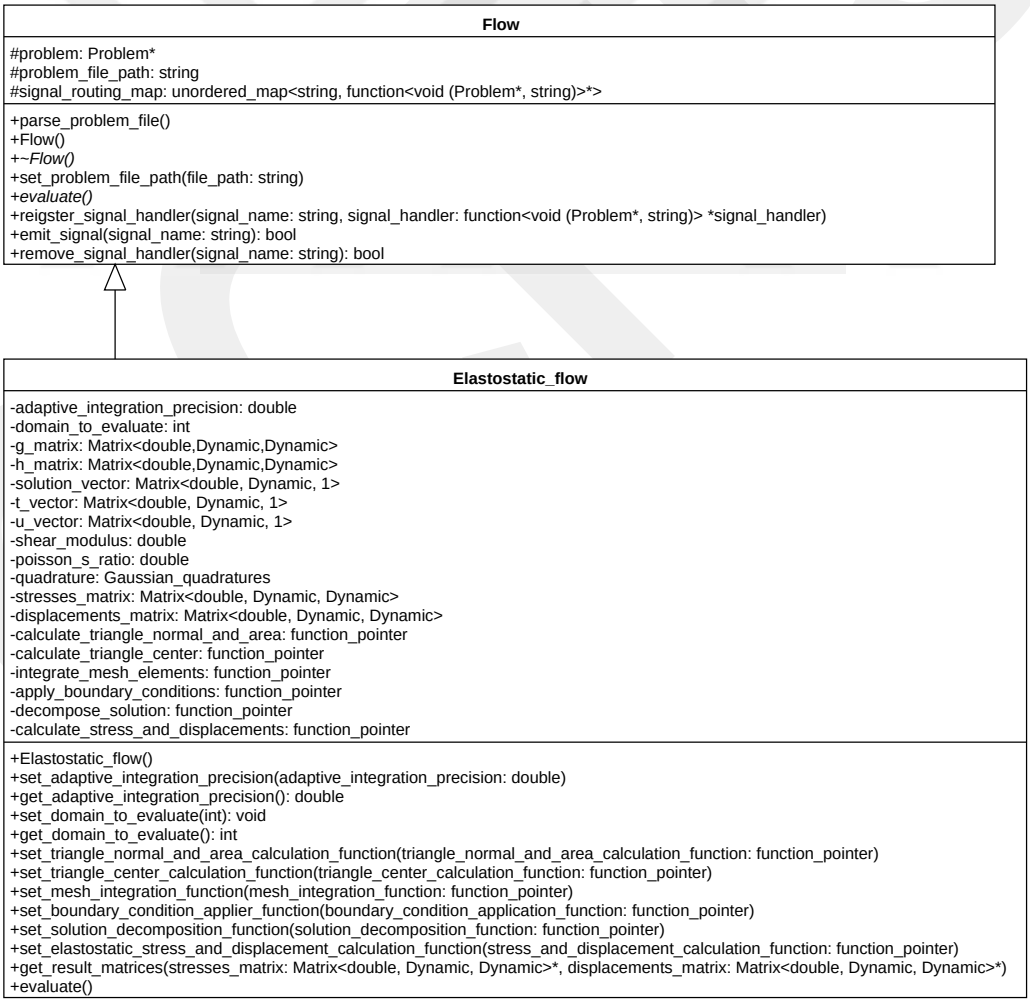


Figure A.6: Detailed view of Elastostatic Flow class.

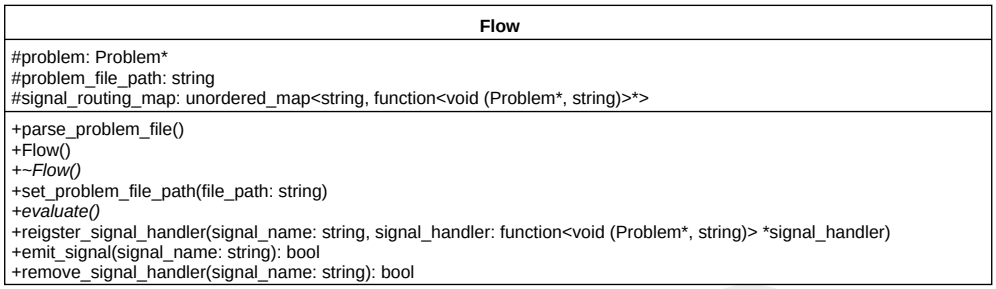


Figure A.7: Detailed view of Flow Core class.

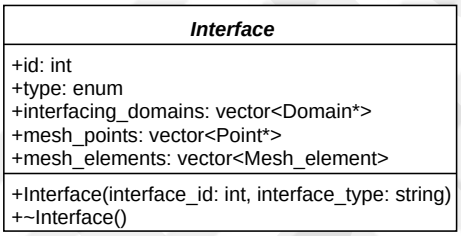


Figure A.8: Detailed view of Interface class.

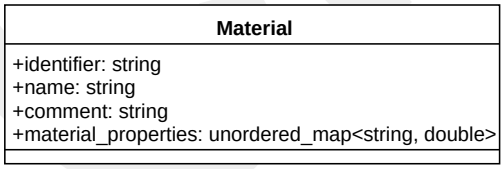


Figure A.9: Detailed view of Material class.

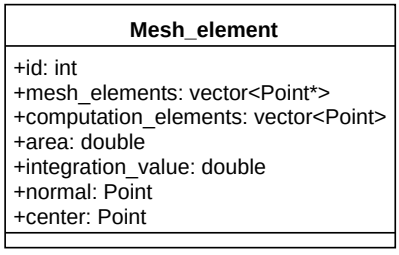


Figure A.10: Detailed view of Mesh Element class.

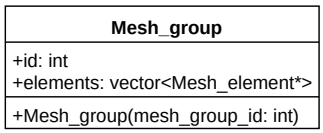


Figure A.11: Detailed view of Mesh Group class.

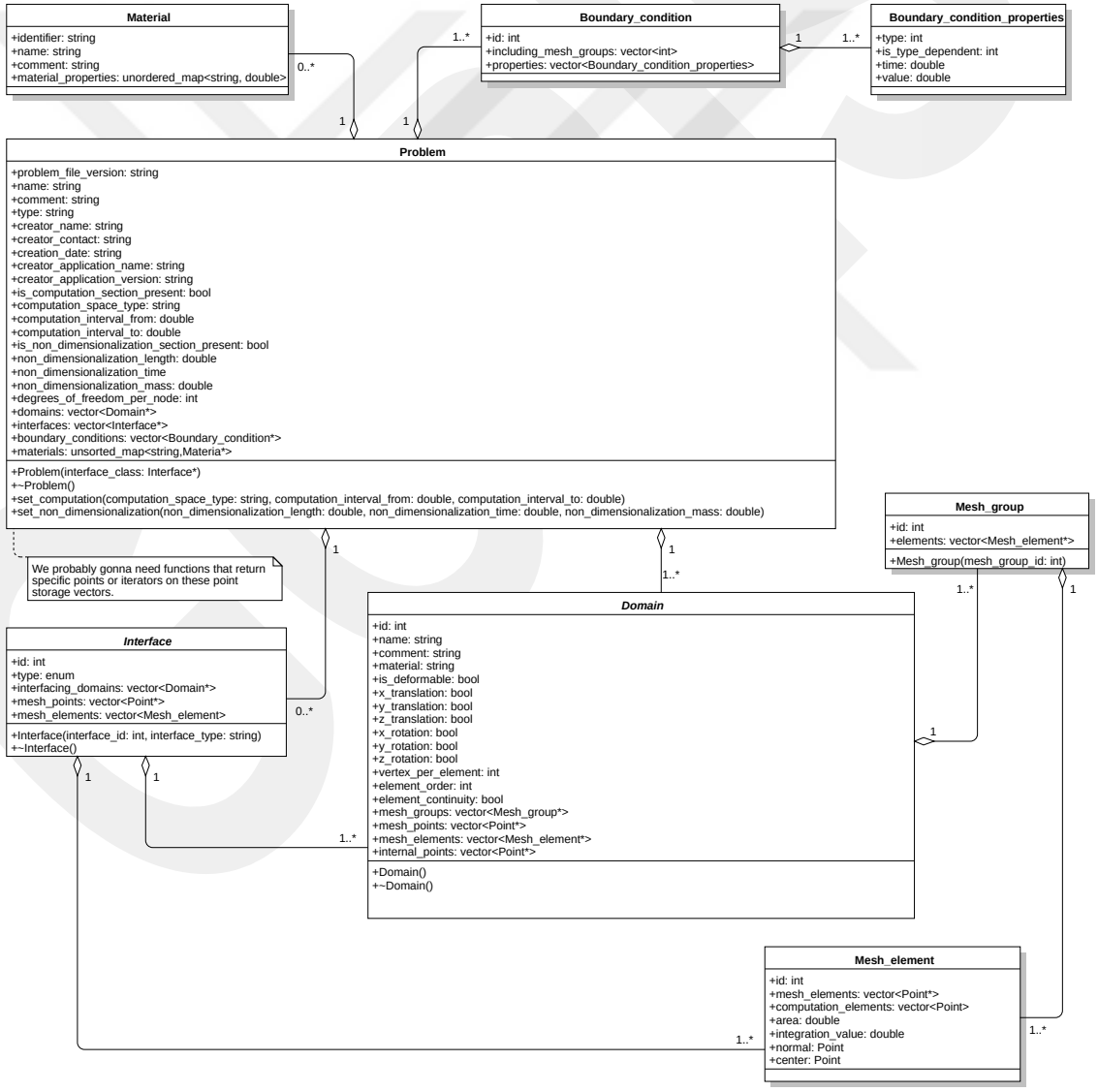


Figure A.12: Overview of Problem class and its relationships.

A.2 NUMA Topology Diagrams of Test Systems

Figure A.13: NUMA topology of the test system named Rigel.

Figure A.14: NUMA topology of the test system named Orion.

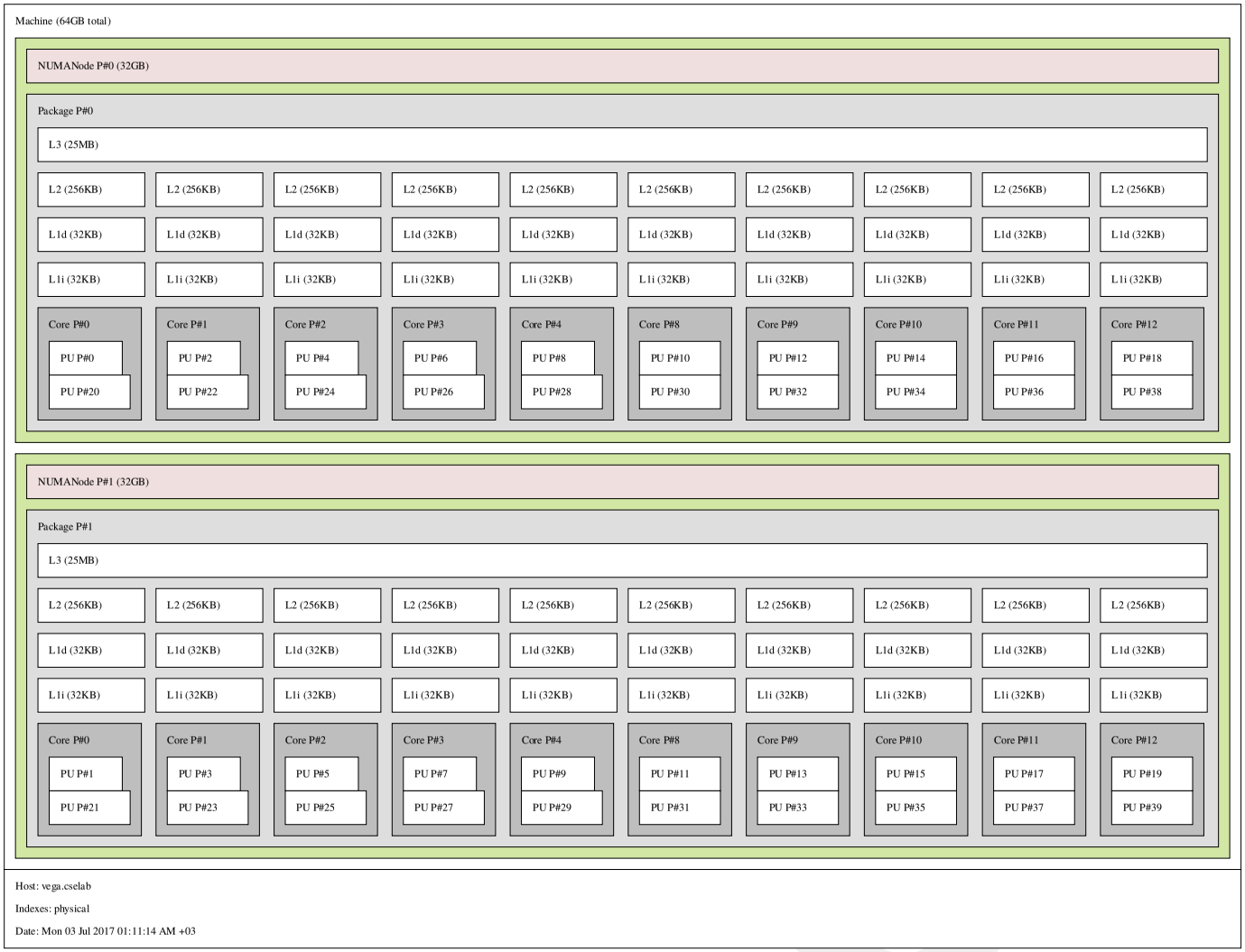


Figure A.15: NUMA topology of the test system named Vega.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Bayındır, Hakan

Nationality: Turkish (TC)

Date and Place of Birth: 05.06.1985, Ankara

Marital Status: Single

Phone: 0 312 2989373

Fax: 0 312 2665181

EDUCATION

Degree	Institution	Year of Graduation
M.S.	Atılım University, Dept. of Computer Engineering	2011
B.S.	Atılım University, Dept. of Computer Engineering	2007
High School	TED Ankara Koleji Özel Lisesi	2002

PROFESSIONAL EXPERIENCE

Enrollment	Place	Year
Researcher	TUBITAK-ULAKBIM, Ankara, Turkey	2006-Cont

PUBLICATIONS

1. Hakan Bayındır, “CRATI: Combinatorial Reverse Auction Trading Infrastructure”, Ms. Thesis, Dept. of Computer Engineering, Atılım Unviersity, June 2007.

SCI & SSCI Papers

1. Bayındır, H., Kılıç, H., & Rehan, M. (2014, December). An agent-based trading infrastructure for combinatorial reverse auctions. In Intelligent Agents (IA), 2014 IEEE Symposium on (pp. 38-44). IEEE.

2. Akman, I., Bayindir, H., Ozleme, S., Akin, Z., & Misra, S. (2011). Lossless text compression technique using syllable based morphology. The International Arab Journal of Information Technology, 8(1), 66-74.