Design and Analysis of Low Frequency Electromagnetic Devices: Exploring and Exploiting Parallelism on Multi-Core Processors

Hussein Moghnieh

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

December 2012

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

To my family and everyone who aspires to contribute to knowledge.

Abstract

The design and analysis of low frequency electromagnetic devices on digital computers using the Finite Element Method (FEM) are computationally expensive and time-consuming. In the past decade, assiduous efforts have been dedicated to exploring and exploiting the parallelism of specific FEM kernels with the aim of enhancing their performance on multi-core processors. Despite having proved advantageous in promoting the speed of specific kernels, how this approach globally impacts the time duration required for the completion of the design process remains unclear, so is the question of whether such an approach takes full advantage of the capacity of multi-core processors. We attempted to overcome these caveats by applying a holistic approach that focuses on the parallel performance of key FEM components, mainly mesh generation, matrix assembly and Preconditioned Conjugate Gradient (PCG).

Our investigation generated three main findings. First, we show that two factors, namely the degrees and exploitation techniques of parallelism differ among the kernels of the FEM solver component using PCG. This fact leads to low thread utilization and a waste of computational resources. Second, contrary to the notion that fine-grained multi-threaded algorithms, albeit involving much synchronization, do not affect the time efficiency of multi-core processors, we show that fine-grained multi-threaded algorithms can in fact be time costly in certain cases involving multi-core processors, particularly so when the ratio of computation runtime to synchronization runtime is relatively small. Third, we found the performance level of algorithms to be inversely related to the size of the problem, suggesting that using domain decomposition techniques to solve smaller problems is likely to pay off.

These findings support the argument that instead of exploring parallelism in single components, a holistic approach focusing on the global design process would be more valuable.

Abrégé

La conception et l'analyse de systèmes électromagnétiques de basses fréquences sur un ordinateur numérique utilisant la méthode des éléments finis (MEF) est d'un point de vue informatique un processus onéreux qui demande du temps. Dans la dernière décennie, des efforts assidus ont été consacrés à l'exploration et l'exploitation du parallélisme des Kernel MEF spécifiques dans le but d'améliorer leur performance sur des processeurs multi-noyaux. Même s'il est avantageux de promouvoir la vitesse de Kernel spécifiques, comment cette approche impacte globalement la durée nécessaire à l'achèvement du processus de conception reste floue, c'est de même pour la question si cette approche profite pleinement des avantages des capacités du processeur multi-noyaux. Nous avons tenté de surmonter ces réserves en appliquant une approche holistique qui concentre sur la performance parallèle des Kernel MEF, surtout, la génération de mailles, l'assemblage matriciel, la multiplication de vecteurs matriciels parsemés et des techniques de préconditionnement fondées sur une factorisation LU inachevée.

Notre enquête a généré trois conclusions principales. Premièrement, nous montrons que deux facteurs, à savoir les degrés et les techniques d'exploitation de parallélisme, différent entre les composantes Kernel du MEF solveur utilisant PCG. Ce fait mène à une faible utilisation du "thread" et à un gaspillage de ressources de calcul. Deuxièmement et contrairement à la notion que les algorithmes multi-thread de granularité fines, bien qu'ils impliquent beaucoup de synchronisation, n'affectent pas l'efficacité du temps de processeurs multi-noyaux, nous montrons que ces algorithmes, peuvent en fait être coûteux en temps dans certain cas impliquant des processeurs multi-noyaux, particulièrement lorsque le rapport d'exécution du calcul de la synchronisation d'exécution est relativement faible. Troisièmement, nous avons trouvé que le niveau de performance des algorithmes était inversement proportionnelle à la taille du

problème, ce qui suggère que l'utilisation des techniques de "domain decomposition" sera advantageux.

Ces résultats appuient l'argument selon lequel au lieu d'explorer le parallélisme dans les composants individuels, une approche holistique centrée sur le processus de conception globale serait plus utile.

Acknowledgments

I would like to thank my thesis advisor Dr. David A. Lowther for his support and patience throughout the whole process of my working on my thesis. His continuous guidance and financial support are what made the completion of this thesis possible.

I also wish to thank my fellow colleagues at the Computational Electromagnetics lab (CEM), David Fernández Becerra, Maryam Mehri Dehnavi, Ali Aghabarati, Evgeny Kirshin and Maryam Golshayan all of whom have been a source of great inspiration for me.

Finally, I am particularly grateful to Malak Abu Shakra, without whom, this thesis would have been rendered a rather impossible mission.

Contents

List of Figures

List of Tables

List of Acronyms

2D	2-dimensional
3D	3-dimensional
AMD	Approximate Minimum Degree Ordering
API	Application Programming Interface
BCSR	Block Compressed Sparse Row
BDC	Brushless Direct Current
BFS	Breadth First Search
BVP	Boundary Value Problem
CAD	Computer-Aided-Design
CCW	Counter-Clock-Wise
CDT	Constrained Delaunay Triangulation
CG	Conjugate Gradient
CMP	Chip Multi-core Processor
COO	Coordinates list sparse storage
CSR	Compressed Sparse Row
DAG	Directed Acyclic Graph
DC	Direct Current
DFS	Depth First Search
DOF	Degrees of Freedom
EM	Electromagnetic
ENIAC	Electronic Numerical Integrator And Computer
EW	Execution Window
FDM	Finite Difference Method
FE	Finite Element
FEA	Finite Element Analysis
FEM	Finite Element Method

FLOP	Floating-Point Operations
FLOPS	Floating-Point Operations per Second
GPU	Graphics Processing Unit
GS	Gauss-Seidel
HYB	Hybrid Sparse Storage Scheme
IC	Incomplete Cholesky
ICCG	Incomplete Cholesky Conjugate Gradient
ILP	Instruction Level Parallelism
ILU	Incomplete LU
I/O	Input/Output
MB	Mega Byte
Mbit	Mega bit
MD	Minimum Degree
MIMD	Multiple-Instruction Multiple-Data
MMD	Multiple Minimum Degree
ND	Nested Dissection
NNZ	Number of Non-Zeros
OS	Operating System
PCDM	Parallel Constrained Delaunay Mesh
PCG	Preconditioned Conjugate Gradient
PDE	Partial Differential Equation
RCM	Reverse Cuthill-McKee
SIMD	Single-Instruction Multiple-Data
SMP	Symmetric Multiprocessors
SMVM	Sparse Matrix-Vector Multiplication
SOR	Successive Over-Relaxation
SW-GS	Sliding-Window Gauss-Seidel

Notations

ϑ	The average number of non-zeros per row of a matrix
$\eta(*)$	The number of non-zeros of $*$, where $*$ represents a matrix or a vector
N	Degrees of freedom of a matrix

Chapter 1

The Design Process Using Computer Aided-Design Software

Contents

1.1 Introduction and Thesis Motivation

Computer-Aided-Design (CAD) software have been widely used by a large number of manufacturing industries and serve as one of the essential components comprising the design process of those industries. In general, a CAD software provides one or more functionalities such as design creation, design modification, design analysis and/or design optimization. The virtualization of these activities on a computer has exerted tremendous impact on the design process, particularly in terms of reducing the time and costs associated with the design and manufacturing processes as well as allowing engineers to achieve better outcomes when dealing with complex designs that would prove rather challenging to elucidate using the traditional methods.

Many CAD software tools are available, with each designed for the purpose of performing analyses in specific domains (i.e. electromagnetic analysis, structural analysis and fluid dynamics analysis) or sub-domains (i.e. low frequency electromagnetic device analysis). Nowadays, a number of CAD systems include a set of tools that are combined together to provide the ability to draw, modify, analyze and optimize a model. Recent advances enabled the incorporation of many physics disciplines, known as multi-physics field solver. The analysis stage, which serves as the core of a CAD system and the design process, has been shown to be time consuming and has been therefore undergoing increasing investigation and research. In this thesis, the analysis stage is often referred to as “field solver”, “electromagnetic (EM) field analysis” or “Finite Element Analysis” (FEA).

The ability of CAD software to infiltrate the design process and to fulfill some of the functionalities that would otherwise require massive efforts from engineers can be partially but not exclusively attributed to the progress in increasing the computational power of digital computers since their creation in 1945. The constellation of factors involved in the computer eco-system, such as the costs associated with software and hardware, the variability of prices between hardware components, computational models (batch processing, time sharing, desktop computer) and mainly the market using the computer (i.e. business market) all represent driving forces that have shaped the computer industry and, as a result, the CAD software industry as well.

The introduction of the multi-core processor has been perceived by domain experts as changing the landscape of computing [1]. Massive research has been geared towards investigating parallelism in methods and kernels that are currently used and implemented in CAD software. Most currently used parallelization and optimization strategies mainly involve the tuning of algorithms and modification of problem structures to fit into a particular processor architecture (i.e. general purpose multi-core processors, GPU and IBM Cell Broadband). This approach of fine-tuning kernels suffers from many issues:

- First, the set of problems used as testing cases to report the runtime performance of an algorithm, be it sequential or be it parallel, is not representative of the typical problems arising in a real life EM design process in terms of size and structure. Tuning algorithms is largely dependent upon those parameters (i.e. problem size and structure), rendering their application on relevant problems rather essential.

- Second, it remains unclear which stage of the analysis exerts the greatest impact on CAD runtime. Time and storage complexity and the degree of parallelism are all indicators of the impact exerted by a kernel on the analysis' runtime. However, efficient implementation and exploiting parallelism remain challenging to achieve.

- Finally, a large amount of research has been conducted with the aim of speeding up the analysis stage of a design process given that is the most time consuming step. The problem with this approach is that it uses all the resources of a desktop computer to accomplish one task at a time. This computational model is akin to that of batch computing in which a user - or a design engineer in this context - waits for an analysis step to finish before proceeding to the next step in a design process. A more suitable computational model is the one that allows an engineer to run, monitor and interact with multiple simulations simultaneously. This computational model better maps a design engineer's approach to a design process onto a desktop computer.

1.2 Thesis Scope and Outline

This thesis presents an overview of the numerical techniques used when designing and analyzing low frequency electromagnetic devices on a desktop computer equipped with a multi-core processor. Of particular interest is the *Finite Element Method* (FEM) since it has been widely used in this particular domain for its robustness in dealing accurately with a large set of problems arising in this field. The parallel performance of iterative solvers such as Conjugate Gradient (CG) and Gauss-Seidel (GS) in addition to preconditioning techniques, sparse matrix-vector multiplication (SMVM), mesh generation and matrix assembly are investigated. The aim is not to fine tune these algorithms for a specific processor architecture, but rather we are interested in understanding their fundamental building blocks, algorithmic time and storage complexity and their scalability using fairly efficient sequential and parallel implementation.

The investigation of each stage of the design process serves to illustrate the bottleneck in each stage and the problems faced exploring and exploiting its parallelism. The long term goal of this work is to show that other types of parallelism should be explored beyond that of fine-grained parallelism, which has been a goal of much research effort aiming at maximizing the utilization of multi-core processors. One such type of parallelism is to use coarse-grained parallelism (or multi-tasking) on a multi-core processor. In this type of parallelism, a design engineer explores simultaneously multiple designs, or investigates multiple variations of the same design, or performs multiple design analyses as part of the optimization process.

Experimental Test-Bed All codes used in this thesis work have been written in C++, compiled using GNU g++ version 4.4.3, and executed on Ubuntu 10.04 (x86_64 Linux kernel version 2.6.32-28).

The experiments have been executed on two generations of quad-core processors that fundamentally differ in terms of architecture and performance. The first processor is the Intel i7-860 processor (code name *Lynnfield*) which contains 4 cores (based on Intel's *Nehalem* micro-architecture) clocked at 2.8 GHz. Each core has a 32 KB private Level-1 (L1) data cache, 256 KB private Level-2 (L2) cache and all cores shares an 8 MB Level-3 (L3) cache. Furthermore, this processor uses Intel's *Hyper-Threading* technology where certain parts of a processor are duplicated to store more thread state resources beyond the available physical resources. It makes the processor logically appear to the OS as having more available threads (i.e. 8 threads instead of 4 threads). The processor is connected to the DDR3/1333 memory sub-system allowing a maximum theoretical bandwidth of 21 GB/s.

The second processor is the dual-socket dual-core AMD Opteron 2214 processor which is an older generation of multi-core system where each core is clocked at 2.2 GHz, has a 64 KB private Level-1 cache and a 2 MB private Level-2 cache. Each socket has its own memory controller to the shared DDR2 memory at a theoretical bandwidth/core of 10.6 GB/s (aggregate of 21.2 GB/s for the dual socket system). Cores from different sockets communicate using AMD's *HyperTransport* technology.

The remainder of this chapter (§ 1.4 - § 1.6) briefly introduces the *Finite Element Method*, gives an overview of the design process and also a brief history of the evolution of the digital computer and its impact in shaping CAD software. Chapter 2 investigates the time and storage complexity of mesh generation. Chapter 3 examines implementation issues in the matrix assembly stage in FEM and also serves to generate the matrix test sets used in the remaining part of the thesis. Chapters 4, 5, and 6 target the solver stage in CAD. Particularly, Chapter 4 experiments with a newly developed variation of Gauss-Seidel based solver while Chapters 5 and 6 investigate the performance of parallel sparse matrix-vector multiplication (SMVM) and parallel preconditioning techniques respectively since they constitute the basic building blocks of the state-of-the-art Conjugate Gradient (CG) based solvers. Finally, Chapter 7 lists the contributions of this work and provides a brief conclusion and future work suggestions.

1.3 Contributions

The contributions of this thesis to the field of electromagnetic device design on a desktop computer can be summarized as follows:

- We investigated the characteristics of sequential and multi-threaded algorithms of FEM, in terms of parallel granularity (e.g. fine- and coarse-grained granularity), synchronization overheads and cache performance, independently of a specific multi-core architecture.

- A holistic approach globally investigating FEM "processing" stage using first, a single matrix storage structure and second, matrix test sets relevant to typical problems in terms of size and structure. This investigation generated three main findings:

  - Two factors, namely the degrees and exploitation techniques of parallelism differ among the kernels of the FEM solver component using PCG. This fact leads to low thread utilization and a waste of computational resources.
  - Although fine-grained multi-threaded algorithms enhance locality of data reference, we show that in cases where the ratio of the computation runtime to the synchronization runtime is relatively small, fine-grained parallelism proves inefficient in that it generates synchronization overheads that prevent the reduction of the final runtime.
  - The performance level of algorithms to be inversely related to the size of the problem, suggesting that using domain decomposition techniques to solve smaller problems is likely to pay off.

- We developed a new parallel iterative solver based on a combination of Gauss-Seidel and Jacobi (i.e. Sliding Windows Gauss-Seidel). We show that despite having not proven advantageous in reducing the time duration required to solve

$Ax = b$, our method reduced the cache misses occurring upon accessing matrix A.

1.4 Computer-Aided Design Analysis Using the Finite Element Method

In general, analyzing a design using CAD involves the following three stages: pre-processing, processing and post-processing (Figure 1.1). The pre-processing stage is considered the most critical for a successful analysis during which the design engineer creates a simplified and computationally feasible model of the physical design. For instance, the designer may choose at the beginning to work with a 2-dimensional design or analyze only part of the device while assuming the effect of the other parts be known. Once the real model is projected into an artificial simplified model, its geometry is plotted using a CAD drawing tool, material properties are set and a suitable mathematical representation is used to model the simplified design. Next, the processing stage consists of the actual steps involved in the *Finite Element Method*, the goal of which is solving the governing partial differential equations over the input geometry and finally during the post-processing stage, the results obtained in the previous step are inversely mapped onto the simplified model so as to present a meaningful set of results to the design engineer.

1.5 Maxwell's Equations

The time dependent Maxwell's equations in their differential form are given by

$$\nabla \times H = J + \frac{\partial D}{\partial t} \tag{1.1}$$

$$\nabla . B = 0 \tag{1.2}$$

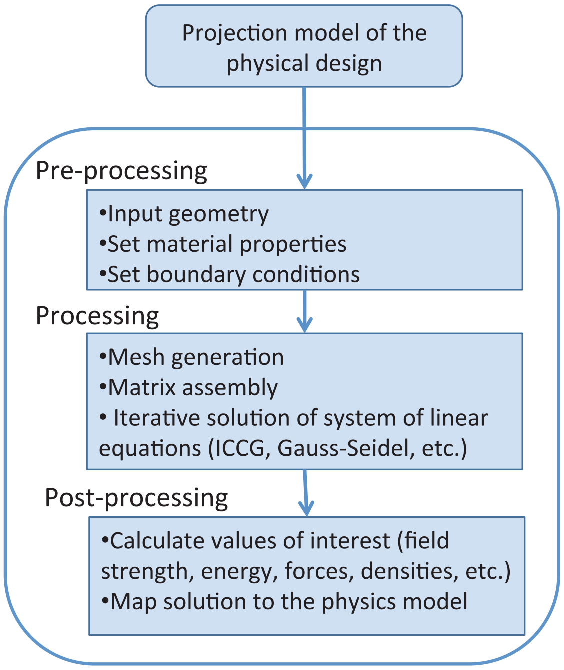

Projection model of the physical design
Pre-processing
•Input geometry
•Set material properties
•Set boundary conditions
Processing
•Mesh generation
•Matrix assembly
• Iterative solution of system of linear equations (ICCG, Gauss-Seidel, etc.)
Post-processing
•Calculate values of interest (field strength, energy, forces, densities, etc.)
•Map solution to the physics model

Figure 1.1: **Stages of a design analysis using CAD.**

$$\nabla \times E = -\frac{\partial B}{\partial t} \tag{1.3}$$

$$\nabla . D = \rho \tag{1.4}$$

Where E is the electric field, H is the magnetic field, J is the transport current density, D the displacement field, B is the magnetic induction field or flux density, ρ is the free charge density and t is time.

D, E, B, and J are related by the following equations:

$$D = \epsilon E \tag{1.5}$$

$$B = \mu H \tag{1.6}$$

$$J = \sigma E \tag{1.7}$$

where ϵ, μ and σ are respectively the permittivity, permeability and conductivity of the material.

When working with electrostatic or magnetostatic problems (i.e. non time dependent), Maxwell's Equations 1.1 and 1.3 reduce respectively to the following:

$$\nabla \times H = J \tag{1.8}$$

$$\nabla \times E = 0 \tag{1.9}$$

Replacing D in Equation 1.4 by Equation 1.5 gives

$$\nabla.(\epsilon E) = \rho \tag{1.10}$$

When solving for the electric potential φ in electrostatic problems (non time dependent), Equation 1.10 reduces to Poisson's equation of the form:

$$-\nabla.(\varepsilon \nabla \varphi) = \rho \tag{1.11}$$

where

$$E = -\nabla \varphi \tag{1.12}$$

(φ is called the *electric scalar potential*).

When solving for the magnetostatic field, the magnetic flux density B is first represented using

$$B = \nabla \times A \tag{1.13}$$

where A is called the *magnetic vector potential.* Subsequently, Equation 1.6 ($B = \mu H$) can be rewritten as

$$\nabla \times A = \mu H \Longleftrightarrow H = \frac{1}{\mu} \nabla \times A \tag{1.14}$$

Finally, replacing Equation 1.14 in Maxwell's Equation 1.8 results in the magnetostatic equation to be solved of the form:

$$\nabla \times (\frac{1}{\mu} \nabla \times A) = J \tag{1.15}$$

The solution of these differential equations, Equation 1.11 for electrostatic problems and Equation 1.15 for magnetostatic problems, usually involves imposing a boundary condition, which is equivalent to setting a known solution on the boundary of the geometric domain or setting the values as constraints; these problems are commonly referred to as "boundary value problems" (BVP). The analytical solution of 1.11 or 1.15 is not possible for the majority of boundary conditions. Instead, there are many numerical techniques that can solve these second order elliptic differential equations [2]. The focus of this thesis will be on the *Finite Element Method*, which has been used with a great success in the design of low frequency electromagnetic analysis because of its ability to deal with complex geometric shapes.

The solution of partial differential equations over a domain using the *Finite Element Method* usually involves the following steps.

1. Discretize the domain using finite elements; triangular elements in 2D and tetrahedral elements in 3D are usually utilized when analyzing low frequency electromagnetic devices.

2. Choose appropriate interpolation functions (otherwise known as shape functions or basis functions).

3. Obtain the corresponding linear equations for a single element by first deriving the weak formulation of the differential equation subject to a set of boundary conditions.

4. Form the global matrix system of equations through the assembly of all the elements.

5. Impose boundary conditions (Dirichlet, Neumann and Cauchy).

6. Solve the linear system of equations using linear algebra techniques.

7. Calculate and visualize the values of interest from the solution obtained in 6.

Most research, since the introduction of the multi-core processor, has focused on speeding up the computationally costly steps of the *Finite Element Method* by using multi-threaded algorithms. However, the steps shown above are usually executed many times within a larger context, that is a design process. Hence, the next section will overview the design process and will also briefly review the impact of the progress of the digital computer on the design process and an engineer's work.

1.6 Problem Context and History

1.6.1 The Design Process

There are many definitions of the design process, where each reflects its author's view on the process based on observation or experience. For instance, in a general context design can be viewed as:

> *...the creative process which starts from a requirement and defines a contrivance or system and the methods of its realization or implementation, so as to satisfy the requirement. It is a primary human activity and is central to engineering and the applied arts* [3].

In the context of engineering design,

> *...it is the use of scientific principles, technical information and imagination in the definition of a mechanical structure, machine or system to perform pre-specified functions with the maximum economy and efficiency* [4].

Those definitions do no accurately describe the commonly used approach when designing low frequency electromagnetic devices, however. In general, a design process can be defined in terms of the activities that are usually implicitly present in it; the authors in [5] defined it to be an iterative, incremental, exploratory, investigative, creative, rational (logic based), decision making and interactive process.

A design model depicts the designer's or scientist's approach to a design problem. A classification of these models leads to what is referred to as design strategy which can be, for example, problem- or solution-oriented [6]. In the problem-oriented approach emphasis is placed upon abstraction and thorough analysis of the problem's structure before generating a range of possible solutions. In the solution-oriented approach, an initial solution is proposed (e.g. by relying on past experience or past solutions), analyzed and then repeatedly modified as the design space and requirements are explored together. It has been identified that both strategies are usually used within the same design process. For instance, in the preliminary steps of a design process, the problem-oriented approach is used. The set of initial possible solutions are usually modeled so that they have less complexity than the real design. It is common to use 2-dimensional analysis where many design details and effects are ignored. Since the preliminary designs are less complex and are less computationally intensive than a full 3-dimensional analysis, it is possible to execute their analysis simultaneously while sharing many design properties and features, such as the material properties or a part of the geometry.

1.6.2 The Design Process of Low Frequency EM Devices

The design process of low frequency electromagnetic (EM) devices usually encompasses the following: 1) the transformation of the requirements into performance and functional specifications, 2) the elaboration of cost, resource and other critical constraints, 3) the mapping and converting of specifications into feasible design solutions, 4) the analysis and optimization of the solution [7]. Designing a set of potential solutions (step 3) and the analysis of each of them (step 4) are the most time consuming engineering design tasks.

The development of a design solution has several distinct phases. Four of these phases are problem structuring, preliminary design, refinement and detailing. Preliminary design is a classical case of a creative, ill-structured problem where alternatives are generated and explored. Refinement and detailing are more structured and the steps to be performed can better be described [8]. For instance, in designing low frequency electromagnetic devices, the solution search space is first explored in the preliminary stage by investigating many different designs that are thought to meet requirements and constraints. At this phase, many details of the design are eliminated and usually a 2-dimensional (2D) analysis is sufficient. In contrast, detailed design is performed using 3-dimensional (3D) analysis where a more accurate electromagnetic field calculation is required. The requirements of these 2 phases (i.e. the preliminary analysis using a 2D model and the detailed analysis using a 3D model) pose different computational requirements. The first (i.e. preliminary design) is a highly multi-tasking activity; the latter (i.e. detailed design) is more computationally intensive and time consuming. In a simplistic form, this design process can be depicted as in Figure 1.2. In this approach the design engineer analyses and re-iterates over a design (i.e. a prototype) until the sought device meets requirements and constraints. This process is repeated many times, whether by changing the computational model for the same design, exploring other design solutions, or changing parameters for a specific design.

In the era of desktop computers equipped with a single core processor, it was natural to think of the iterative, multi-phase design process as a being a sequence of ordered stages where the field simulation (e.g. finite element analysis) is the most time consuming stage. Therefore, the focus of research and companies working on finite element analysis tools has been to speedup the analysis stage by devising numerical techniques which efficiently map onto desktop computers, thereby reducing the design waiting time. Implementing sequential algorithms using the Von Neumann programming model on single core microprocessors has been convenient for developing scalable and usable CAD software. With every new microprocessor and larger faster memory, the same CAD system is able to solve more complex problems.

On a multi-core processor, designing parallel algorithms has been the main chal-

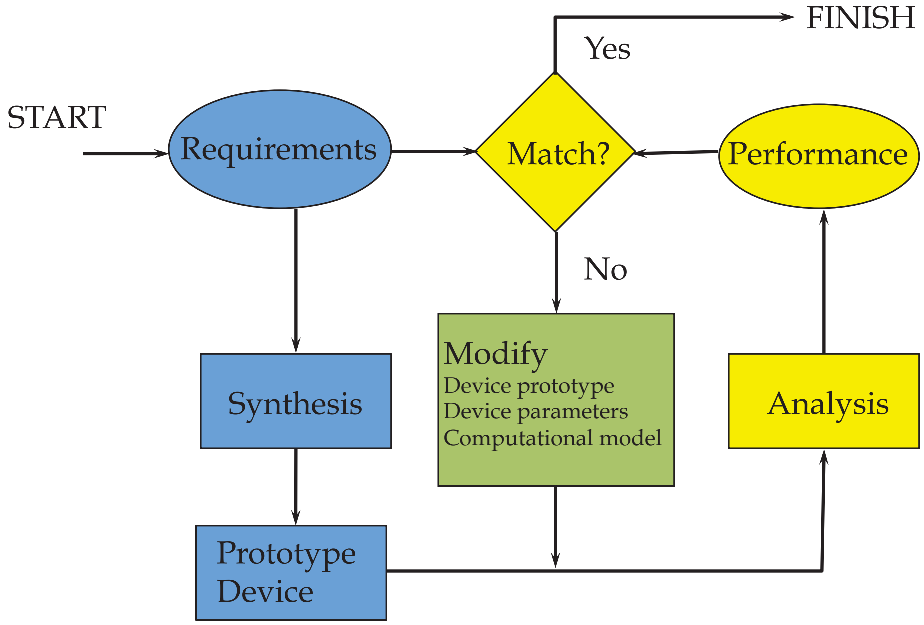


Figure 1.2: **A simplistic form of a design process.**

lenge for software development in general and for CAD software development in particular. First, the sequential algorithms that have been developed in the past to run efficiently on a single core microprocessor must be revisited to explore their amenability to parallelism. Second, even if parallelism can be explored in these algorithms, ideally we want them to scale as the number of processors/cores increases. Third, the performance of parallel algorithms is highly dependent on the multi-core architecture and its programming model, in addition to computer parameters such as the bus speed and the memory subsystem performance. Most of the research in design analysis exploits every new architecture by fine tuning already existing algorithms. There is little understanding of how these algorithms map onto current multi-core systems or what is their expected performance when the architecture changes. Furthermore, even if linear speedup is being attained on some algorithms or methods, the effectiveness of such work in reducing the overall time for the design process is not clear.

The gain in a specific kernel or method of the design analysis, although it might seem beneficial in its own context, might not reflect a great gain in the overall time of the design process. This means, while efforts have been focusing on optimizing a parallel algorithm on a specific architecture, other potentially more serious bottlenecks could be left. This problem, while not apparent on a processor with a few cores, could be a bottleneck as the core number increases and other system properties change, such as cache speed, bus speed, etc.

1.6.3 Design Analysis: From Batch Computing to Desktop Computer

Many physical phenomena in science and engineering, such as the analysis of electromagnetic devices, can be modeled by partial differential equations (PDEs). The use of computational tools to solve partial differential equations has been of interest for engineers even before the invention of the digital computer in early 1940's (i.e. Colossus and the ENIAC[1]). It was common to use analog computers such as the differential analyzer in the 1930's - even until well into the 1970's - for the solution of differential equations arising in the field of electrical engineering [9] or the use of a mechanical computer which uses springs, gears and other devices to solve structural problems [10]. The differential analyzer was an electromechanical device which was used by the military to solve the equations which for the motion of a projectile were readily adaptable to these machines. However, the need for faster and more accurate calculations compelled them to fund the creation of a digital computer [11]. During the 1950's and soon after the introduction of the digital computer, the numerical techniques field of study became popular and ultimately led to the introduction of the *Finite Element Method* [12, 13]. It is worth noting that during the early days of numerical computations, developed methods used more computations than storage; this is in contrast to the techniques that had been used before the invention of the digital computer (i.e. the differential analyzer) [13].

The ENIAC was created in 1945 to perform mathematical computations faster than previously available analog computers by taking advantage of electronics technology

[1] ENIAC: Electronic Numerical Integrator And Computer.

(vacuum tubes). It was not a memory based stored program computer, and setting it up for a new job involved reconfiguring the machine by means of plugging and switching [14]. In 1949, UNIVAC I (UNIVersal Automatic Computer I) was created, setting the principles of an instruction based computer. Instructions were saved in memory, rather than being implemented through hard wiring. A few years afterwards, Von Neumann demonstrated the usefulness and generality of these computers in solving many problems, such as weather prediction, by devising the mathematical model of the phenomenon [15].

A typical process for running finite element analysis on these computers was for the design engineer to manually draw and mesh a model. Without graphics input/output terminals, this step was very error-prone. A coding form with the applicable data generated from this step would subsequently be given to a keypunch operator who produced a deck of punched cards to be submitted for execution in a batch mode. Upon completion, the design engineer would use the potential solution produced by the computer to manually calculate the fields of interest and plot the data. It was obvious that an end to end solution was needed for this operation to be appealing for business aimed at saving man hours in designs, and this includes a better computational model (i.e. time sharing) and graphics input/output terminals [10]. Until a better system was in place, the finite element analysis using a digital computer remained a domain of research in labs and universities (i.e. General Electric, McGill University, etc).

A key hardware development was the introduction of the IBM System 360 product line in 1964 which included the Model 2250 refresh graphics terminals. Computers based on this architecture were called mainframes and were used for data intensive input/output processing. The IBM 360 architecture was successful since it had used microcode in its processor technology leading to programs written for subsequent IBM 360 based architecture computers being backward compatible [16]. Many CAD software tools emerged in this era and it was customary for engineering companies to lease mainframes to perform analysis. There were many remaining issues in this computational model. First, the price to lease a CAD seat was still expensive. Second, CAD companies had to develop their own code to deal with graphics input/output.

To further reduce the costs incurred by using mainframes, minicomputers started to emerge in the late 1960's and early 1970's. These computers were cheap enough to be owned and hosted by engineering firms. Furthermore, they used a time sharing computational model, which is more effective for problem-solving situations than the batch computing model [17]. In this model users could simultaneously access the computer via terminals. Meanwhile (i.e. after the introduction of the minicomputer), operating systems (OS) included more functionality, such as open GL, X-Windows and MOTIF, to deal with graphical Input/Output (I/O). Incorporating pre-processing and post-processing in finite element analysis tools and the move from batch processing to a time sharing computation model,in addition to decoupling the hardware from the software by incorporating more functionalities from vendor software into the OS, were all key points in turning CAD systems into usable and essential tools used in the design process of companies [18, 19]. Furthermore, the decreasing price of storage relative to processing price had changed the way algorithms were written. CAD companies which did not cope with the changes that had been occurring since the invention of the first digital computer did not survive [10].

The introduction of the microprocessor, the Intel 8008 in 1971 and later the Intel 8080 in 1975, was a turning point in the computer industry. Personal computers or desktop computers were widely available, minimizing the need for minicomputers as the microprocessor technology progressed. The implications of the desktop computer on the CAD industry are notable in its ability to provide high user-computer interaction at an affordable price. It released engineering firms and CAD software firms from the high cost of previous generation computers due to its decentralized computational model.

The discussion so far has emphasized the importance of the computational model (time sharing), the use of graphics terminals for input/output and the dropping price of computers as determining factors in shaping CAD software. This has served as a compelling reason to investigate the bottlenecks in current techniques used to leverage the use of desktop computers equipped with multi-core processors in order to show that there are many factors that support the need of a CAD software that work beyond the traditional batch computing paradigm.

Chapter 2

Finite Element Method Mesh Generation

Contents

2.1 Introduction

The “processing” stage of the field analysis of an EM device using the *Finite Element Method* has attracted much attention due to its computationally intensive nature. It includes mesh generation, matrix assembly and the solution of linear equations. Mesh generation serves as the focus of this chapter, which begins by reviewing a state-of-the-art technique used to generate a quality mesh (i.e. Delaunay mesh generation using incremental point insertion) by explaining each kernel involved in the sequential mesh generation using this technique and the computational complexity associated with each of them. Next, an optimized mesh generator developed by Shewchuk, called *Triangle* [20], will be analyzed using Intel’s Vtune Analyzer [21] to investigate its cache performance and the number of processor cycles spent in each kernel. Finally, a discussion on parallel mesh generation techniques will be presented.

2.2 2D Mesh Generation

Mesh generation refers to the technique utilized to discretize the geometric domain Ω into non overlapping elements. In a 2D low frequency EM analysis, triangular shapes are commonly, but not exclusively, used as the basic element, particularly when irregular regions or domains are at play. Other element shapes are also possible (e.g. rectangles). This thesis will refer to the vertices and sides of triangles as nodes and edges, respectively. Figure 2.1 shows an example of an initial 2D mesh generation of a brushless direct-current (DC) motor model using triangular elements. The triangles share edges and nodes such that $\Omega = \bigcup_{i=1}^{ne} E_i$, where E denotes an element, and ne denotes the number of elements.

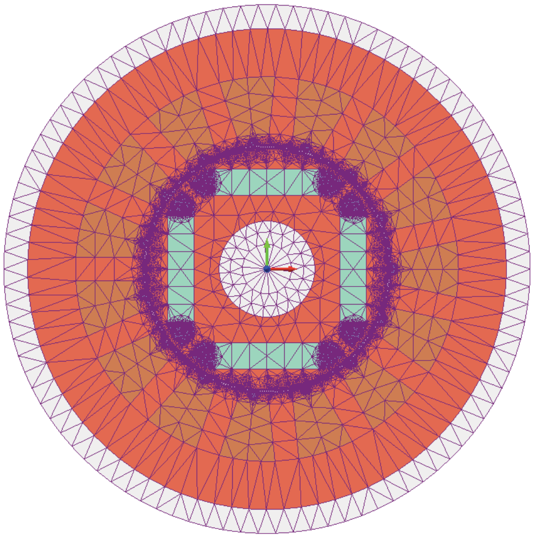


Figure 2.1: **Initial 2D mesh of a brushless DC (BDC) motor.** The model was obtained from the examples set provided by MAGNET (2D/3D electromagnetic field simulation software by INFOLYTICA, www.infolytica.com) [22].

The discretization of a domain is a specific case of a more general problem encountered in computational geometry, i.e. that of triangulation. Given a set of vertices $P = p_1, p_2, ..., p_n$, the triangulation $T(P)$ denotes the set of non overlapping triangles, with corners on the input vertices, such that all vertices are covered. There can be exponentially many triangulations that can be obtained from a point set P with widely varying appearance. Mesh generation, on the other hand, is a triangulation problem where quality measures are imposed on the shape of triangles, size of angles, edge lengths and area (size) of a triangle, rendering it a rather challenging problem. Of all domains utilizing meshing techniques (e.g. graphics, solid modeling), generating meshes for numerical techniques such as FEM is perhaps the most challenging given the stringent requirements it imposes on the mesh quality [23].

2.3 Delaunay Triangulations

The Delaunay triangulation $DT(P)$, introduced by Delaunay [24] in 1934, of a set of vertices $P = p_1, p_2, ..., p_n$ is a triangulation $T(P)$ with a set of criteria imposed on the quality of the triangles. One such criterion relates to maximizing the minimum angle of every triangle, which has proven to be quite advantageous in terms of working on meshes. In order to impose such a quality measure on triangulating a point set P, the *"inCircle"* property, as illustrated in Equation 2.1, has to be preserved for every triangle; each circumscribing circle of a triangle's vertices should not contain a vertex from another triangle. If $inCircle(a, b, c, p) < 0$ then p would fall within the circumscribing circle of the triangle $\triangle(a, b, c)$, thereby violating the Delaunay criterion. Figure 2.2 illustrates a Delaunay triangulation of 5 vertices.

$$inCircle(a, b, c, P) = \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ p_x & p_y & p_x^2 + p_y^2 & 1 \end{pmatrix} > 0 \qquad (2.1)$$

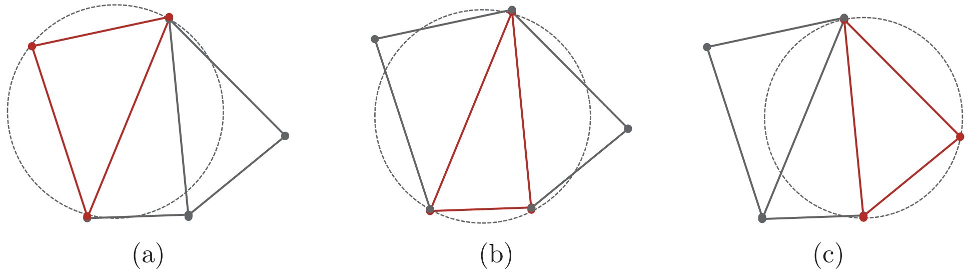


Figure 2.2: **Initial Delaunay triangulation:** Each triangle complies with the Delaunay criterion. The circumscribing circle of each triangle does not contain a vertex from another triangle.

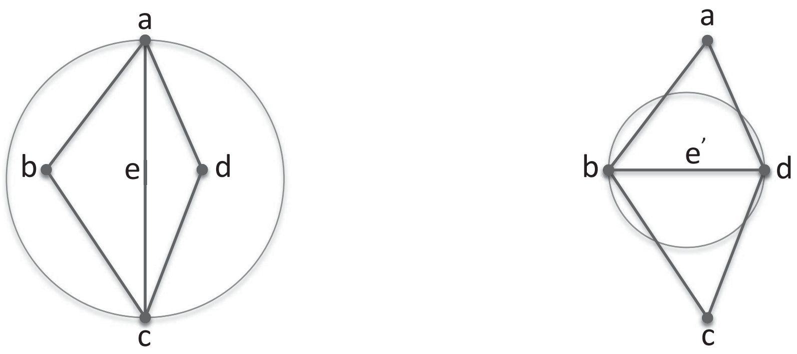
Many techniques to create a Delaunay triangulation from a set of points are available, mainly the divide-and-conquer [25], plane-sweep [26] and incremental point insertion [27, 28, 29] ones. All these algorithms have been proved to have a time complexity of $O(n \log n)$ [30]. Shewchuk [20] has compared efficient implementations of these techniques and found that the divide-and-conquer to be superior to all on the DEC 3000/700 with a 225MHz Alpha processor. The performance of incremental point insertion is poor, as most of its time is dedicated to "point location". Nevertheless, incremental point insertion algorithms are advantageous in that they are robust and highly amenable to mathematical proof (algorithm termination and guaranteed mesh quality).

2.4 Quality Mesh Generation using Incremental Point Insertion

Incremental point insertion algorithms start with performing a simple triangulation of a set of p points and continue to insert points and restructure the triangles following each insertion to maintain the Delaunay properties. In order to attain a quality mesh using the incremental point insertion technique, two primary steps are crucial. The first step involves creating an initial mesh using a pre-defined set of input points and edges. The input usually defines the boundary and the shape of the object being meshed in addition to defining additional points within that object. The second major step is termed the *Delaunay Mesh Refinement* and involves adding additional points, referred to as *Steiner* points, into the triangulation so that the final mesh meets to user's requirements in terms of triangle sizes (areas) and triangle quality (minimum angle), while at the same time maintaining the Delaunay properties. The resulting mesh typically contains n nodes, where $n \gg p$. In both steps, inserting a point requires a restructuring of the mesh to maintain Delaunay criterion as illustrated in the next section.

2.4.1 Structural Update

The structural update is defined as the set of operations aimed at re-triangulating part of the mesh to preserve the Delaunay criterion following the insertion of a given point. This can be performed using either Lawson's [27] edge flip restructuring algorithm or Bowyer/Watson's [28, 29] cavity restructuring algorithm.



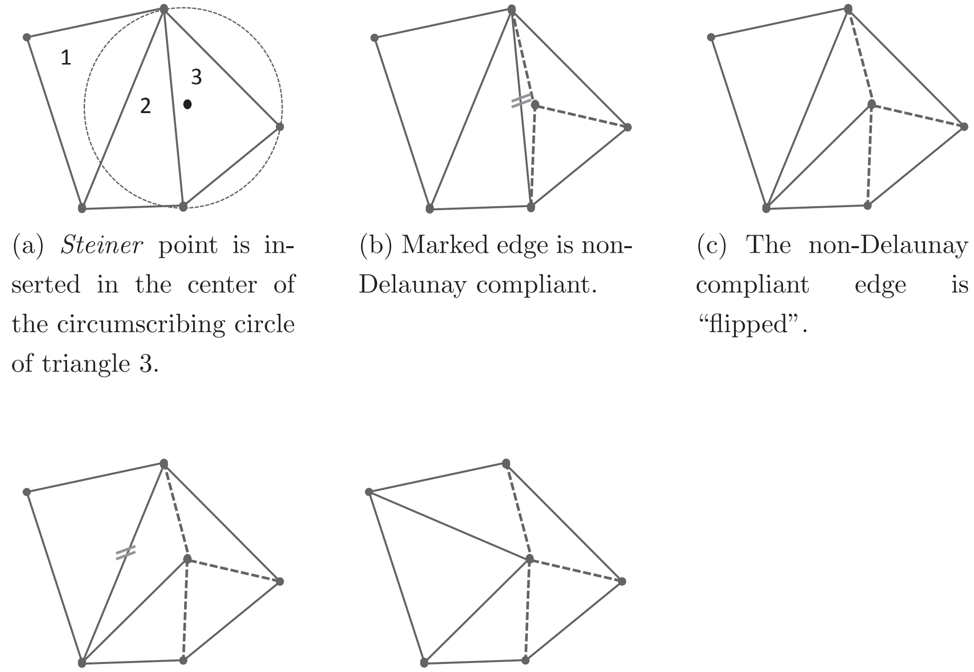
(a) Edge e non Delaunay compliant

(b) Edge e is flipped to yield e' which is a Delaunay compliant edge

Figure 2.3: **Edge flip algorithm:** (a) If the circumscribing circle of a common edge e between 2 triangles (Δ_{abc} and Δ_{acd}) contained a vertex of the mesh, then the edge would not comply to Delaunay. (b) Eliminating this edge and subsequently connecting vertices that are not yet connected to the quadrilateral formed by the triangles Δ_{abc} and Δ_{acd} would yield a Delaunay compliant triangulation.

Lawson's "edge flip" functions by "flipping" edges that are non Delaunay compliant as illustrated in Figure 2.3. In (a), the common edge e between the two triangles, Δ_{abc}

and Δ_{acd}, is non-Delaunay compliant since its circumscribing circle contains a vertex. As shown in Figure 2.3b, eliminating this edge and subsequently connecting vertices that are not yet connected of the quadrilateral formed by the triangles Δ_{abc} and Δ_{acd} would yield a Delaunay compliant triangulation. Algorithms utilizing this technique to re-triangulate meshes follow a point's insertion function in the following manner: upon the insertion of any point, a depth-first-search (DFS) or a breadth-first-search (BFS) is conducted on all of the surrounding triangles and edges that are non-Delaunay compliant are "flipped". For instance, given the Delaunay triangulation previously shown in Figure 2.2, a point that is inserted into triangle number 3 is connected to each vertex in that triangle as shown in Figure 2.4a. In this example and following the point insertion, two edges were "flipped" so that the triangulation maintains the Delaunay criterion. The worst case scenario would occur when such an insertion affects all existing triangles, that is when all edges of the triangulation have to be "flipped". An upper bound of time steps required in such a case is $O(k)$, where k is the number of triangles at the time of insertion. However, experimental findings have shown that random point insertions would only affect a few edge flips, thus rendering restructuring upon each point insertion to be easily considered to have a time complexity of $O(1)$ provided affected triangles have been found.



(a) *Steiner* point is inserted in the center of the circumscribing circle of triangle 3.

(b) Marked edge is non-Delaunay compliant.

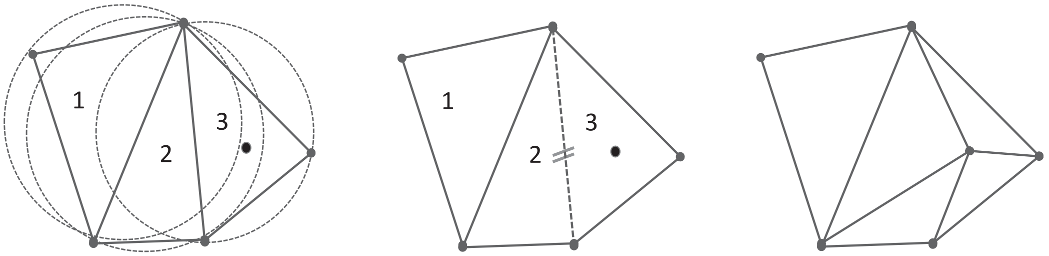
(c) The non-Delaunay compliant edge is “flipped”.

(d) Marked edge becomes non-Delaunay compliant.

(e) The non-Delaunay compliant edge is “flipped”.

Figure 2.4: **Lawson’s incremental point insertion:** (a) a new *Steiner* point is inserted in the center of the circumscribing circle of triangle 3 (i.e. a case of mesh refinement). (b) each vertex of the containing triangle is connected to the *Steiner* point. (c) (d) (e) show 2 edges that are “flipped” to maintain the Delaunay criterion.

Bowyer/Watson [28, 29] provided another restructuring technique in order to maintain Delaunay criterion upon point insertion. In this method, and as illustrated in Figure 2.5, when a point is inserted into a triangle (triangle 3 in this example), triangles, whose circumscribing circle contains this point, get destroyed (triangle 2 and 3, but not triangle 1). Each of the vertices of the polygon cavity created from destroying non-Delaunay compliant triangles becomes connected to the newly inserted point.



(a) Insertion location of a point.

(b) Two triangles lost their Delaunay criterion due to the newly inserted point.

(c) The common edge(s) of the non Delaunay triangles due to a new point insertion are eleminated

Figure 2.5: **Bowyer/Watson incremental point insertion:** (a) A pre-defined point p is inserted into the triangulation (i.e. the case of initial mesh generation). (b) Upon locating the triangle containing p (in this case triangle 3), the triangles whose circumscribing circle contains p are destroyed, in this case triangles 2 and 3. (c) then the newly inserted point is connected to each vertex of the cavity created.

2.4.2 Initial Mesh Generation

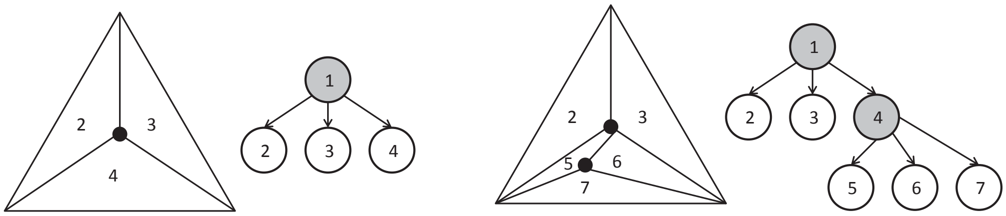
One of the major challenges encountered in the initial mesh generation is “point location”. This is mainly due to the fact that the points to be inserted are already pre-defined (i.e. their x and y coordinates are already set) on the geometry prior to starting the triangulation. The challenge is related to identifying the triangle which contains the point at the time of its insertion. If “point location” is implemented in a naive way, a search over all triangles would be required for each point insertion (i.e. $O(n)$ for each point insertion) where for each triangle the *counter-clock-wise* (CCW) test (Equation 2.2) is performed to determine whether or not a given point falls within that triangle.

The Counter-Clock-Wise (CCW) test: in order to determine whether or not a p_i falls within the triangle (p_1, p_2, p_3) using CCW, we need only obtain the directions of rotation along the triplets (p_1, p_2, p), (p_2, p_3, p) and (p_3, p_1, p). The point is inside if and only if the three directions are positive: $D_{12} > 0$, $D_{23} > 0$ and $D_{31} > 0$.

$$D_{12} = \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_i & y_i \end{pmatrix} \qquad D_{23} = \begin{pmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_i & y_i \end{pmatrix} \qquad D_{31} = \begin{pmatrix} 1 & x_3 & y_3 \\ 1 & x_1 & y_1 \\ 1 & x_i & y_i \end{pmatrix} \tag{2.2}$$

Many efficient algorithms have been devised to achieve n point insertions in $O(n \log n)$ time [31, 32, 30]. One such algorithm is related to using a Directed Acyclic Graph (DAG) to represent the history of every triangle that has been created throughout the Delaunay triangulation. Each node in this graph represents a triangle [30]. The root node represents the first triangle containing all p points to be inserted. When a triangle is split into multiple triangles, its node is kept in the graph, gets labeled as "destroyed", and points to the nodes corresponding to the newly created triangles. Despite the fact that a history of all created triangles is kept, the storage complexity of the DAG is $O(n)$ [33].

The construction of the DAG data structure is illustrated in Figures 2.6, 2.7 and 2.8. In Figure 2.6a, a node is inserted in the first triangle (triangle 1) which contains all the p points to be inserted in the initial mesh generation process. The new point divides the triangle into 3 triangles (numbers 2, 3 and 4) and the node in the DAG corresponding to triangle 1 are grayed out to highlight the fact that its corresponding triangle has been "destroyed". The same process is repeated when a node is inserted into triangle 4 (Figure 2.6b).

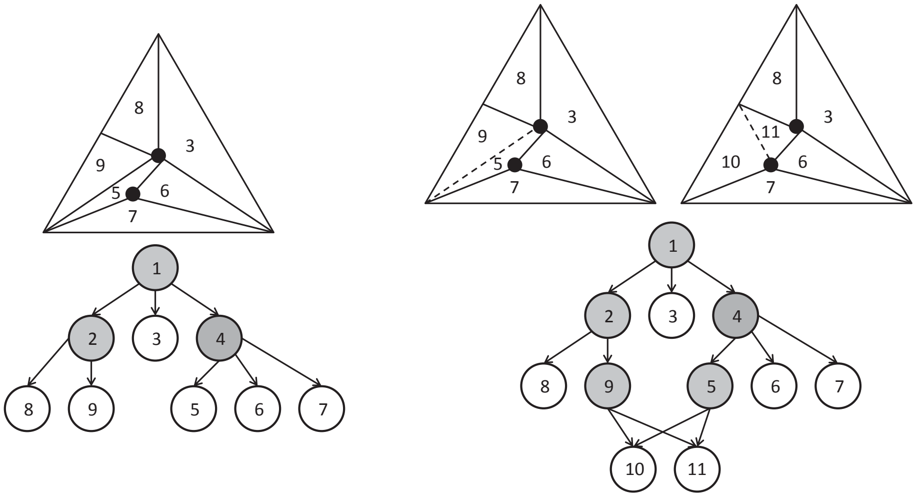


(a) New point is inserted in triangle 1.

(b) New point is inserted into triangle 4.

Figure 2.6: **Point location using Directed Acyclic Graph (1):** (a) Showing the DAG after inserting a point in the main triangle which contains all the points p to be inserted. (b) Showing the DAG after inserting a point into triangle 4.

Restructuring the triangulation after inserting a point into triangle 4 is accomplished as such: assuming that the left edge of the main triangle is "constrained" or a boundary segment (i.e. cannot be "flipped" or removed), then it is split at its mid-point since its circumscribing circle contains vertex 4 causing triangle 2 to be divided into triangle 8 and triangle 9 (Figure 2.7a). Vertex 4 is said to be *encroaching* upon a boundary segment [34]. Furthermore, in order to maintain the Delaunay criterion, Figure 2.7b shows an edge being flipped which causes triangle 5 and triangle 9 to be replaced by triangle 10 and triangle 11.



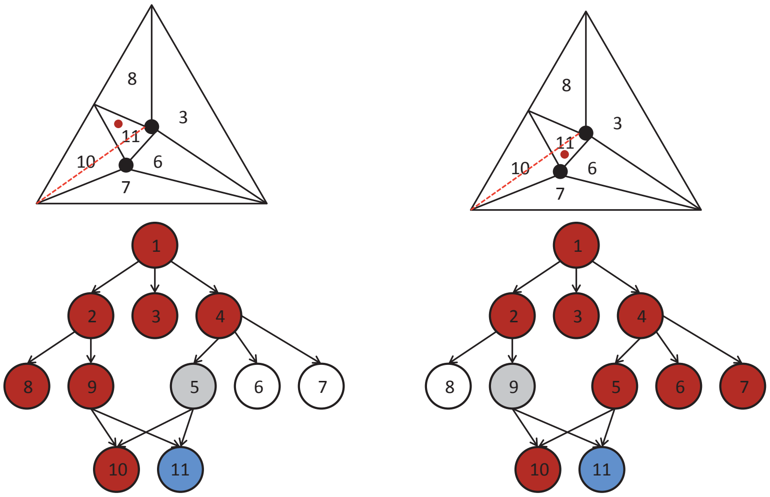
(a) The left edge of the main triangle is split.

(b) Edge "flipping" to maintain Delaunay criterion.

Figure 2.7: **Point location using Directed Acyclic Graph (2):** (a) Many triangles lost their Delaunay criterion after a new point was inserted into triangle 4. Assuming that the left edge of the main triangle is "constrained" (i.e. cannot be "flipped" or removed), then it is split in it mid-point since its circumscribing circle contains another vertex (vertex 4). Triangle 2 is replaced by triangles 8 and 9. (b) Edge is flipped to maintain the Delaunay criterion causing triangles 5 and 9 to be replaced by triangles 10 and 11.

Next, the process of traversing the DAG data structure to locate a point inserted into triangle 11 is illustrated. Depending on the exact location of that point in triangle 11, there are two paths that can be traversed when locating the triangle. Those paths are shown in Figure 2.8a and Figure 2.8b. In each possibility, the maximum nodes that are visited and tested using the counter-clock-wise test are highlighted in red.

Initial Delaunay triangulation of p points is not a time consuming process relative to the time it takes to generate a quality mesh of n points since $p \ll n$, despite the fact that "point location" performed for each of the p points during initial mesh generation is more time consuming than when inserting a point upon refinement. Most mesh generation time is usually spent in refining a mesh to meet a quality measure as will be explained in the next section.



(a) Locating a point inserted into triangle 11.

(b) Locating a point inserted into triangle 11.

Figure 2.8: **Point location using Directed Acyclic Graph (3):** (a) Showing the traversed nodes in the DAG (in red) to locate the triangle containing a newly inserted point. Despite that in cases (a) and (b) the point is in triangle 11, the traversed DAG path is different in each case. The location of the point relative to a *destroyed* triangle (i.e. triangle 2) affects the path to be followed.

2.4.3 Delaunay Mesh Refinement

Creating a Delaunay triangulation from a set of points, p, does not result in a quality mesh. Triangles may still be poorly shaped due to small or large angles despite the fact that Delaunay triangulation maximizes the minimum angle. Small angles cause poor conditioning and large angles cause discretization error. Other requirements for a quality mesh are to impose user requirements on the area of the triangles and the ability to generate more elements (smaller elements) in regions where higher accuracy is needed.

Generating a quality mesh by refinement of an initial Delaunay based triangulation can be completed using Chew's [35, 36] and Ruppert's [37] algorithms. These algorithms share many similarities, they tackle poorly shaped triangles or triangles that do not fit into the user requirement by inserting *Steiner* points in the circumcenter[1] of bad triangles and re-triangulating, either by using Lawson's edge flip or Bowyer/Watson cavity expansion, to maintain the Delaunay criteria. In those algorithms, a quality measure of a bad triangle is its *B=circumradius*[2]*-to-shortest edge ratio.*. Chew's algorithms [35, 36] consider a triangle with a B larger than 1 to be bad and guarantees a mesh which will not have triangles with angles smaller than 30°. Ruppert's algorithm sets this bound to be $\sqrt{2}$ leading to a mesh with no angles less than 20.7°.

Shewchuk has implemented an optimized sequential 2D mesh generator that he called *Triangle* [20] based on Chew's and Ruppert's algorithms, albeit with modifications [38]. It is considered by many to represent the fastest sequential 2D mesh generator software and is used as a baseline to evaluate the speedup of parallel mesh generators. Table 2.1 shows the duration in seconds required to refine an initial mesh that has 12,032 vertices (23,968 triangles). The same mesh has been refined by imposing different requirements on the maximum triangle area size. The most refined mesh consumed a duration of approximately 27 seconds to be generated and has 9,202,654 nodes, suggestive of a large problem size (2GB of memory).

Shewchuk's Delaunay mesh generation implementation in *Triangle* considers a triangle to be the basic building block of a mesh; this is in contrast to an alternative representation which considers an edge to be the basic building block of a mesh. Each triangle is represented using 6 pointers. Three of which point to the triangle's vertices and 3 to the neighboring triangles. A search over neighboring triangles in this case is a

[1] Circumcenter of a triangle is the center of its circumscribing circle.

[2] Circumradius: the radius of the circumscribing circle

depth-first-search [39] (or a breadth-first-search [40]) [41] that starts from the current triangle, investigates neighboring triangles and so on. A total estimate of memory required to represent a mesh, not taking into account the data structures required by the algorithm (i.e. tree and queue data structures) is estimated in Equation 2.3.

$$\begin{aligned}\text{Memory(bytes)} = & \ [\text{number of nodes} \times (2+k) \times \text{sizeof(double)}] \\ + & \ [\text{number of triangles} \times 6 \times \text{sizeof(pointer)}]\end{aligned} \tag{2.3}$$

where k is the number of attributes to be annexed to each node (e.g. material properties).

Table 2.1: Delaunay mesh refinement times of an initial mesh (12,032 vertices and 23,968 triangles) of a brushless DC motor using Shewchuk's *Triangle* 2D mesh generator.

Maximum triangle area size (mm^2)	Number of vertices	Number of triangles	Time on i7-860 (seconds)	Time on AMD (seconds)
0.01	736,637	1,470,964	0.898	2.155
0.005	1,473,761	2,944,521	1.816	4.322
0.002	3,681,377	7,357,461	4.696	9.886
0.001	7,361,041	14,715,479	8.939	20.770
0.0008	9,202,654	18,397,332	11.973	27.254

Next, the performance of *Triangle* was analyzed using Intel's VTune Analyser [21] for the case when the initial mesh of the brushless DC motor which contains 12,032 nodes (Figure 2.1) was refined to contain 9,202,654 nodes. The chart in Figure 2.9 shows the distribution of execution cycles spent on different kernels of the application. Inserting a point, which involved searching for the triangles affected by the insertion and re-triangulating (using Lawson's edge "flipping" technique) and in addition testing for the *"inCircle"* property exhausted the largest amount of the application's execution cycles (*insertvertex*=27.16%, *"incircle"*=27.41%). Such a result has also been reported in [40] which found that re-structuring, using the Bowyer/Watson cavity expansion technique, accounts for 58% of the mesh generation time. Despite that, those kernels exhibited data access locality since only 18% of the total application execution cycles were spent retrieving long latency data (DRAM) due to misses on the last level cache (Level-3) of an Intel i7-860 processor. On the other hand, kernels aimed at identifying (*testtriangle*), queuing(*enqueubadtriang*), and dequeuing(*dequeuebadtriang*) bad triangles suffered from close to 80% of L3 cache misses.

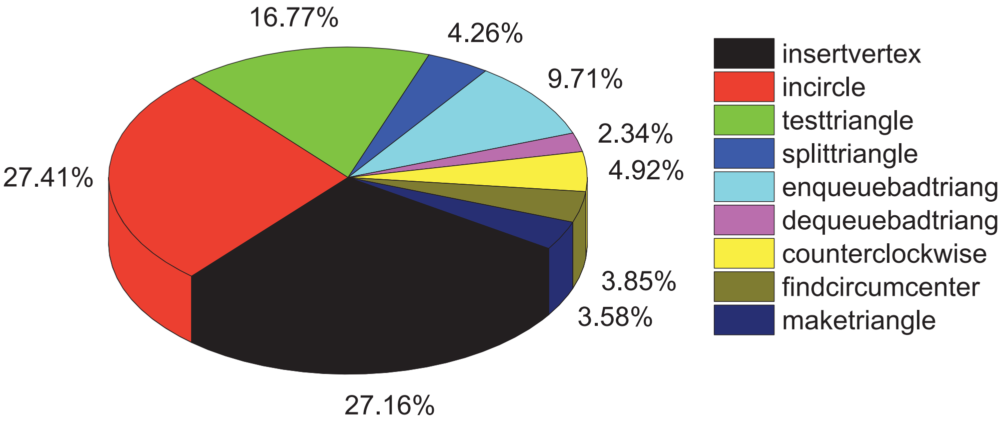
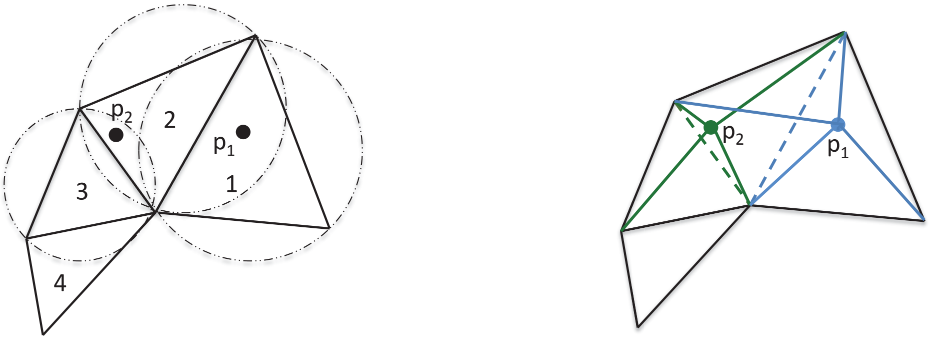


Figure 2.9: **Shewchuk's *Triangle* performance evaluation.** Distribution of workload of a Delaunay refinement of an initial mesh of 12,032 nodes. The maximum triangle area of the refined mesh was set to $0.00008mm^2$.

2.5 Parallel Mesh Refinement

Parallel mesh refinement aims at dividing the geometry into sub-domains and at the simultaneous insertion of multiple *Steiner* points in each sub-domain. As discussed earlier in § 2.4.1, each of the inserted points causes structural updates on one triangle or more. Challenges are encountered when multiple point insertions affect common triangles as illustrated in Figure 2.10. In this example, assuming that the Bowyer/Watson cavity expansion algorithm is performed, the cavity expansion given the insertion of p_1 has destroyed both triangles 1 and 2, whereas, the cavity expansion given the insertion of p_2 has destroyed triangles 2 and 3. The overlapping between cavities given triangle 2 being shared has been shown to be preventative of the simultaneous insertion of p_1 and p_2. Many techniques aimed at overcoming this barrier have been suggested. A survey of these techniques and the research progress up until 2005 can be found in [42]. In this survey, the author distinguished between two types of domain decomposition: continuous and discreet domain decomposition. In the former, the geometry of the prototype under investigation is broken down into sub-problems. In the latter, the coarse mesh of the main problem is divided into sub-domains. The author has further identified various parallel meshing techniques based on the coupling levels between sub-domains (i.e. tightly coupled, decoupled and partially coupled domains)



(a) Edge e non Delaunay compliant

(b) Conflicting cavities

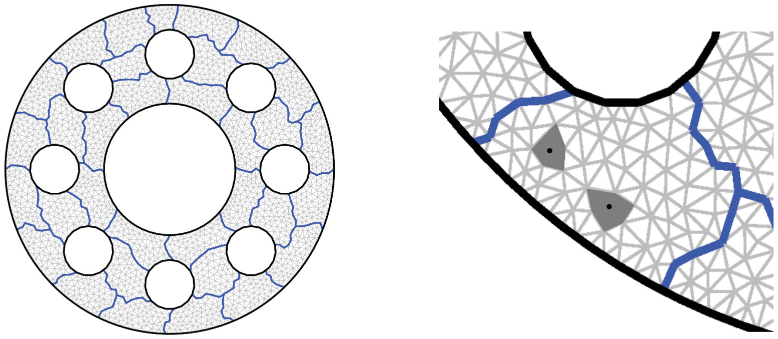
Figure 2.10: **Conflicting 2 points insertion into a mesh.** The cavity created from destroying Δ_1 and Δ_2 after inserting p_1 overlaps with the cavity created from destroying Δ_2 and Δ_3 after inserting p_2.

Tightly coupled methods treat sub-domains as one whole entity even if they are distributed across multiple processors (in the case of a multi-processor machine). Upon the simultaneous insertion of *Steiner* points into each of the sub-domains, any form of subsequent restructuring of a set of triangles would be permitted to propagate to the other domains unless they are not restructuring the same triangles, in which case conflicts between expanded cavities would arise. Commonly used tightly coupled parallel Delaunay mesh generators utilize an “optimistic” approach, where multiple points are concurrently inserted into the mesh without performing any pre-analysis of a cavity conflict possibility [43, 44, 45]. Typically in an optimistic approach, threads wishing to modify the locked structure must either acquire locks on each of the triangles or vertices they wish to modify throughout the restructuring process. However, if a triangle or vertex was already locked by a thread then other threads must either rollback their operations and insert points in different locations or wait for locks to be released by the thread holding them. Either way, waiting to acquire a lock or rollbacks, a loss of computational resources is unavoidable which is the reason why many argue that tightly coupled methods are less efficient than are other parallel meshing techniques (i.e. decoupled and partially coupled methods) particularly on multi-processor machines where such fine-grained synchronization is computationally expensive. Despite that, published results have shown that good performance can be attained on both platforms (multi-core and multi-processors) provided there is an efficient implementation of rollbacks and synchronization processes. For instance, on a cluster of workstations (16 processors), six times speedup has been attained when refining a 3D mesh where 20% of the processing time was spent on frequent polling for asynchronous messages arriving from other domains. Kulkarni [44] has achieved a three times speedup on four processors. Further, Batista [45] has modified an efficient sequential 3D mesh generator (CGAL [46]) and was able to attain a speedup of five on an eight cores processor. Chrisochoides and Nave [47, 48] reported $O(\log p)$ speedup on p processors, where $p \gg N$ and N is the number of sub-domains.

Decoupled mesh generator methods have been shown to be both advantageous as well as appealing to CAD software in that they enable reusing the code of sequential mesh generators. These methods rely upon continuous domain decomposition (or geometric domain decomposition) in which the geometry is divided (prior to meshing) into sub-domains, each of which is meshed and independently refined through the use of sequential mesh generators. The interfaces between sub-domains, known as "separators", consist of triangles that are constructed prior to meshing and are made Delaunay compliant. Separators should appear in the final triangulation when sub-domains are joined together [49], a condition that is difficult to maintain when the geometry under investigation is complex, making this method impractical in real life problems. One of the reasons a separator cannot appear as is in a the final triangulation is that the angles created between the separator and the mesh are very small, producing a bad quality mesh and in such a case, the separators are referred to as being Delaunay inadmissible. Linardakis [50] obtained super-linear speedup by using parallel 2D mesh generator based on decoupled methods. Also, he has laid out criteria that separators should meet in order to be able to produce a quality mesh.

The partially decoupled method has been the method of choice since its emergence as it utilizes discrete domain decomposition that takes place following the initial mesh generation. Further, this method sets clear boundaries separating domains by choosing triangle edges. Under this approach, the problem is reduced to one which is similar to a constrained Delaunay triangulation (CDT) technique [51]. In CDT, it is not possible for the constrained edges to be flipped or destroyed throughout the creation process of the cavity, albeit they are susceptible to splitting (i.e. the case of an encroached edge). Whenever a sub-domain splits an edge boundary, it has to communicate the end nodes of the edge to be split to other sub-domains that are sharing the constrained edge. Chernikov [52] implemented an asynchronous communication scheme between sub-domains in which he aggregated multiple *split* messages into one to reduce communication overhead on the condition that the order of splitting is maintained. He reported $O(p)$ speedup using this method where more than one billion triangles were generated in less than 3 minutes on 100 processors.

An example of a mesh generator that combines both partially decoupled and tightly coupled methods is the Parallel Constrained Delaunay Mesh (PCDM) [40, 53]. In PCDM, coarse-grained parallelism is explored by decomposing a discrete mesh into sub-domains to run on multi-processors. The sub-domains are partially decoupled where only *split* messages are exchanged between sub-domains as explained above and illustrated in Figure 2.11a. Medium-grained parallelism is explored within each sub-domain using the optimistic approach in inserting multiple points as depicted in Figure 2.11b. Furthermore, since cavity expansion accounts for 58% of the software runtime, the author used multiple threads in order to work on expanding each of the cavities (i.e. fine-grained parallelism).



(a) Coarse-grained parallel mesh generation

(b) Medium-grained parallel mesh generation

Figure 2.11: **Multigrain parallel mesh generation:** (a) In coarse-grained parallel mesh generation, *Steiner* points are inserted in each sub-domain simultaneously. Sub-domains sharing edges need to communicate only when an a constrained edge is split. (b) medium-grained parallelism works when multiple points are inserted into the same sub-domain. Fine-grained parallelism is explored using more than one thread to expand the same cavity. Reprinted with permission [40].

Another parallel mesh generation approach that is distinct from the above discussed ones involves the pre-analysis of *Steiner* point independence and subsequent insertion of points among which a conflict would be implausible. This approach appears to be effective as well as valid given the high degree of parallelism exhibited by mesh refinement [40]. Antonopoulos *et al* have conducted statistical analysis to estimate the lower and upper bounds (i.e. degree of parallelism) of the number of concavities amenable to concurrent expansion while the number of processors increased from 32 to 512. The authors reported that the extent of parallelism yielded was sufficient to enable its execution on 512 threads. The only exception for that would be the case when the refinement process reaches the last 1,000 cavities and most of the mesh conforms to user's set quality measures. Further, the author found that regardless of the structure and size of the problem, the average number of triangles in a given cavity approximates five. This argument also extends to three other distinct problems, each of which was refined to 1 million and 10 millions triangles. Spielman *et al* [54, 55, 56] performed investigations, that serve as examples of parallel mesh generation, that explore the high degree of parallelism in mesh generation. *Steiner* points were inserted concurrently if their corresponding circumcircles do not contain each other centers. The resulting parallel algorithm achieved $O(\log m)$ on m processors, where m is the output size.

2.6 Concluding Remarks

The state-of-the-art mesh generation research discussed in this chapter as well as the experiments on *Triangle* provide compelling evidence supporting the performance of sequential Delaunay mesh generators as key in attempting to elucidate efficient parallel mesh generators, regardless of the nature of the utilized parallel techniques. The performance of sequential Delaunay mesh generators is largely dependent upon the underlying data structure that is used to store the mesh.

Generally, the mesh generation process has proven to be time efficient; efficient sequential implementation has shown that it is feasible for mesh problems that are considered to be large on a desktop computer in a duration that does not exceed one minute. Clearly, such a duration, relative to the time duration consumed throughout the overall design process, proves to be quite short [57]. On multi-processor machines, parallel mesh generation has shown to scale even for a large number of processors.

Multi-core processors are considered advantageous when tightly coupled techniques are utilized. This is primarily due to the fact that these methods involve extensive use of synchronization primitives and rollbacks. Moreover, fine-grained parallelism has been explored on a multi-core processor using dedicated multiple threads for the purpose of expanding each cavity upon points insertions. However, the decoupled technique (i.e. coarse-grained parallelism) and parallel mesh generation methods that pre-analyze the domain to insert independent *Steiner* points do not take advantage of the shared cache on multi-core processors. Those techniques naturally do not exhibit temporal or spatial locality of access. In this case, the cache should be divided in such a way as to avoid cache contention and also false cache sharing.

Chapter 3

Finite Element Matrix Assembly and Matrix Test Sets

Contents

3.1 Introduction

The process of discretization of a 2D space geometry and creating a mesh is an important geometric task that should enable the appropriate representation of the PDE to be solved over the domain. The size of the mesh elements and the quality of the triangles are of critical importance as they determine the accuracy of the solution. However, solving the governing differential equation over the discretized domain is the core of the *Finite Element Method*. This usually involves 4 steps: 1) discretize the

PDE over the mesh, 2) assemble the discretized parts (elements) into a global matrix A, 3) apply boundary conditions to obtain a system of equations $Ax = b$ and 4) finally solve it. This chapter focuses on the assembly process (step 2) and subsequent chapters investigate the solver part. Since the details of step (1) are beyond the scope of this thesis, it will briefly described it in the remainder of this section and readers seeking a detailed explanation are referred to the books [58, 59].

In low frequency EM device analysis, the goal is to solve the general elliptic second order partial differential equation (Equation 3.1) over the discretized geometry. In the case of static analysis, this equation reduces to solving Poisson's equation $\nabla.(\varepsilon\nabla\varphi) = \rho$ (Equation 1.11). A new formulation of Equation 1.11, termed as the weak form, is required to enable it to be discretized and applied on each mesh element separately from the other elements.

$$\frac{\partial}{\partial x}(\alpha_x \frac{\partial \phi}{\partial x}) + \frac{\partial}{\partial y}(\alpha_y \frac{\partial \phi}{\partial y}) + \beta\phi = f \tag{3.1}$$

Next, the goal is to approximate the solution within each element as a combination of the unknowns within that element which depends on the type of FEM formulation used. For instance, in a linear first-order nodal finite element analysis, the unknowns are the nodes of the triangles (i.e. three unknowns ϕ_1, ϕ_2, ϕ_3) per triangle and the unknown function ϕ is approximated as:

$$\phi^e(x, y) = a^e + b^e x + c^e y \tag{3.2}$$

where a, b and c coefficients are to be determined. When a more accurate solution is desired, more unknowns can be added to each element. For instance, adding nodes in the middle of each edge would lead to quadratic element analysis (second-order) and the function between the six unknowns becomes:

$$\phi^e(x, y) = a^e + b^e x + c^e y + d^e x^2 + e^e xy + f^e y^2 \tag{3.3}$$

In both cases, and even with higher order elements, the process of matrix assembly remains the same. Equations 3.2 and 3.3 can be expressed as a sum of interpolation functions as shown in

$$\phi_j^e(x,y) = \sum_{j=1}^{u} N_j^e(x,y)\phi_j^e \tag{3.4}$$

where u is the number of nodal points within an element. Equation 3.1 can now be written in terms of these unknowns and integrated over all elements; this form is known as the weak formulation and is given by

$$A_{ij}^e = \int\int_{\Omega^e} (\alpha_x \frac{\partial N_i^e}{\partial x}\frac{\partial N_j^e}{\partial x} + \alpha_y \frac{\partial N_i^e}{\partial y}\frac{\partial N_j^e}{\partial y} + \beta N_i^e N_j^e)\, dxdy \tag{3.5}$$

$$b_i^e = \int\int_{\Omega^e} f N_i^e dxdy \tag{3.6}$$

where

$$[A]\{\phi\} = \{b\} \tag{3.7}$$

3.2 Matrix Assembly Data Structure

The process of matrix assembly is not considered to be time consuming. It is an $O(n)$ process, since it consists of iterating once over all mesh elements. For each element, two operations are performed. The first is to approximate the solution of the field within each element by applying Equation 3.5, which would result in a dense matrix structure $L_{u\times u}^e$ for each element e where u depends upon the formulation and the number of unknowns in an element. The second operation is to map each entry of the dense matrix $L_{u\times u}^e$ to a global matrix A (Equation 3.7). This step constitutes a significant portion of the total assembly cost mainly because the global matrix A is sparse. Inserting and updating entries in a sparse matrix, even when its structure is a priori known, is not trivial. Both sequential and parallel implementation details of this step are explained and experiments are carried out to determine the efficiency of such a process.

The basic data structure for a finite element mesh consists of 3 arrays. The first is an array P that holds the space coordinates of the nodes of the mesh, which, in the case of a 2D analysis is an $n \times 2$ array of double precision floating-point numbers (i.e. $P_{n\times 2}$). The triangles are stored in a $t \times 3$ array T where each row contains the three nodes of each of the t triangles. Finally the edge data structure contains all the boundary edges of the 2D geometry. This is usually an $E_{(edge\times 2)}$ matrix, where the number of edges depends on the geometry. Typically, there exists a need for additional information that is either part of the basic mesh data structure or organized in different arrays. For instance, physical parameters related to the material properties of each node or edge are stored as well as any boundary conditions.

Algorithm 3.1 Matrix assembly of a first-order nodal finite element formulation.

```
Input: $T_{t\times 3}$ stores the indices of nodes of each of the $t$ triangles
Output: Sparse $n \times n$ matrix $A$
$A = sparse(n, n)$ stores the contributions of all elements into an $n \times n$ matrix
for $ele = 1 \to t$ do
    for $i = 1 \to 3$ do
        # Obtain the global row location $iGlobal^{ele}$ of the nodes in $T_{ele,-}$
        $iGlobal^{ele} := T_{ele,i}$
        for $j = 1 \to 3$ do
            # Obtain the global column location $jGlobal_{ele}$ of the nodes in $T_{ele,-}$
            $jGlobal^{ele} := T_{ele,j}$
            $A(iGlobal^{ele}, jGlobal^{ele}) := A(iGlobal^{ele}, jGlobal^{ele}) + L^{ele}_{i,j}$
        end for
    end for
end for
```

Algorithm 3.1 and Figure 3.1 illustrate the assembly process of a first-order nodal finite element analysis. Each node is considered to be an unknown, hence the contribution of each element e to the global matrix A is an $L^{e}_{3\times 3}$ dense matrix, usually referred to as the "local matrix". Each entry of a triangle's local matrix L^{e}_{ij}, where $i = 1, 2, 3$ and $j = 1, 2, 3$, is then mapped to a location in the global matrix, $A_{iGlobal,jGlobal}$ where $iGlobal = T(i)$ and $jGlobal = T(j)$. As shown in Table 3.1, T is a matrix that lists

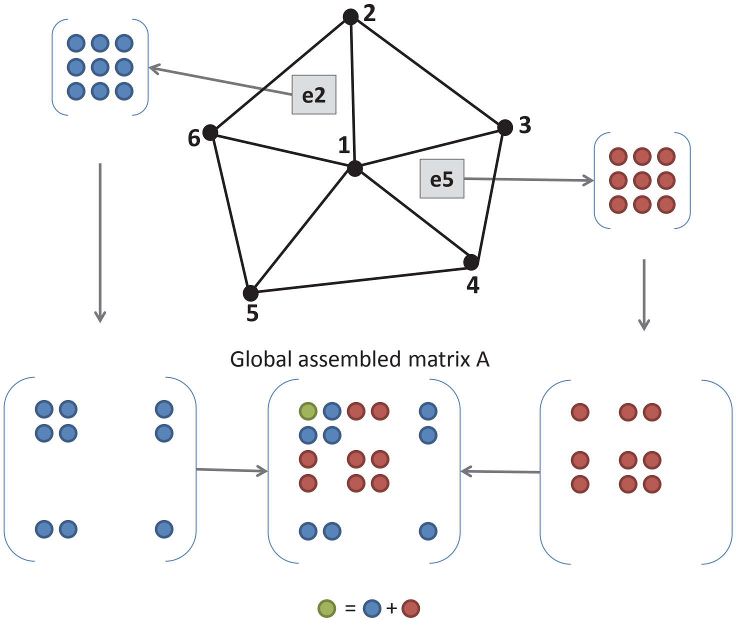


Figure 3.1: **First-order nodal finite element assembly process of two elements.** The assembly process of two elements ($e = 2$ and $e = 5$) into the global matrix A. Each element produces a local 3 × 3 dense matrix which is mapped and aggregated into the global sparse matrix A. One or more entries of the local matrices can map into the same location in the global matrix (shown in green).

the global node number of each of the triangle's nodes (i.e. node 1, node 2, and node 3). For instance, suppose that $L^5_{3\times3}$ is the local matrix of element $e = 5$ (i.e. Δ_{143}). In order to map the entry L^5_{13} to a location in the global matrix A, the local index 1 is mapped to the global index 1 since $T^5(1) = 1$ and the local index 3 is mapped to the global index 4 since $T^5(3) = 4$. Of note, the local matrix L is not explicitly calculated, but instead two nested *for-loops*, each of size 3, are executed where in each iteration an entry is aggregated into the global matrix A, as illustrated in Algorithm 3.1 (line 2 - line 11)

Table 3.1: **Connectivity table of a first-order nodal finite element formulation.** Table showing the connectivity matrix T which is used to map the index of an entry of a triangle's local matrix L to its corresponding index in the global matrix A.

Element number	Node 1	Node 2	Node 3
$e = 1$	1	3	2
$e = 2$	1	2	6
$e = 3$	1	6	5
$e = 4$	1	5	4
$e = 5$	1	4	3

Algorithm 3.2 and Figure 3.2 illustrate the matrix assembly process of a second-order nodal finite element analysis. The assembly process resembles that of a first-order nodal analysis, except that additional nodes are inserted in the middle of each of the edges of the mesh and the connectivity matrix T is augmented to accommodate the new nodes (Table 3.2). Each element contains 6 unknowns (3 vertex nodes and 3 mid-edge nodes), hence the contribution of each element to the global matrix A is a 6×6 dense matrix.

Algorithm 3.2 Matrix assembly of a second-order finite element nodal formulation.

```
Input: $T_{t\times 3}$ stores the indices of nodes of each of the $t$ triangles
Output: Sparse $n \times n$ matrix $A$
  $K = sparse(n, n)$ stores the contributions of all elements into an $n \times n$ matrix
  for $ele = 1 \to t$ do
    for $i = 1 \to 6$ do
      # Obtain the global row location $iGlobal^{ele}$ of the nodes in $T_{ele,-}$
      $iGlobal^{ele} := T_{ele,i}$
      for $j = 1 \to 6$ do
        # Obtain the global column location $jGlobal_{ele}$ of the nodes in $T_{ele,-}$
        $jGlobal^{ele} := T_{ele,j}$
        $A(iGlobal^{ele}, jGlobal^{ele}) := A(iGlobal^{ele}, jGlobal^{ele}) + L^{ele}_{i,j}$
      end for
    end for
  end for
```

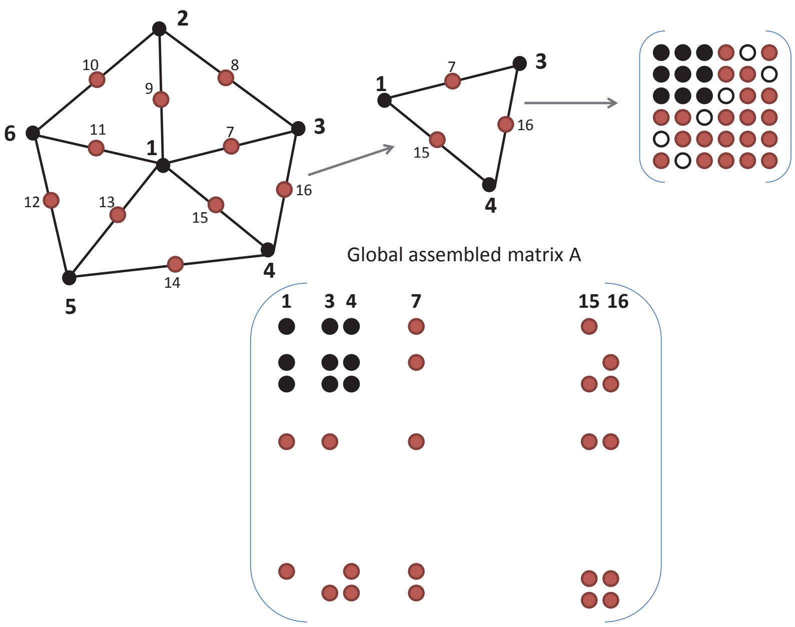


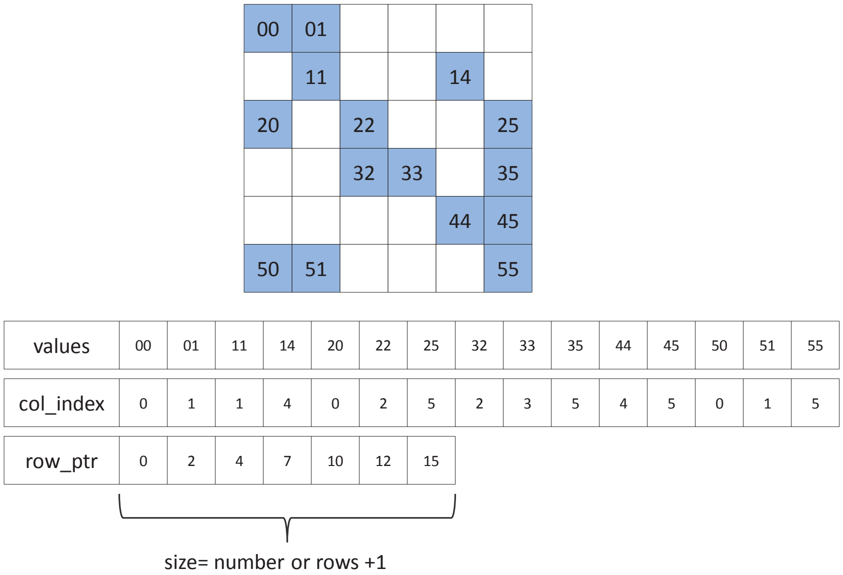
Figure 3.2: **Second-order nodal finite element assembly process of one element.** The assembly process of element Δ_{143} into the global matrix A. Mid-nodes are added into each edge of the triangulation.

Table 3.2: **Augmented connectivity table of a second-order nodal finite element formulation.** Table showing the connectivity matrix T which is used to map the index of an entry of a triangle's local matrix L to its corresponding index in the global matrix A.

Element number	Node 1	Node 2	Node 3	Mid-node 1-2	Mid-node 2-3	Mid-node 1-3
$e = 1$	1	3	2	7	8	9
$e = 2$	1	2	6	9	10	11
$e = 3$	1	6	5	11	12	13
$e = 4$	1	5	4	13	14	15
$e = 5$	1	4	3	15	16	7

3.3 Matrix Assembly Using The ELLPACK Sparse Storage Format

Sparse storage schemes are powerful ways to save memory space when storing and processing sparse matrices. Only non-zero entries in a matrix are stored which constitutes a small fraction of the zero entries. Despite their effectiveness in saving storage, they lead to algorithms that are more complex than those of a dense storage scheme, both in terms of data structures and the flow of program control.



00	01				
	11			14	
20		22			25
		32	33		35
				44	45
50	51				55

values	00	01	11	14	20	22	25	32	33	35	44	45	50	51	55
col_index	0	1	1	4	0	2	5	2	3	5	4	5	0	1	5
row_ptr	0	2	4	7	10	12	15								

Figure 3.3: **Compressed Sparse Row Storage (CSR).** The non-zero entries of the matrix are stored in the *values* array. Each entry in the *values* array has a corresponding entry in the *col_index* array to indicate its column index in the matrix. The *row_ptr* array holds the index of the first non-zero of each row in the *values* array.

The Compressed Sparse Row storage format (CSR) is a simple and widely used storage format that can store any matrix regardless of its structure. Figure 3.3 illustrates an example of the Compressed Sparse Row storage. Only the non-zero entries of the matrix are stored in the *values* array. Since all non-zeros are stored contiguously in the same array, The *row_ptr* array holds the index of the first non-zero of a each row in the *values* array. For instance, $row\_index[i]$ contains the index of the first non-zero of row i in the *values* array, that is, $values[row\_index[i]]$ returns the value of the first non-zero of the i^{th} row. Each entry in the *values* array has a corresponding entry in the *col_index* array to indicate its column index in the matrix.

When assembling a matrix without knowing its final structure, that is, when not knowing the number of non-zeros in each row and their distribution, the Compressed Sparse Row Storage, which stores all matrix entries in contiguous locations, becomes difficult to use. Each new entry insertion into a matrix stored in CSR requires two steps that can be completed in $O(n)$ time where n is the number of non-zeros. The first step, which can be preformed in $O(1)$ time, is to locate the position in the *values* array and the *col_index* at which the new entry is to be inserted. The second step, which can be performed in $O(2n)$ time, is to expand *values* and *col_index* arrays and shift all the elements to the right of the newly inserted entry. This process is illustrated in Figure 3.4 where a new entry, A_{02}, was inserted in the first row. After locating the position of the new entry in the *values* array (highlighted in red), the elements to its right were shifted by one location. The same was applied to the *col_index* array.

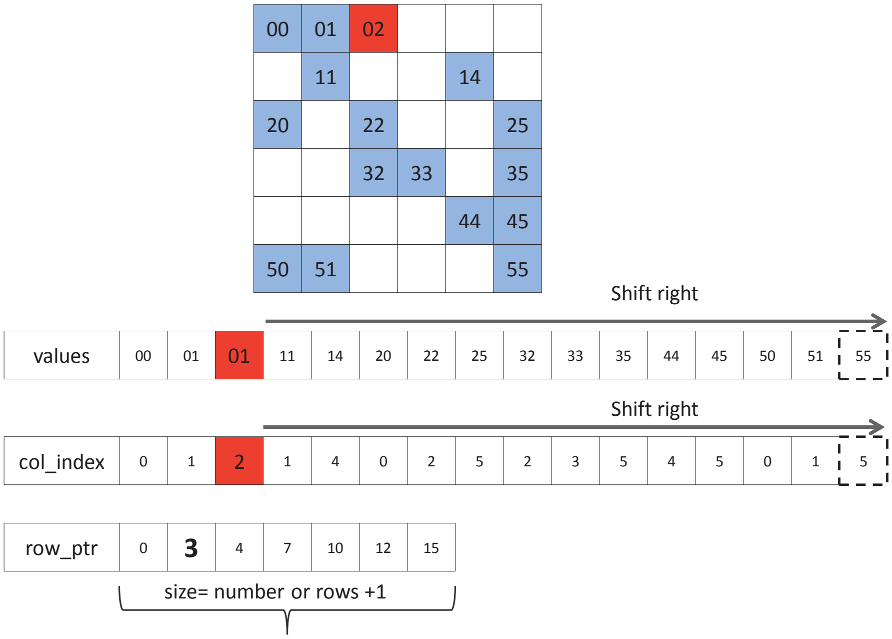
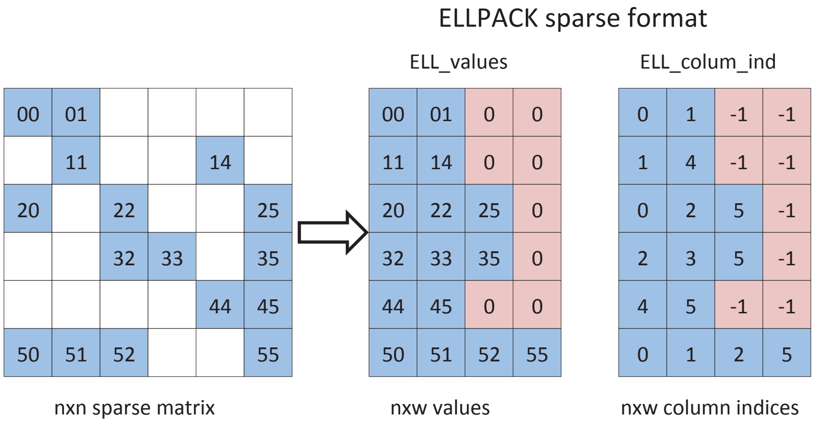


Figure 3.4: Inserting a non-zero entry into a CSR storage.

In order to overcome the limitation encountered when incrementally adding entries into a matrix stored in CSR, a variation of this storage, termed as "dynamic CSR", was developed in this thesis. Dynamic CSR creates a separate dynamic data structure for each row in a matrix so that adding an entry in a row would not impact the other rows. Not only does this storage work when the final data structure of the matrix is not known, but also when the maximum number of non-zeros per row is also not known. This case is encountered when developing preconditioners for the Conjugate Gradient Method (Chapter 6). Fortunately, in the case of matrix assembly in FEM, the maximum number of non-zeros of any row can be roughly estimated since it depends on the FEM formulation used. In such a case, a more suitable choice of a sparse storage than the dynamic CSR is to use the ELLPACK sparse storage [60].

The ELLPACK sparse format stores a $N \times N$ sparse matrix into two $N \times W$ dense data structures (ELL_values and ELL_column_ind) as shown in Figure 3.5. ELL_values stores the values of non-zeros in each row in a condensed form and pads the remaining spaces with zeros. ELL_column_ind stores the column index of each corresponding non-zero in the *ELL_values* and "-1" for the padded non-zeros. The size W corresponds to the maximum number of non-zeros per row. When the number of non-zeros per row is less than W, zeros are padded to fill the remaining locations.



00	01				
	11			14	
20		22			25
		32	33		35
				44	45
50	51	52			55

00	01	0	0
11	14	0	0
20	22	25	0
32	33	35	0
44	45	0	0
50	51	52	55

0	1	-1	-1
1	4	-1	-1
0	2	5	-1
2	3	5	-1
4	5	-1	-1
0	1	2	5

Figure 3.5: The ELLPACK sparse storage format.

The ELLPACK sparse storage format resolves the issue encountered when using the CSR storage, as discussed above, by pre-allocating a fixed amount of W entries for each row, where w is equal to the maximum number of non-zeros per row of the final matrix structure. However, this format has a disadvantage in that un-necessary locations are filled with zeros when the number of non-zeros in a row is less than w, those are known as *fill-ins*. Ideally, if the number of non-zeros in each row is W, then no fill-ins occur. In the case of matrices generated from FEM or FDM, the maximum number of non-zeros per row is low which makes the ELLPACK a suitable sparse storage scheme for matrix assembly, hence it will be used when performing matrix assembly experiments.

3.4 Parallel Matrix Assembly

Matrix assembly is a massively data-parallel process, as all elements can be processed simultaneously. However, one arising challenge is the race condition; that is, when multiple threads are attempting to modify the same memory location of the global matrix concurrently without synchronization. For instance, looking again at the matrix assembly process of a first-order nodal FE formulation illustrated in Figure 3.1, triangles Δ_{126} and Δ_{143} (elements $e = 2$ and $e = 5$ respectively) share node number 1. If a thread is aggregating the entry $L^2_{1,1}$ of the local matrix of $e = 2$ and another thread is aggregating the entry $L^5_{1,1}$ of the local matrix of $e = 5$, then there is a possibility that a race condition could occur since both entries map to the same location in the global matrix, that is $A_{1,1}$.

Two approaches to solve the problem of the race condition discussed above have been identified. The first is to use a *critical section*[1] when accessing entries in the global matrix [61] and the second is to use a coloring technique where elements that do not share nodes or edges are assigned the same colors [62, 63, 64, 65, 66] and are processed simultaneously. All of these techniques are applicable on both multi-core and many-core based processors. The matrix assembly experiments implemented in this chapter follow the first approach.

A new approach that has been developed by [67] involves eliminating the matrix assembly stage by utilization of the domain decomposition technique. Each element of the mesh is considered to be a sub-domain that is solved independently of the other elements. It is plausible to regard this approach as a non-overlapping additive Schwarz domain decomposition technique [68], where the size of the domain is that of a mesh element. However, it is important to be mindful of shortcomings from which this approach suffers, including that of low convergence since it utilizes the Jacobi method in addition to having sub-domains that are of a very small size which consequently increase the number the Jacobi iterations.

[1] Critical section: a section or a piece of a code that can be accessed by only one thread or process at a time.

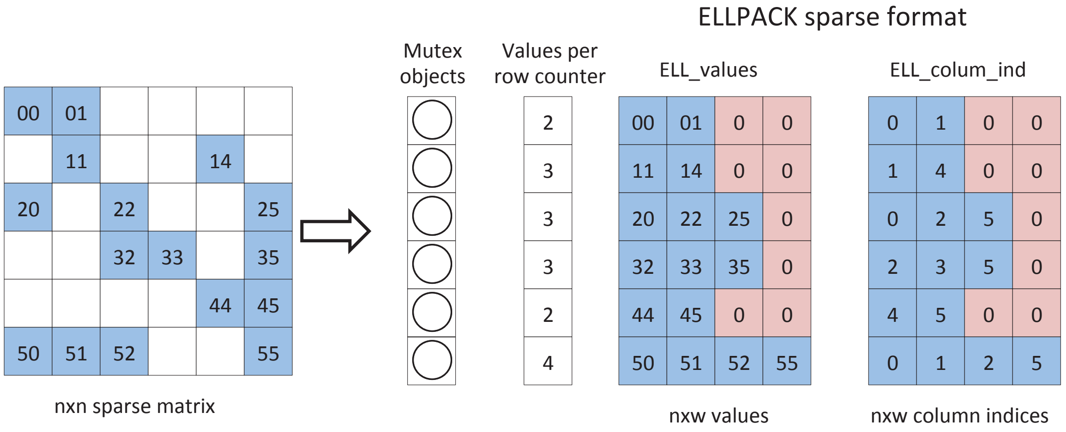


Figure 3.6: **Synchronized ELLPACK storage.** Each row in the global matrix has a corresponding *mutex* object stored in “Mutex objects” array. A thread must acquire a lock on the *mutex* object before it can modify the corresponding row in the global matrix.

3.5 Matrix Assembly Experiments

Algorithm 3.3 Multi-threaded matrix assembly using the ELLPACK sparse storage - First-order nodal finite element formulation.

Input: $T_{t\times 3}$ stores the indices of nodes of each of the t triangles
Output: ELLPACK sparse $n \times n$ matrix A

```
$K = sparse(n, n)$ stores the contributions of all elements into an $n \times n$ matrix
for $ele = 1 \to t$ do
    for $i = 1 \to 3$ do
        # Obtain the global row location $iGlobal^{ele}$ of the nodes in $T_{ele,-}$
        $iGlobal^{ele} := T_{ele,i}$
        $\boxed{\textbf{AquireLock( MutexObjects(}iGlobal^{ele}\textbf{) )}}$
        for $j = 1 \to 3$ do
            # Obtain the global column location $jGlobal^{ele}$ of the nodes in $T_{ele,-}$
            $jGlobal^{ele} := T_{ele,j}$
            $InsertOrUpdateELLPACK(iGlobal^{ele}, jGlobal^{ele}, value)$
        end for
        $\boxed{\textbf{ReleaseLock( MutexObjects(}iGlobal^{ele}\textbf{) )}}$
    end for
end for
```

In order to investigate the performance of parallel matrix assembly using *critical* *sections* on multi-core processors, *Pthreads*[2]*mutex*[3] objects have been used to synchronize the access of multiple threads to the global matrix rows. For this purpose, an array of n *mutex* objects was created where each object corresponds to a row in the global matrix as shown in Figure 3.6. Typically, in order for a thread to add or modify entries on a row of the global matrix, it must acquire a lock on the *mutex* object corresponding to that row. After the thread finishes its modifications, it releases the lock to make it available for other threads. For example, a thread that is assembling an element of 3 unknowns (ϕ_1, ϕ_2, ϕ_3) must aggregate the total of 3×3 entries in the global matrix. Each 3 of these entries are added onto the same row of the global matrix; hence, a total of 3 locks are required on 3 different *mutex* objects. This is illustrated in Algorithm 3.3 (line 6).

[2] POSIX threads: a multi-threading library on Linux OS.

[3] Mutex: Mutual Exclusion is a synchronization primitive object for multi-threads.

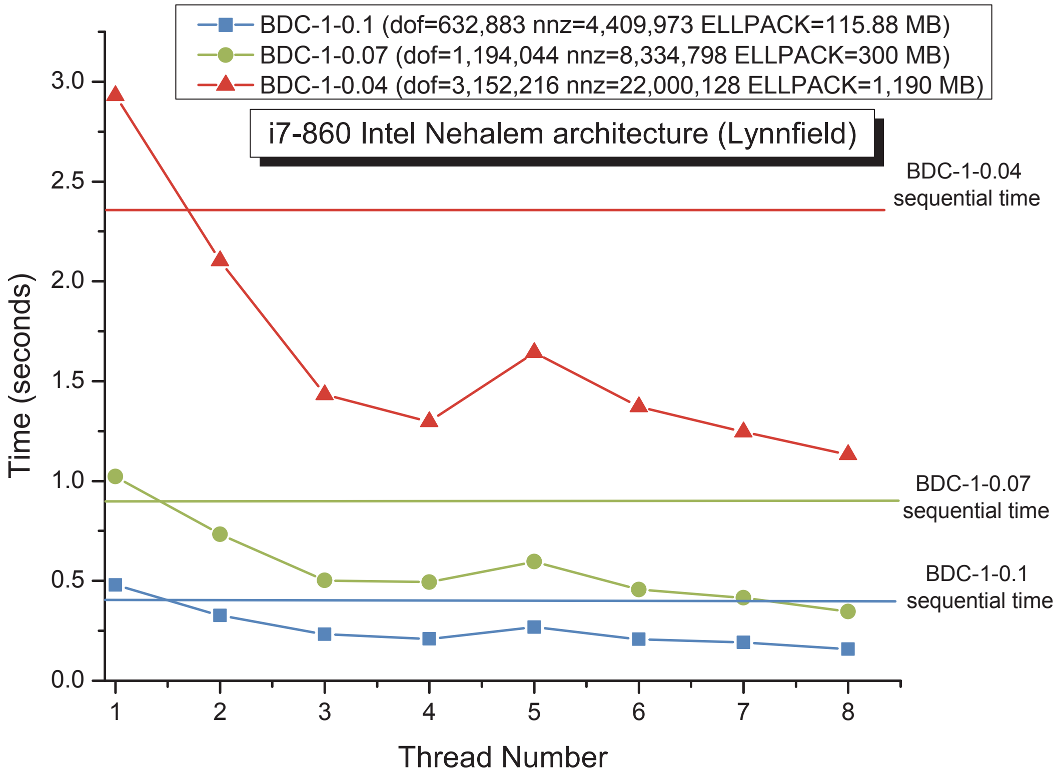


Figure 3.7: Parallel matrix assembly timings in seconds of an Intel quad-core i7-860 processor. The horizontal lines show the sequential runtimes of each matrix. The difference of runtimes between sequential execution and multi-threaded execution using 1 thread highlights the cost of calling *Pthreads* API.

Figure 3.7 and Figure 3.8 show the runtimes in seconds of the parallel assembly of 3 matrices using a first-order nodal finite element formulation on a quad-core Intel i7 processor and an AMD Opteron dual-socket dual-core processor respectively. The sequential runtimes are small (a few seconds) despite the fact that these matrices are considered to be those of realistic average size problems. The runtimes were reduced by more than 50% relative to 1-thread execution when the number of threads was 4. Notice the difference in runtimes between sequential execution (no synchronization) shown in horizontal lines and runtimes of 1 thread. This difference highlights the cost of calling the *Pthreads* API[4] $3 \times n$ times. The overhead of calling a *Pthreads* API although it appears to be large in here, is not the main concern in multi-threaded

[4] API: Application Programming Interface.

applications. Instead it is the wait time that could incur when a thread is waiting for a *mutex* object to be released by another thread. In matrix assembly, this occurs when threads are simultaneously processing mesh elements that share vertices and edges. In the case of FEM, the possibility of threads waiting to acquire a lock is small since the number of shared vertices or edges is low; it related to the average number of non-zeros per row.

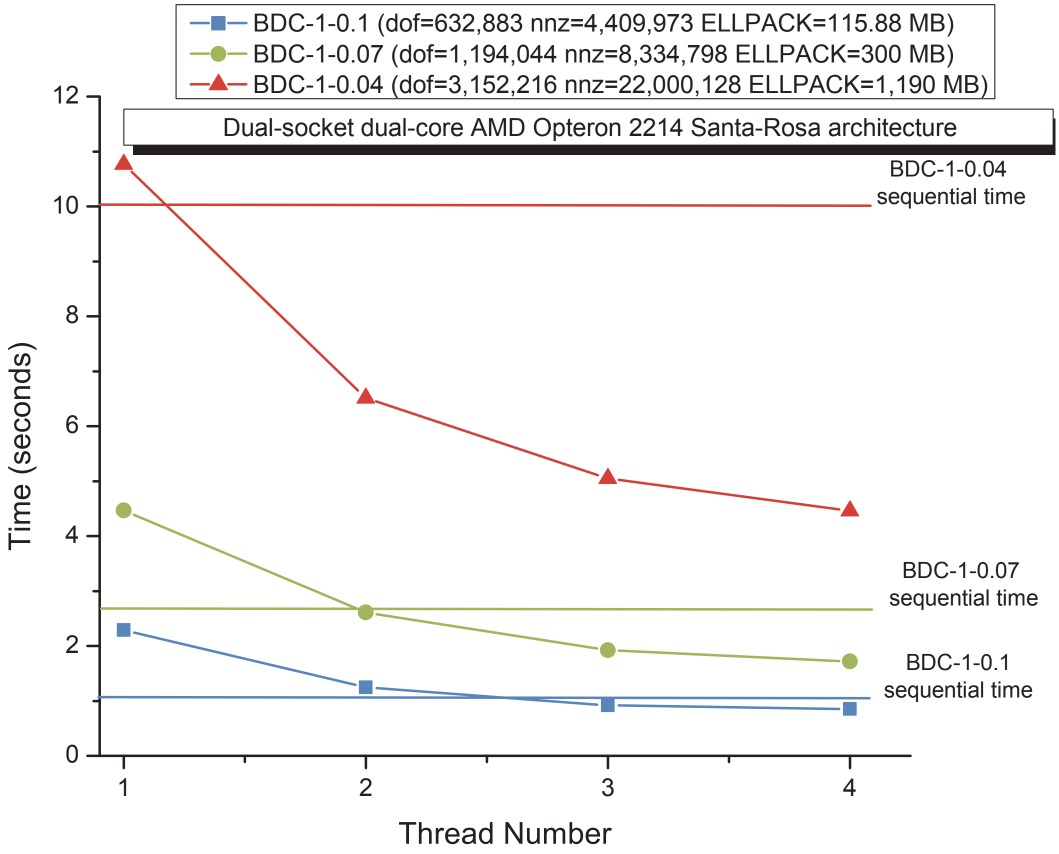


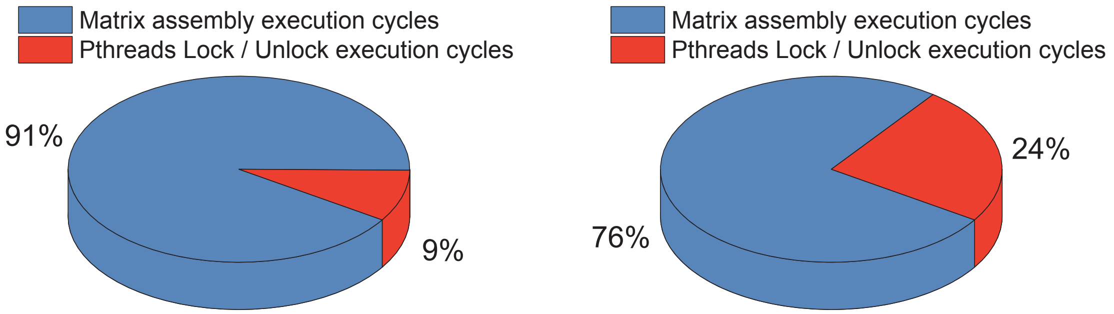
Figure 3.8: **Parallel matrix assembly timings in seconds of an AMD dual-socket dual-core Opteron processor.** The horizontal lines show the sequential runtimes of each matrix. The difference of runtimes between sequential execution and multi-threaded execution using 1 thread highlights the cost of calling *thread* API.

Synchronization and Cache Data Locality The time it has taken to complete the matrix assembly process in the previous experiments was very small (only few seconds), hence, it was not possible to accurately measure the total time spent on synchronization (i.e. calling *Pthreads* API and waiting to acquire a *mutex* lock). Instead, Intel's VTune analyzer was used to count the number of execution cycles spent on synchronization. In the case of matrix assembly using 1 thread, this number constituted around 9% of the total cycles spent on matrix assembly (see Figure 3.9a). This number reflects only the time to call *Pthreads* API, since there was no time or cycles wasted waiting to a acquire a lock (no other threads were competing to acquire a lock). As anticipated, when using 4 threads, more cycles were halted during synchronization, and in this case the percentage of time wasted increased to 24% (see Figure 3.9b).

It is important to note that the cycles counted do not reflect the number of cycles needed to complete a task. A large number of CPU cycles were wasted while the application was waiting to retrieve data from the main memory (DRAM) due to misses on the last level cache (i.e. Level-3 in the case of Intel i7-80 processor). The percentage of wasted cycles was close to 35% when assembling the matrix using sequential code and this number increased to 40% when executing the matrix assembly using 4 threads indicating that the matrix assembly process is not cache optimal. Hence, the synchronization cost of 24% when running 4 threads cannot only be attributed to the fact that threads were waiting to acquire a lock, but also to the fact that more cache misses incurred in this case.

The experiments presented so far were based on assembling a matrix using first-order nodal finite element formulation. In this case and as it will be illustrated in the next section, the average number of non-zeros per row of the final matrix is calculated to be 7. This number reflects the level of interconnections between nodes of the mesh. In other words, it means, that on average each node is connected to 7 other nodes. The higher this number (i.e. 3D case or higher order FE formulation), the more it is expected that threads will wait to acquire a *mutex* lock during matrix assembly.

The remainder of this chapter will list the matrix test sets used throughout this thesis. As explained before, it was important to have a matrix set that is representative



(a) Matrix assembly using 1 thread.

(b) Matrix assembly using 4 threads.

Figure 3.9: **Execution cycles wasted on Pthreads Lock/Unlock synchronization primitives in matrix assembly.** (a) 9% of cycles were spend on calling Pthreads primitives when executing matrix assembly using 1 thread. (b) 24% of cycles were spend on calling Pthreads primitives when executing matrix assembly using 4 threads.

of the type of workload that a design engineer deals with when using field analysis software to design low frequency EM devices.

3.6 Matrix Test Sets

Speedup, scalability and cache performance of multi-threaded algorithms are all main factors that determine the success and usability of a method on a multi-core processor. However, these performance measures depend on the size and the structure of the input problem. This thesis focuses on analyzing the performance of a specific set of problem structures arising from commonly used FEM formulation techniques. Furthermore, the sizes of the input problems were devised to investigate the impact of problem size on cache behavior.

The remaining part of this chapter lists the matrix test sets used throughout this thesis. The first sets (§ 3.6.1) are obtained from applying the *Finite Element Method* on a 2D model of a brushless DC motor (BDC) (Figure 3.10) and a 3D model of a transformer (Figure 3.11). The second set of matrices (§ 3.6.2) is obtained from applying the *Finite Different Method* (FDM) on a parallel plate capacitor problem and finally the last test set (§ 3.6.3) several matrices that have been widely used in recently published research papers which have investigated the performance of sparse matrix-vector multiplication [69, 70, 71, 72, 73].

3.6.1 Finite Element Matrix Test Sets

The matrices in Table 3.3, Table 3.4 and Table 3.5 have been assembled from applying first-order nodal, second-order nodal and edge finite element formulations respectively on the same 2D problem (i.e. a brushless DC motor - BDC). First, the geometry of the BDC model was initially meshed and refined many times by imposing an upper bound on the element size each time. This led to many meshes of different sizes. For each mesh size, matrices were assembled using the three formulations. This is to highlight the changes in the matrix's structure, the matrix's degrees of freedom and the number of non-zeros per row for different formulations of the same problem. The names of the matrices are indicative of their size and the FEM formulation used to assemble them. For instance, BDC-1-0.5 is a matrix generated from applying a first-order nodal finite element formulation on a brushless DC mesh of maximum element size of 0.5 mm. The maximum and average number of non-zeros per row listed for each matrix are important for the analysis of the performance of sparse matrix-vector multiplication (SMVM) preformed in Chapter 5. Also, they serve as an indicator as to whether a matrix is suitable to be stored using the ELLPACK (explained earlier in § 3.3) or the HYBRID sparse storage formats which will be explained in Chapter 5. It is important to note that boundary conditions and material properties were not applied when generating these matrices. We are only interested in the size and the structure of the arising matrices and not their convergence rate.

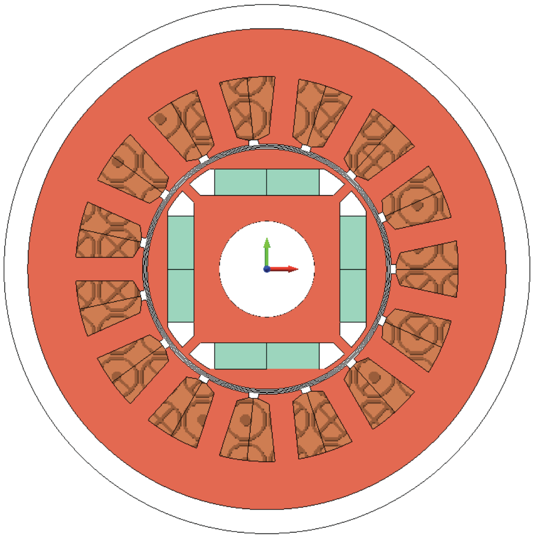


Figure 3.10: **2D brushless DC motor.** The model was obtained from the examples set provided by MAGNET (2D/3D electromagnetic field simulation software by INFOLYTICA, www.infolytica.com) [22].

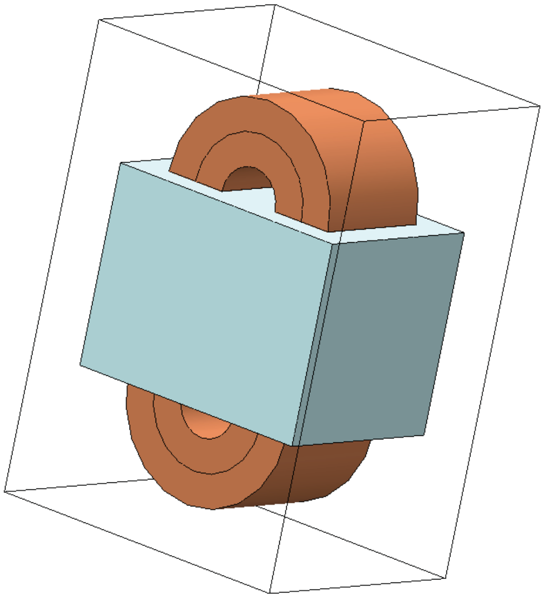
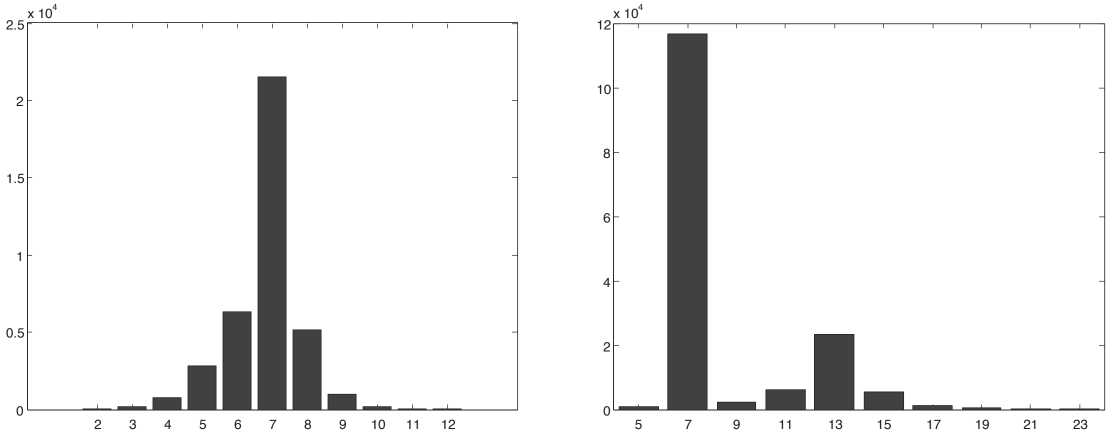


Figure 3.11: **3D transformer**. The model was obtained from the examples set provided by MAGNET (2D/3D electromagnetic field simulation software by INFOLYTICA, www.infolytica.com) [22].

3.6.1.1 Brushless DC Motor Finite Element Formulations

Table 3.3: **Brushless DC Motor - 2D first-order nodal finite element formulation.** The average number of non-zeros per row is 7. The number of non-zeros in the majority of rows is 7. (NNZ = number of non-zeros)

Matrix	Degrees of freedom	NNZ	Average NNZ/row	Maximum NNZ/row	CSR storage size (MB)
BDC-1-0.5	38,084	259,188	7	12	3.1115
BDC-1-0.3	95,518	662,286	7	13	7.9436
BDC-1-0.2	204,570	1,415,366	7	13	16.9741
BDC-1-0.1	632,883	4,409,973	7	16	52.8824
BDC-1-0.07	1,194,044	8,334,798	7	22	99.9391
BDC-1-0.04	3,152,216	22,000,128	7	33	263.7962



(a) The number of non-zeros per row distribution for the matrix BDC-1-0.5.

(b) The number of non-zeros per row distribution for the matrix BDC-2-0.5.

Figure 3.12: **First-order and second-order nodal finite element non-zeros distribution:** Histograms showing the distribution of non-zeros per row for the matrices BDC-1-0.5 and BDC-2-0.5 assembled from the same mesh using first-order and second-order nodal finite element formulations respectively.

Table 3.4: **Brushless DC Motor - 2D second-order nodal finite element formulation.** The average number of non-zeros per row is 9. The number of non-zeros in the majority of rows is 7. (NNZ = number of non-zeros)

Matrix	Degrees of freedom	NNZ	Average NNZ/row	Maximum NNZ/row	CSR storage size (MB)
BDC-2-3.66	48,031	407,733	9	37	4.8494
BDC-2-1	74,479	631,491	9	25	7.5110
BDC-2-0.5	156,669	1,327,737	9	23	15.7924
BDC-2-0.3	384,421	3,259,039	9	25	38.7632
BDC-2-0.2	817,701	6,937,023	9	25	82.5072
BDC-2-0.1	2,537,685	21,540,633	9	31	256.1935
BDC-2-0.07	4,787,651	40,664,669	9	43	483.6336
BDC-2-0.04	12,660,592	107,560,044	9	60	1279.2

Table 3.5: **Brushless DC Motor - triangular finite element edge formulation.** The average number of non-zeros per row is 5 and the maximum number of non-zeros per row is 5.

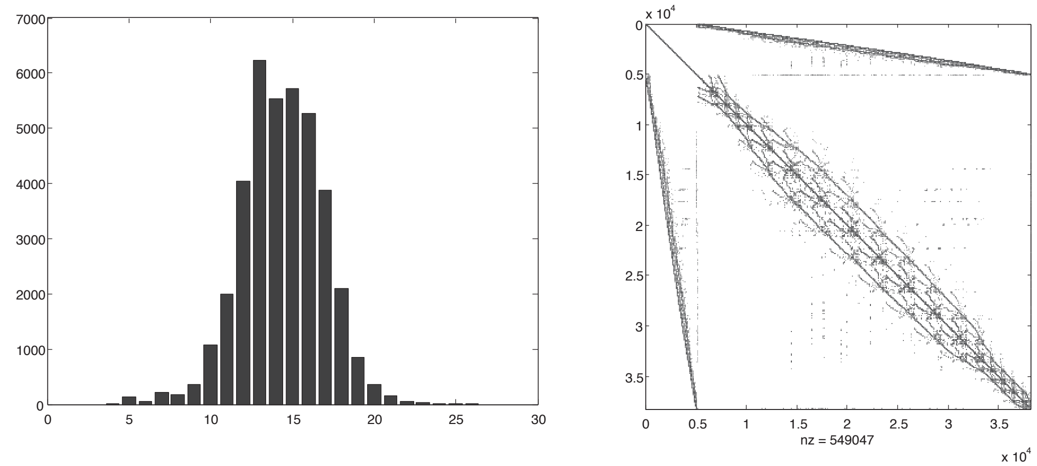
Matrix	Degrees of freedom	NNZ	Average NNZ/row	Maximum NNZ/row	CSR storage size (MB)
BDC-1-1	55,772	278,168	5	5	3.3961
BDC-0-0.5	117,282	584,658	5	5	7.1383
BDC-0-0.3	287,841	1435413	5	5	17.5250
BDC-0-0.2	612,529	3,056,677	5	5	37.31
BDC-0-0.1	1,901,614	9,494,878	5	5	115.9143
BDC-0-0.07	3,589,051	17,931,763	5	5	218.90
BDC-0-0.04	9,492,389	47,437,511	5	5	579.01

3.6.1.2 First-Order Tetrahedral Finite Element Formulation

A 3D transformer model (Figure 3.11) has been meshed by imposing different upper bounds on the element volume to create four different meshes. The first-order finite element formulation is then applied to each mesh. The resulting assembled matrices are shown in Table 3.6. Both the histogram of the number of non-zeros per row and the matrix structure of ET0-0.08 are shown in Figure 3.13. In general, matrices generated from a first-order nodal finite element formulation are more suitable to being stored in ELLPACK than those generated from a higher order nodal finite element formulation.

Table 3.6: 3D transformer - First-order tetrahedral finite element formulation.

Matrix	Degrees of freedom	NNZ	Average NNZ/row	Maximum NNZ/row	CSR storage size (MB)
ET-1-0.08	38,234	549,047	15	26	6.42
ET-1-0.04	409,531	5,999,230	15	31	70.21
ET-1-0.01	1,975,427	28,927,159	15	39	338.58
ET-1-0.01R	2,666,039	39,535,927	15	33	462.62



(a) Histogram of the number of non-zeros per row

(b) Non-zeros sparsity of ET-0.08

Figure 3.13: **Sparsity and non-zeros distribution of ET-0.008.** (a) Sparsity of ET-0.08 and (b) Histogram of the number of non-zeros per row.

3.6.2 Finite Difference Method Matrix Test Set

The matrices shown in Table 3.7 were obtained from applying the *Finite Difference Method* on a problem composed of parallel plate capacitor. The matrices resulting from this method are structured and have higher rate of convergence than the problems obtained from using the *Finite Element Method*. They will be used to investigate a new multi-threaded Gauss-Seidel method for the solution of systems of linear equations (Chapter 4).

Table 3.7: **Parallel plate capacitor - Finite Difference Method**. The maximum and average number of non-zeros per row is 5.

Matrix	Degrees of freedom	NNZ	Average NNZ/row	Maximum NNZ/row	CSR storage size (MB)
FDM-40K	40,000	199,200	5	5	2.4
FDM-122K	122,500	611,100	5	5	7.5
FDM-250K	250,000	1,248,000	5	5	15.2
FDM-1M	1000,000	4,996,000	5	5	61
FDM-4M	4,000,000	19,992,000	5	5	244
FDM-16M	16,000,000	79,984,000	5	5	976
FDM-49M	49,000,000	244,972,000	5	5	2,990

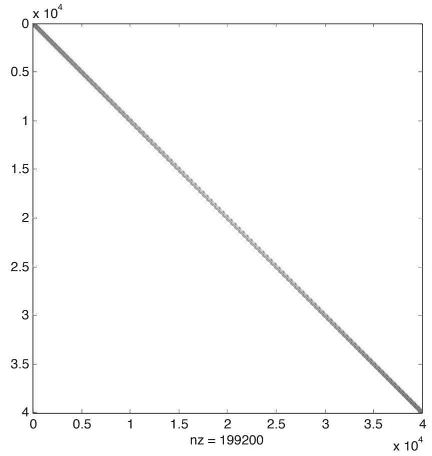


Figure 3.14: **Sparsity of FDM-40k matrix.**

3.6.3 Matrix Market Miscellaneous Test Set

The following table includes a collection of matrices which were obtained from Matrix Market [74]. The matrices have been widely used in recent research papers investigating the performance of SMVM. However, these matrices have different structure than the matrices obtained from applying the *Finite Element Method*. For instance, some of them have high number of non-zeros per row which is advantageous when optimizing SMVM. Furthermore, their sizes (DOF and CSR storage size) are small relative to a realistic problem size.

Table 3.8: Miscellaneous matrix test set.

Matrix	Degrees of freedom	NNZ	Average NNZ per row	Maximum NNZ per row	CSR storage size (MB)
Protein	36,417	4,344,765	120	204	49.86
FEM/Sphere	83,334	6,010,480	73	81	69.1
FEM/Cantilever	62,451	4,007,383	65	78	46
Wind Tunnel	217,918	11,524,432	53	180	132.72
FEM/Harbor(CFD)	46,835	2,374,001	50	145	26.83
FEM/Shipsec	140,874	7,813,404	26	68	41.38
Economics	206,500	1,273,389	7	74	15.36
Epidemiology	525,825	2,100,225	4	4	26
Circuit	170,998	958,936	6	353	11.62

3.7 Concluding Remarks

The matrix assembly process has a time complexity of order $O(n)$ and does not present a bottleneck in the finite element analysis process. The runtime to assemble a matrix of a large size problem takes only few seconds and this time was reduced by 40% when using 4 threads.

The granularity of parallel matrix assembly can be considered to be medium. Threads need only to synchronize access to a global matrix structure, but do not depend on each other's data. The level of interdependency between threads depends on the finite element formulation used to assemble the matrix and on the problem dimension (e.g. 2D or 3D). The higher the dimension of the problem and the higher the order of the finite element formulation, the more matrix rows are interconnected. However, in general, this dependency is low in matrices obtained from using the *Finite*

Element Method and it can be quantified by the average number of non-zeros per row and the maximum number of non-zeros in a row of the final matrix.

The work hereinafter focuses on the solver part of the *Finite Element Method*. Particularly, a new multi-threaded Gauss-Seidel technique is presented in the next chapter (Chapter 4) followed by two chapters which investigate sparse matrix-vector multiplication (Chapter 5) and Conjugate Gradient preconditioning techniques (Chapter 6) respectively.

Chapter 4

Sliding-Window Gauss-Seidel as a Solver for Sparse Systems of Linear Equations

Contents

4.1 Overview

Iterative solvers aim at solving a system of equations $Ax = b$ (linear or non-linear) where A is a large sparse matrix and x is the unknown vector to be calculated. They do so by iterating a large number of times over the matrix A, hence a main goal of research on iterative solvers has been to reduce the number of iterations required to reach convergence (i.e. increase its convergence rate). The introduction of the Preconditioned Conjugate Gradient (PCG) method [75] - precisely the incomplete Cholesky

Conjugate Gradient (ICCG[1]) - has been a turning point in iterative solver techniques, especially when it was widely adopted on microprocessors as a fast sequential iterative solver. Despite that, the study of *stationary* iterative methods such as Jacobi, Gauss-Seidel (GS) and Successive Over-Relaxation (SOR) remained an interesting field in scientific computing. The introduction of many massively parallel multi-processor machines in the late 1970's and early 1980's rejuvenated the work in *stationary* methods to explore their parallelism. The advances in the microprocessor technology, where the speed was doubling every 18 months [76], and the introduction of the multi-grid method [77], where Gauss-Seidel is a basic building block of this method (smoothing operator) has attracted considerable research aimed at optimizing Gauss-Seidel on microprocessors.

This chapter investigates a new cache efficient multi-threaded iterative solver which combines Jacobi and Gauss-Seidel. On a multi-core processor, *stationary* based iterative solvers such as Jacobi and Gauss-Seidel are appealing to use during the optimization process. For instance, many optimization techniques analyze the same design using different design parameters. A solution obtained from one design instance can be reused as the initial solution of another design instance with slightly different parameters.

Stationary iterative solvers are also useful in design approaches that use *Case Based Reasoning* (CBR) [78, 79, 80]. In CBR, a database of previous design solutions is searched for one or more that closely matches the problem requirements. The set of solutions are then adapted so that a new design is created that matches the problem requirements. Furthermore, since a *stationary* iterative solver smooths and dampens the error after each iteration, the design engineer can view the convergence towards the solution at each step. This real-time solution tracking can assist the design engineer in making a decision as whether to continue or abandon a running field analysis instance.

[1] ICCG: a Preconditioned Conjugate Gradient Method which uses a preconditioner, referred to as incomplete Cholesky (IC), that has the same sparisty as the original matrix.

Cache Optimization in Iterative Solvers Since iterative solvers iterate a large number of times over the system of equations $Ax = b$, a large number of cache misses are usually generated. A small reduction of the number of cache misses in each iteration greatly reduces the solver's runtime. This is especially true on a multi-core processor since the gap between the slow data access due to a small cache relative to the high number of flops available is higher on chip multi-core processor (CMP) than other hardware architectures. Therefore, efficient cache management is critical to achieve better performance.

At this point, it is worth identifying the three types of cache misses that could occur on a microprocessor or a multi-core processor:

- **Compulsory misses, intrinsic misses or cold misses** are all terms to refer to the misses that are due to the first access to data. The size of the cache and the associativity have no effect. However, data prefetching either explicitly by the application, or implicitly by the processor reduces this type of cache miss.

- **Conflict misses (cross interference)** are misses that depend on the cache-memory associativity and the cache replacement policy. When two data are associated with the same cache line, early eviction of the cache line could occur.

- **Capacity misses (self interference)** are misses that are due to the size of the working data set relative to the cache size. They decrease with increased cache size. When cache associativity increases, capacity misses will increase however conflict misses will decrease.

Furthermore, on a multi-core based processor, two additional types of cache misses could occur. The first is referred to as *true cache sharing* wherein, two cores modifying the same data will invalidate each other's private caches (L1 and L2 in the case of Intel i7-860 processor). Cache invalidation can also happen when two cores are accessing different data that happen to be on the same cache line in what is known as *false cache sharing*.

The work on optimizing iterative solvers has been on reducing the amount of compulsory cache misses that are due to the irregular access to x in $Ax = b$ especially in the case when A is an unstructured matrix. Most of the work in the literature on sequential and parallel Gauss-Seidel deals with structured problems arising from the *Finite Difference Method* where cache optimization techniques such as loop tiling and reordering have been applied [81, 82, 83, 84, 85]. Whereas this chapter's work experiments with both structured and unstructured matrices. It is important to note that optimization to reduce cache misses used in SMVM and discussed in the next chapter are also applicable to Gauss-Seidel since both have very similar operations.

4.1.1 Parallel Gauss-Seidel Techniques

GS and SOR are inherently sequential methods since the calculation of each component depends on the latest update of other dependent components. However, the Jacobi method is highly parallelizable since each component of the solution vector x when solving $Ax = b$ can be calculated simultaneously with the other components. Nevertheless, GS and SOR are favored over Jacobi since they have higher convergence rate.

Techniques to explore parallelism in GS and SOR can be divided into two broad categories. The first explores parallelism in Gauss-Seidel by identifying the nodes of the mesh, grid, or elements of the solution vector x that are independent of each other. Parallel Jacobi is then applied on each set of independent nodes (i.e. simultaneously solving all nodes within a set). This parallel technique preserves the order of update on the x vector leading to results similar to sequential Gauss-Seidel. One way to label independent nodes is to use coloring techniques, which assign the same color to independent nodes. When $Ax = b$ arises from a five-point finite difference approximation, then two colors (e.g. red-black) are used to group independent nodes [86, 87]. Nodes of the same color are then divided on a multiprocessor (SIMD[2] or MIMD[3]) where Jacobi style sweeps are executed on each node. A second sweep is required to handle the black color. The 2 sweeps are equivalent to one Gauss-Seidel iteration. For higher order discretization, more colors are needed, thus increasing the number of sweeps required per iteration. Synchronization and communication is required after each sweep. For a review on multi-color SOR on both vector and multiprocessors refer to [88].

The second category of techniques is a combination of Jacobi and Gauss-Seidel, hence the end result does not lead to the same number of updates or order of updates as the original Gauss-Seidel. The goal of these techniques is to balance the benefit of Jacobi and Gauss-Seidel. On a multi-processor machine, equations or nodes are divided into blocks that are distributed on processors. Each processor executes Gauss-Seidel on its working set, while blocks synchronize results at the end of each iteration. In order to reduce the number of synchronization points and communication overheads, asynchronous parallel Gauss-Seidel schemes have been introduced. In these techniques, a processor when executing the i^{th} iteration can use a value of x calculated by other processors in the $i^{th} - s$ iteration, where $1 \leq s \leq k$ and k is a predefined fixed integer. In the specific case when $k = 1$, this method would be identical to Jacobi's, wherein the solutions of all processors are synchronized following each iteration. A survey on these techniques can be found in [89]. Pipelining thread techniques have been used to maintain data locality [90, 91] on Symmetric Multiprocessors machines (SMP). A survey of the implementation of the above techniques on vector and arrays of processors can be found in [92].

4.2 Sliding-Window Gauss-Seidel Methodology

In order to increase the convergence of Gauss-Seidel and maximize data reuse, a new multi-threaded Gauss-Seidel method has been developed in this thesis's work, which is

[2] SIMD: Single-Instruction Multiple-Data.

[3] MIMD: Multiple-Instructions Multiple-Data.

called *"Sliding-Window Gauss-Seidel"* (SW-GS). This novel method has been recently utilized by another research group that modified it with the aim of better fitting a GPU based processor for the purpose of image processing [93]. It has been particularly useful on a GPU since it reduces the amount of costly memory transfers from-and-to the GPU. Furthermore, the author of [94] improved the performance of SW-GS through register data reuse and loop unrolling techniques. Finally, it is important to mention that optimization techniques which are applied on SMVM (e.g. cache blocking) and discussed in the next chapter are applicable to SW-GS.

SW-GS shares many commonalities with the pipelined parallel Gauss-Seidel developed for shared-memory multiprocessors [91] (Sequent Balance 21000[4]). Both methods will be explained and the differences will be highlighted.

Let $EW_{(u,v)}$ (*Execution Window*) be a set of $W = v - u$ contiguous rows of the matrix A, where u and v represent the indices of the first and last row of this working set respectively. Let $GaussSeidel(EW_{(u,v)})$ be the terminology to represent applying Gauss-Seidel's Equation 4.1 on the rows of $EW_{(u,v)}$.

$$x^i_{k+1} = \frac{1}{a_{ii}}(b_i - (\sum_{j=u}^{i-1} a_{ij}x^j_{k+1} + \sum_{j=i+1}^{v} a_{ij}x^j_k)) \quad i = u, ..., v. \tag{4.1}$$

Using this terminology, the sequential Gauss-Seidel represented in Equation 4.2 can be written as $GaussSeidel(EW_{(1,N)})$, where N is the degrees of freedom (DOF) of A. Furthermore, the sequential Gauss-Seidel is equivalent to the successive execution of Gauss-Seidel on contiguous, non-overlapping k blocks of rows each of size W; $EW_{(1,W)}$, $EW_{(W+1,2W)}$, ..., $EW_{(kW+1,N)}$ as illustrated in Figure 4.1 (Gauss-Seidel).

$$x^i_{k+1} = \frac{1}{a_{ii}}(b_i - (\sum_{j=1}^{i-1} a_{ij}x^j_{k+1} + \sum_{j=i+1}^{n} a_{ij}x^j_k)) \quad i = 1, ..., n. \tag{4.2}$$

Sliding-Window Gauss-Seidel is an extension of the original Gauss-Seidel in which the

[4] Sequent Balance 21000: 12-processor 16 Mbyte shared-memory multiprocessor released in 1986.

Execution Window $EW_{(u,v)}$ of size W rows, slides by S rows where $S <= W$, that is, if a thread executes $GaussSeidel(EW_{(u,v)})$ at iteration k, the same thread will execute $GaussSeidel(EW_{(u+S,v+S)})$ at iteration $k+1$ as shown in Figure 4.1 (pass # 1 of Sliding-Window-Gauss-Seidel). Furthermore, a thread does not view the matrix as starting at row 1 and ending at row N, but rather a continuous circular buffer of rows. Thus, an *Execution Window* can include rows from both the end and the beginning of the matrix. When $W > S$, a thread re-iterates over part of the rows again so as to lower the cache misses that are due to capacity misses and reuse the entries in the matrix A while present in the cache. In this case, a single pass over the system of linear equations using SW-GS updates each element of the solution vector x W/S times, leading to a lower number of iterations for the system to converge, at the cost of increasing execution time per iteration. The choices of W and S should be balanced so that the total working set present at any time fits the cache size. This depends on the storage used by W rows and the number of threads. It can be noted that when $W = S$, each element x is updated once per iteration, which is the case of a regular Gauss-Seidel.

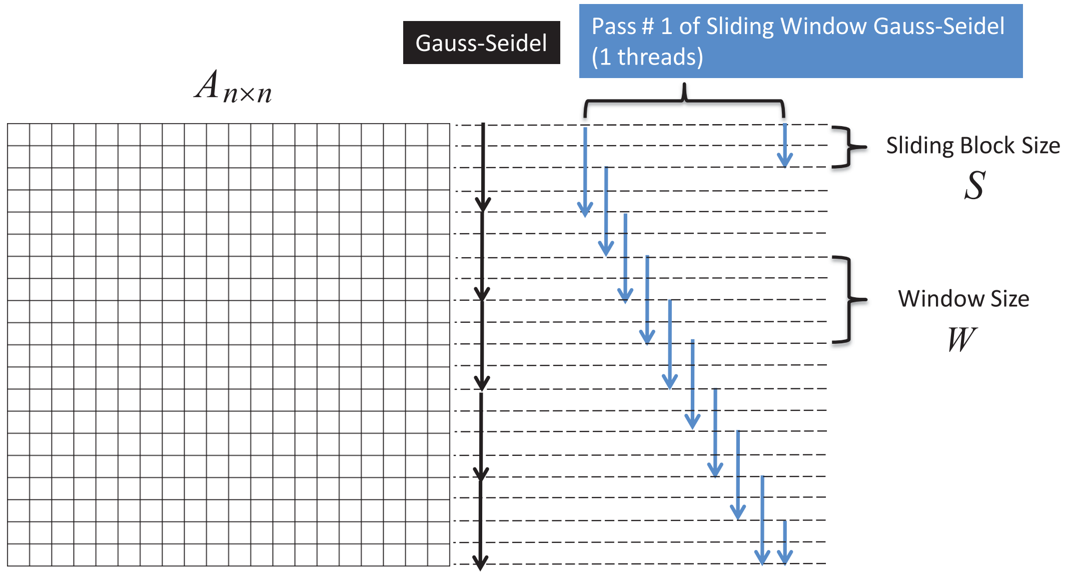


Figure 4.1: One thread Sliding-Window Gauss-Seidel.

4.2.1 Multiple Threads Sliding-Window Gauss-Seidel

An iteration of a multi-threaded SW-GS proceeds in the following way. Initially, available threads are assigned to contiguous, non-overlapping *Execution Windows* as shown in step 1 of Figure 4.2, where four threads T_4, T_3, T_2 and T_1 are assigned to $EW_{(1,W)}$, $EW_{(W+1,2W)}$, $EW_{(2W+1,3W)}$ and $EW_{(3W+1,4W)}$ respectively. Each thread executes Gauss-Seidel on its corresponding working set and stores its solution in a local vector *xLocal* of size W. During simultaneous Gauss-Seidel execution, all threads have read-only access to a global solution vector *xGlobal* which holds the solution from the previous step $k-1$. When a thread finishes executing Gauss-Seidel, it waits at a barrier for the remaining threads to finish executing before proceeding to the synchronization stage. In this stage, all threads update the global solution vector by mapping their partial solution *xLocal* to their corresponding location in the global vector. Once the synchronization step is completed, all threads slide their working set by S rows to perform step $k+1$. For instance, T_1 which executed $GaussSeidel(EW_{(1,W)})$ at step k, proceeds to execute $GaussSeidel(EW_{(u+S,v+S)})$ at step $k+1$.

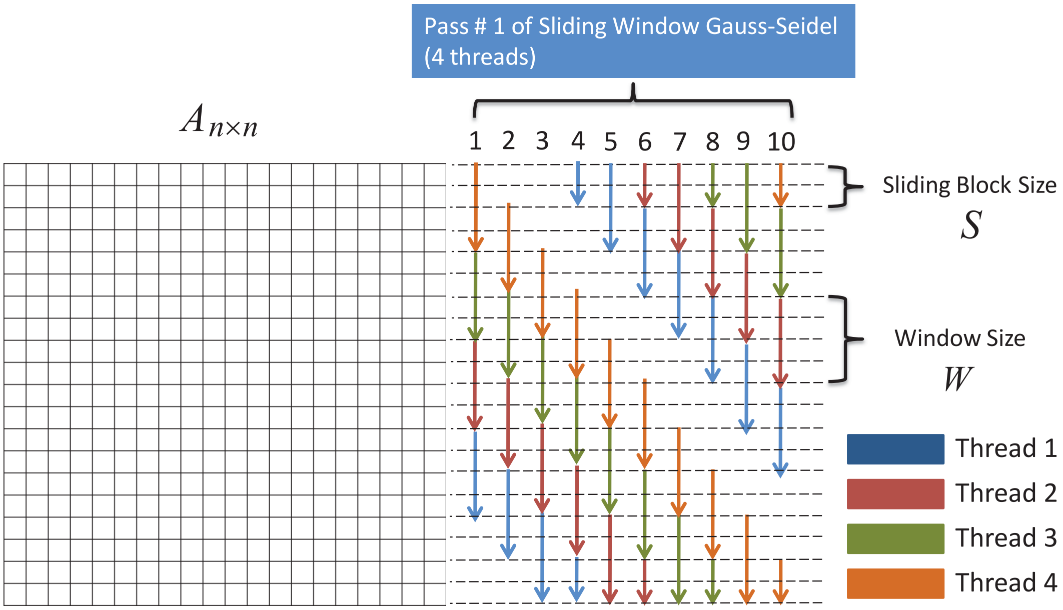


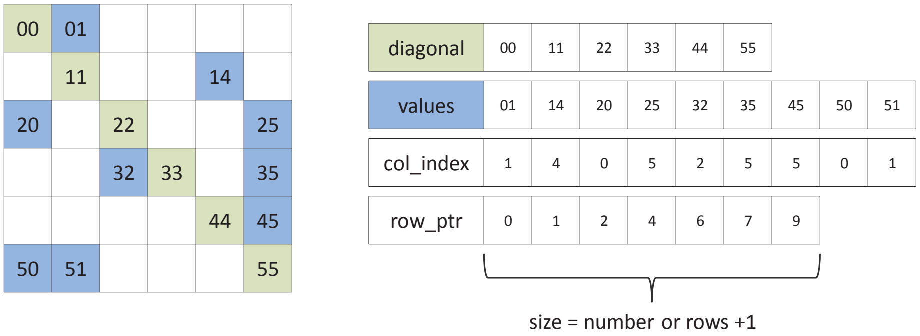
Figure 4.2: Four threads Sliding-Window Gauss-Seidel.

This pipelining of threads has proven to be effective on a SMP machine [91]. The difference between SW-GS and the one developed by [91] is that in the latter, each process holds its own solution vector of size N where synchronizations between all copies of the solution vectors takes place whenever a new value of a solution x^i is calculated. This poses significant synchronization overhead as it requires N synchronizations for each iteration and performs T updates on each element of x, whereas SW-GS requires N/S synchronizations for each iteration and performs $T \times (W/S)$ updates on x. Both methods are similar when $W = S = 1$. When $W > S > 1$, SW-GS becomes a combination of Jacobi (inter-thread) and Gauss-Seidel (intra-thread). Gauss-Seidel Equation 4.1 can be re-written to reflect such a change as

$$x^i_{k+1} = \tfrac{1}{a_{ii}}(b_i - \sum_{\substack{j=u \\ j \neq i}}^{v} a_{ij} xLocal^j_{k+1} + \sum_{j=1}^{u-1} a_{ij} xGlobal^j_k + \sum_{j=v+1}^{v} a_{ij} xGlobal^j_k) \qquad i = u, \ldots, v. \tag{4.3}$$

4.2.1.1 Multi-Thread Sliding-Window Gauss-Seidel Implementation

When calculating a new value x^i_{k+1} using Gauss-Seidel (i.e. Equation 4.2), a vector inner-product of all entries of row i of matrix A (i.e. $A(i, -)$) and all elements of the solution vector x_k is required, except for the diagonal entry $A(i, i)$. To avoid using a conditional *if* statement to exclude the diagonal entry when performing the inner-product, the original CSR storage scheme has been modified to store the diagonal entries in a separate array (i.e. *diagonal*) from the array that stores the non-zero elements (i.e. *values*) as shown in Figure 4.3.



diagonal	00	11	22	33	44	55			
values	01	14	20	25	32	35	45	50	51
col_index	1	4	0	5	2	5	5	0	1
row_ptr	0	1	2	4	6	7	9		

Figure 4.3: **Modified CSR storage scheme.** The diagonal entries are stored separately from the non-diagonal entries of A.

Despite the optimization discussed above, multi-threaded SW-GS has implementation drawbacks. Algorithm 4.1 shows the execution steps followed by each thread, however for illustration purpose, it does not detail the implementation of using CSR storage, but assumes a dense storage. Only the diagonal entries are stored in a separate array to highlight the optimization discussed in the previous paragraph. It can be seen that two synchronization barriers are needed. The first is when all threads wait after finishing executing Gauss-Seidel (line 22) so that they update the global solution from their local copy of the solution. The second is when threads modify their working set (i.e. W and S) (line 24). Furthermore, line 13 shows that there is a conditional *if* statement in the inner loop of the inner-product. Furthermore, since SW-GS is a combination of Gauss-Seidel and Jacobi, threads access solution elements either from a previous iteration $xGlobal_{k-1}$ or solution elements from the partial solution vector $xLocal_k$ according to the modified Gauss-Seidel Equation 4.3. This adds a conditional *if* statement in the inner loop of the inner-product as shown in line 13 of Algorithm 4.1.

4.2.2 SW-GS Floating-Point Operations

The number of floating-point operations of a single iteration of sequential Gauss-Seidel is 2 × number of non-zeros (NNZ), where $2 \times (NNZ - N)$ are performed on the sum

Algorithm 4.1 Execution flow of a thread in SW-GS showing two synchronization barriers.

```
$Ax = b$: system of linear equations to be solved.
$N$: the degrees of freedom of $Ax = b$.
$E_{(i,j)}$: the execution window of Gauss-Seidel.
$W$: the number of rows in $E_{i,j}$.
$S$: the sliding window size.
$xLocal$: array of size $W$.
$xGlobal$: array of size $N$.
$diag$: array which stores the diagonal elements of matrix $A$.
repeat
    for $row = i \rightarrow j$ do
        $a_{ii} \leftarrow diag(i)$
        for $column = 1 \rightarrow n$ do
            if  $column < i$ or $column > j$ then
                $x \leftarrow xGlobal(column)$
            else
                $x \leftarrow xLocal(column \bmod i)$
            end if
            $sum \leftarrow sum + x \times a_{ij}$
        end for
        $xLocal(row \bmod i) \leftarrow \left(\frac{b(i)-sum}{a_{ii}}\right)$
    end for
    $\boxed{\textbf{Gauss Seidel barrier}}$
        $xGlobal \leftarrow xLocal$
    $\boxed{\textbf{Synchronization barrier}}$
        Increase $i$ and $j$ by $S$ rows
until  $r = A \times xGlobal/b \leq error\_tolerance$
```

part of Equation 4.2, and $2 \times N$ operations are to subtract the right hand side vector and to divide by the diagonal entries. In the case of SW-GS, the number of floating-point operations per iteration depends on the window size, block size and the number of threads. It can be easily verified that this number is given by Equation 4.4.

$$FLOP_{iteration} = T \times (W/S) \times (2 \times NNZ) \tag{4.4}$$

4.3 Multi-threaded Sliding-Window Gauss-Seidel: Scalability and Convergence

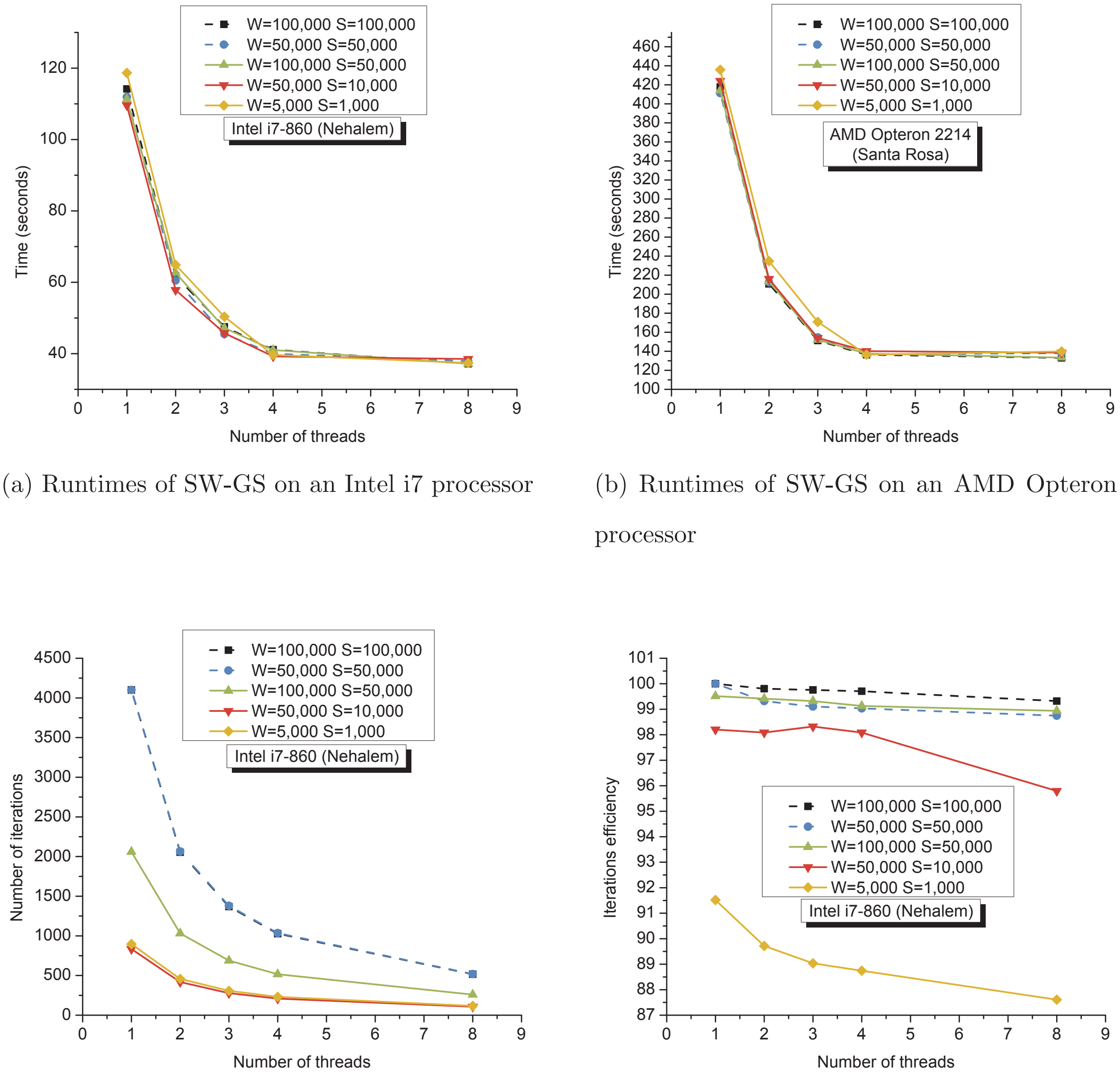
A number of experiments have been carried out on the FDM-1M, FDM-40K and BDC-1-0.1 matrices. The first two matrices were obtained from a finite difference problem (Chapter 3 - Table 3.7). FDM-1M (DOF=1,000,000 NNZ=4,996,000) occupied 61 MB when stored in CSR format and FDM-40K (DOF=40,000 NNZ=199,200) occupied 2.4 MB (i.e. it fits in the last level cache of the processors used in the experiments). On the other hand, BDC-1-0.1 (DOF=632,883 NNZ=4,409,973) is a non structured matrix obtained from a finite element problem and occupied 51 MB (Chapter 3 - Table 3.3).

Overall, the use of a small window size W and a sliding-window S where $S < W$ did not reduce the execution times of SW-GS relative to the execution time when using $W = S$. Although using $W < S$ decreased the amount of capacity cache misses relative to using $W = S$, synchronization overheads and convergence inefficiency resulting from using $W < S$ over-shadowed the performance gain due to the reduction of cache misses. The following experiments in this section will illustrate these results.

4.3.1 FDM-1M and FDM-40K Experiments

4.3.1.1 FDM-1M

Multi-threaded SW-GS was first applied on FDM-1M (DOF=1,000,000 CSR=61MB) by using different values of window sizes (W) and block sizes (S) which were empirically chosen. Runtime results on both an Intel i7-860 processor (Figure 4.4a) and an AMD Opteron 2214 dual-socket dual-core processor (Figure 4.4b) have shown that there were no significant differences in performance relative to the choices of different values of W and S.



(a) Runtimes of SW-GS on an Intel i7 processor

(b) Runtimes of SW-GS on an AMD Opteron processor

(c) Number of iterations

(d) Iterations efficiency

Figure 4.4: **FDM-1M Sliding-Window Gauss-Seidel execution times.** Sliding-Window Gauss-Seidel of FDM-1M (DOF=1,000,000 NNZ=4,996,000 CSR size=61 MB). Relative residual error = 10^{-3}.

A further investigation of these results revealed the reason of the similar performance of SW-GS experiments despite using different values of windows sizes W and block sizes S. Figure 4.4c shows the total number of iterations by all threads for every experiment. It has been noted that when $S \ll W$, the total number of passes over the system of linear equations was much lower than the number of passes when $W = S$, hence reducing capacity cache misses. However, it is important to note that the reduction of the number of passes does not mean that convergence of SW-GS was better when $S < W$ than when using $S = W$ or better than the convergence of a sequential Gauss-Seidel, in fact, it is the opposite. Reducing the block size S relative to the window size W and increasing the number of threads led to a method that is a combination of Gauss-Seidel and Jacobi. The number of updates that a SW-GS performs on each element of the solution vector x depends on the number of threads and the sizes of W and S. For i iterations, the total number of updates on each entry of x, u_{SW-GS}, can be calculated using Equation 4.5.

$$u_{SW-GS} = i_{SW-GS} \times T \times \frac{W}{S} \tag{4.5}$$

Since SW-GS is a combination of Gauss-Seidel and Jacobi, its convergence is always less than that of a sequential Gauss-Seidel, from which it can be deduced that SW-GS would require, at least, to update the solution vector x as many times as a sequential Gauss-Seidel in order to reach the same solution; i.e. $u_{SW-GS} \geq u_{GS}$. One way to quantify the convergence efficiency of SW-GS relative to a sequential Gauss-Seidel is to divide the total number of updates on the vector x by Gauss-Seidel to the number of updates of SW-GS on x for a given solution error tolerance which I will refer to as "Iterations efficiency" and is given in Equation 4.6.

$$\xi_{iteration} = \frac{u_{GS}}{u_{SW-GS}} \times 100\% \tag{4.6}$$

In the case of a sequential Gauss-Seidel which is equivalent to a Sliding-Window Gauss-Seidel ($T = 1$ and $W = S$), the number of updates u_{GS} is equal to the number of iterations i_{GS}, i.e. $u_{GS} = i_{GS}$. Equation 4.6 can be rewritten as

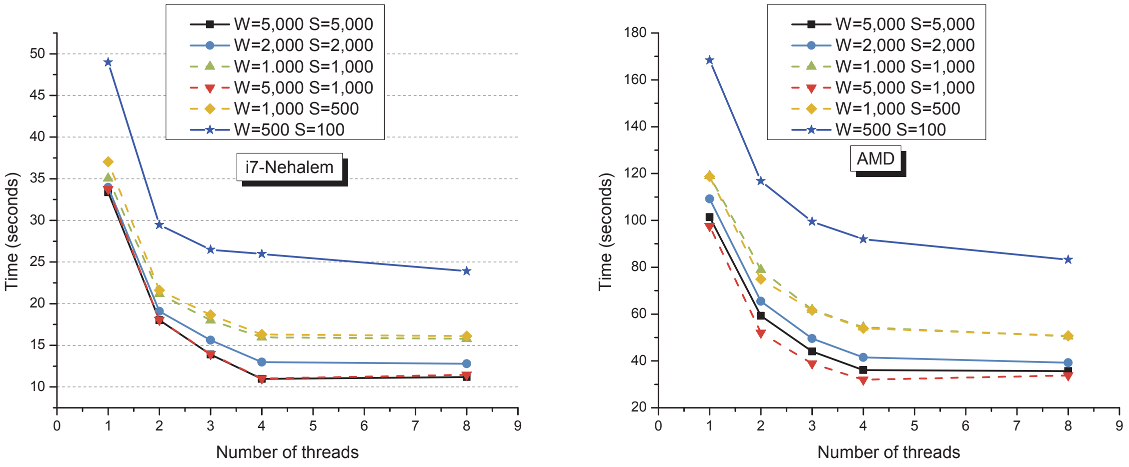
$$\xi_{iteration} = \frac{i_{GS}}{i_{SW-GS} \times T \times \frac{W}{S}} \times 100\% \tag{4.7}$$

For the same FDM-1M problem, the efficiency of iterations using different values of windows sizes (W) and block sizes (S) are shown in Figure 4.4d. Despite the fact that those experiments resulted in a similar execution runtimes (Figure 4.4a), their iteration efficiencies were different (ranging between ≈ 87% and ≈ 100%). The lower a window size W and the higher the ratio $\frac{W}{S}$, the less the iteration efficiency was. An example of this observation is for the experiments (W=5,000 S=1,000) and (W=50,000 S=10,000) where both had $\frac{W}{S} = 5$. The fact that these experiments had such a low iteration efficiency and similar execution times to all other experiments can only lead to the following conclusion: using a smaller window size (W)and higher $\frac{W}{S}$ ratio reduces capacity misses by reducing the number of the overall iterations over the coefficient matrix A. However, this cache locality enhancement is canceled by two sources of execution overheads. The first is related to the convergence of SW-GS (which is related to the values of W, S and the number of threads). The iteration efficiency is an indicator of the convergence properties of SW-GS. The second execution inefficiency is due to the fact that more synchronization barriers are executed in SW-GS for lower values of W and higher $\frac{W}{S}$ ratio. The number of synchronization barriers executed for i SW-GS passes is given in Equation 4.8 which is similar to the number of updates on an element of the solution x (Equation 4.5).

$$nbarriers_i = 2 \times i \times T \times \frac{W}{S} \tag{4.8}$$

4.3.1.2 FDM-40K

To further validate the conclusion of the effect of W and S on cache data locality and convergence, SW-GS was applied on FDM-40K. This matrix has a CSR storage size (CSR=2.4 MB) smaller than the size of the last level cache of an Intel i7-860 processor and an AMD Opteron processor. Since the matrix fits in the cache, no capacity misses occurred and there was no advantage from using SW-GS with low values of W and high $\frac{W}{S}$ ratio. For example, in the case of $W = 500$ and $S = 100$, the efficiency per iteration of SW-GS was lower and the number of synchronization barriers were higher than when using larger window size W and higher $\frac{W}{S}$ ratio. This explains the end results which showed a discrepancy in the execution times as shown in Figure 4.5 for Intel i7-860 and AMD Opteron processors.



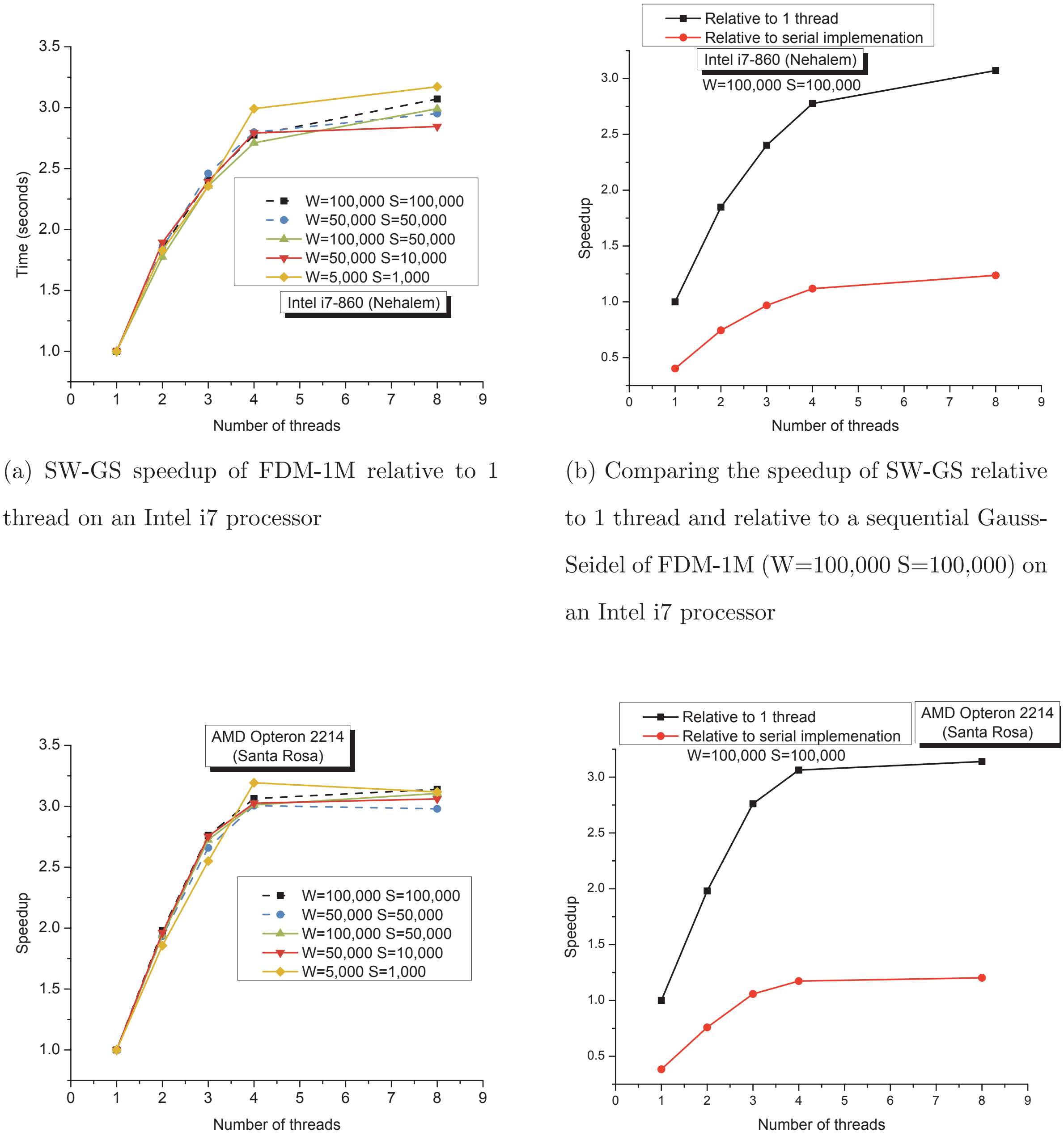
(a) Runtimes of SW-GS on an Intel i7 processor

(b) Runtimes of SW-GS on an AMD Opteron processor

Figure 4.5: FDM-40K Sliding-Window Gauss-Seidel execution times. Sliding-Window Gauss-Seidel of FDM-40K (DOF=40,000 NNZ=199,200 CSR size=2.4 MB). Relative residual error = 10^{-6}.

4.3.1.3 FDM-1M Scalability

The scalability of SW-GS was also investigated for the FDM-1M matrix . The speedup of SW-GS when compared to 1 thread execution showed scalability on both an Intel and AMD processor as shown in Figure 4.6a and Figure 4.6c respectively. However, the speedup of a SW-GS when compared to a sequential implementation of Gauss-Seidel was lower than when compared to a single thread SW-GS. The fact that a multi-threaded SW-GS performs poorly relative to a sequential Gauss-Seidel highlights the overheads of synchronization. The magnitude of this overhead is illustrated in Figure 4.6b (Intel i7-860) and Figure 4.6d (AMD Opteron 2214) by showing SW-GS speedup graph when calculated relative to a 1 thread and relative to a sequential Gauss-Seidel.



(a) SW-GS speedup of FDM-1M relative to 1 thread on an Intel i7 processor

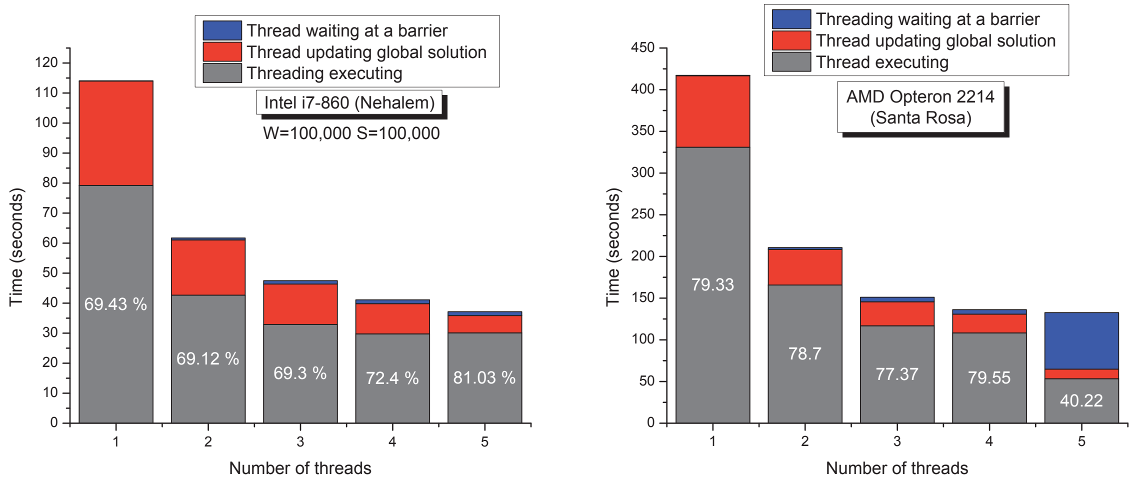
(b) Comparing the speedup of SW-GS relative to 1 thread and relative to a sequential Gauss-Seidel of FDM-1M (W=100,000 S=100,000) on an Intel i7 processor

(c) SW-GS speedup of FDM-1M relative to 1 thread on an AMD Opteron processor

(d) Comparing the speedup of SW-GS relative to 1 thread and relative to a sequential Gauss-Seidel of FDM-1M (W=100,000 S=100,000) on an AMD processor

Figure 4.6: **FDM-1M speedup of Sliding-Window Gauss-Seidel.** Speedup of FDM-1M (DOF=1,000,000 NNZ=4,996,000 CSR size=61 MB) on an Intel i7 and an AMD Opteron processors. The high cost of synchronization is illustrated in (b) and (d).

The source of synchronization overheads in a multi-threaded application depends on two factors. The first is due to the amount of data to be synchronized and exchanged between threads and the second is due to the time a thread spends waiting at synchronization barriers. The cost of these overheads has been calculated for the case of FDM-1M ($W = 100,000$ and $S = 100,000$) and is reported in Figure 4.7. the grey box shows the percentage of time SW-GS is executing, the blue box shows the percentage SW-GS is spending synchronizing data among threads and finally, the red box shows the time SW-GS threads are waiting at a barrier. As anticipated, most of the overhead in synchronization was due to data synchronization. The fact that there were no overheads due to threads waiting at a barrier indicates that balanced thread execution (i.e. data are partitioned equally among threads). One exception to this observation is the fact that there was barrier overhead when running 8 threads on a quad-core AMD Opteron processor due to the limited thread resources (i.e. 4 threads). Although, the Intel i7-860 has physically 4 cores, its *Hyper-Threading* technology made it possible to run 8 threads.



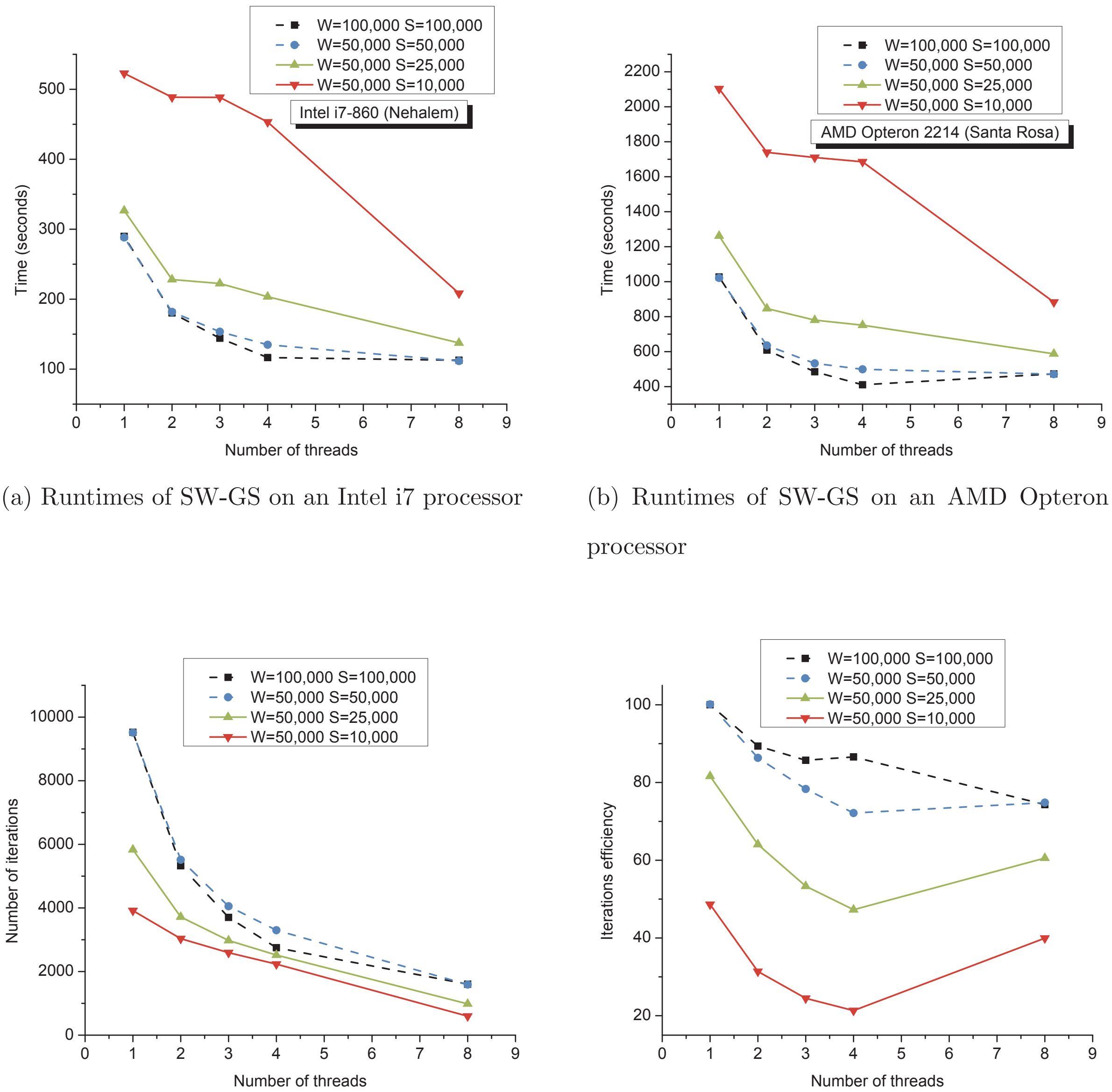
(a) Thread execution efficiency of FDM-1M (W=100,000 S=100,000) on an Intel i7 processor

(b) Thread execution efficiency of FDM-1M (W=100,000 S=100,000) on an AMD Opteron processor

Figure 4.7: **FDM-1M: SW-GS threads utilization.** The percentage of runtime SW-GS is spending on thread execution and synchronization (FDM-1M DOF=1,000,000 NNZ=4,996,000 CSR size=61 MB).

4.3.2 BDC-1-0.1 Experiments

SW-GS was applied on BDC-1-0.1 (DOF=632,883 NNZ=4,409,973 CSR=52.88 MB), which is a non-structured matrix having the same size as FDM-1M (CSR=61 MB) analyzed in the previous section. In both cases (i.e. BDC-1-0.1 and FDM-1M), the SW-GS convergence rate was degraded due to using low values of W and high $\frac{W}{S}$ ratios. In the case of FDM-1M, this degradation of performance was balanced by a decrease of cache misses leading to similar SW-GS runtimes regardless of the W and S values used. However, in the case of BDC-1-0.1, the degradation of the SW-GS convergence rate was severe. This has led to more discrepancy in SW-GS runtimes with varying W and S sizes as shown in Figure 4.8). The degradation of convergence has been quantified by evaluating the iteration efficiency of each experiment on BDC-1-0.1. For instance, the iteration efficiency decreased to 20% when running SW-GS using 4 threads and setting $W = 50,000$ and $S = 10,000$ while the worst iteration efficiency in the case of FDM-1M was 88%.



(a) Runtimes of SW-GS on an Intel i7 processor

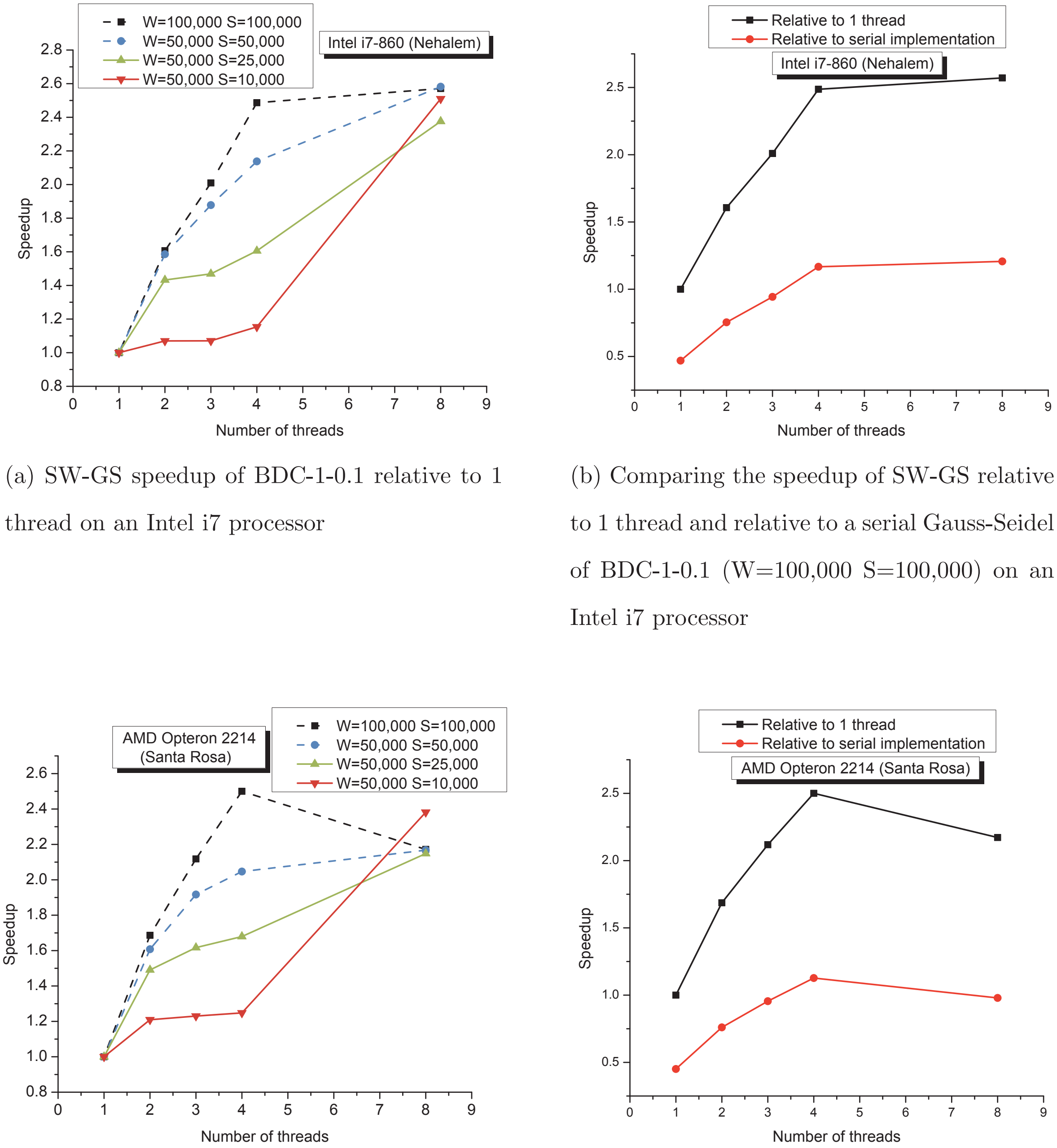
(b) Runtimes of SW-GS on an AMD Opteron processor

(c) Number of iterations

(d) Iterations efficiency

Figure 4.8: **BDC-1-0.1 Sliding-Window Gauss-Seidel execution times.** Sliding-Window Gauss-Seidel of BDC-1-0.1 (DOF=632,883 NNZ=4,409,973 CSR=51 MB). Relative residual error = 10^{-3}.

The speedup and scalability of SW-GS was also investigated in the case of BDC-1-0.1. Overall, the runtime and speedup results (Figure 4.9) show the same trend of SW-GS scalability as when applied to FDM-1M as the number of threads increased, but the maximum achievable speedup was less than that of the FDM-1M. The main reason is that not only did BDC-1-0.1 have low convergence rate, but also thread utilization was reduced due to the un-even workload distribution among threads. BDC-1-0.1 is an unstructured matrix, hence the number of non-zeros in a given thread's working set (i.e. W rows) differs from other threads. This fact is validated in Figure 4.10 which shows the time spent by threads waiting at a barrier (shown in blue).



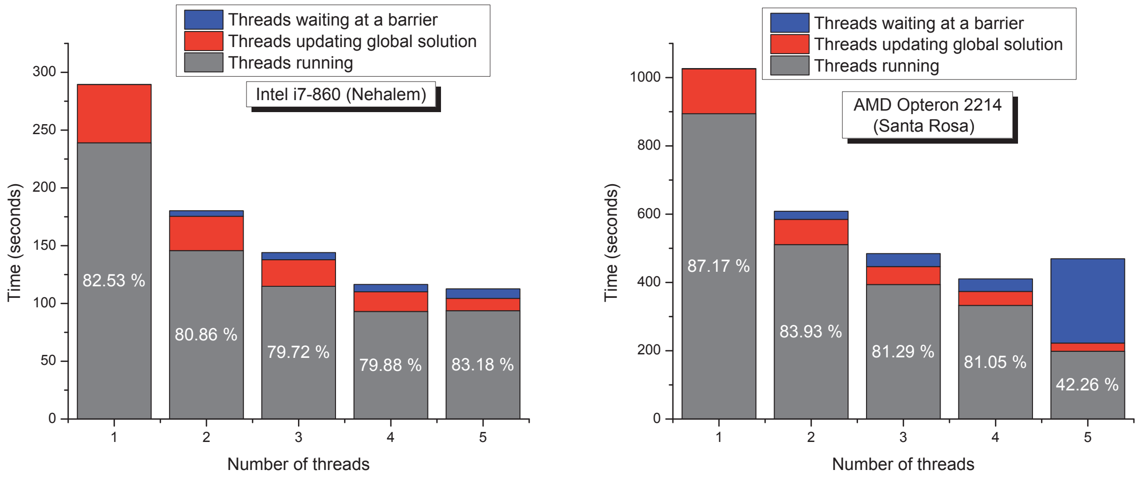
(a) SW-GS speedup of BDC-1-0.1 relative to 1 thread on an Intel i7 processor

(b) Comparing the speedup of SW-GS relative to 1 thread and relative to a serial Gauss-Seidel of BDC-1-0.1 (W=100,000 S=100,000) on an Intel i7 processor

(c) SW-GS speedup of BDC-1-0.1 relative to 1 thread on an AMD Opteron processor

(d) Comparing the speedup of SW-GS relative to 1 thread and relative to a sequential Gauss-Seidel of BDC-1-0.1 (W=100,000 S=100,000) on an AMD processor

Figure 4.9: **BDC-1-0.1 speedup of Sliding-Window Gauss-Seidel.** Speedup of BDC-1-0.1 (DOF=632,883 NNZ=4,409,973 CSR=51 MB) on an Intel i7 and an AMD Opteron processors. The high cost of synchronization is illustrated in (b) and (d).



(a) Thread execution efficiency of BDC-1-0.1 (W=100,000 S=100,000) on an Intel i7 processor

(b) Thread execution efficiency of BDC-1-0.1 (W=100,000 S=100,000) on an AMD Opteron processor

Figure 4.10: **BDC-1-0.1: SW-GS threads utilization.** The percentage of runtime a SW-GS is spending on thread execution and synchronization (BDC-1-0.1 DOF=632,883 NNZ=4,409,973 CSR=51 MB).

4.4 Concluding Remarks

Iterative solvers incur large amounts of cache misses. A large percentage of those misses are due to the limited capacity of the size of the cache memory relative to the size of the coefficient matrix A. They are known as "capacity misses". However, in this chapter, it has been shown that it is possible to decrease the capacity cache misses caused by the size of matrix A. This has only been made possible by the fact that iterative solvers based on Jacobi and Gauss-Seidel (i.e. often referred to as *stationary* based iterative solver) do not impose stringent requirements on the order of execution of rows. Hence, a variation of these two methods has led to the SW-GS presented in this thesis, which allowed multiple threads to re-iterate over parts of the coefficient matrix A, hence maximize its usage while present in the cache.

The success of SW-GS in reducing capacity misses did not translate into a gain in reducing the overall iterative solver runtime, since it decreased the convergence effect of Gauss-Seidel and also it increased the number of synchronization points. This raises the question as to whether current multi-core processors are suitable to being used for fine-grained parallelism.

The granularity of a multi-threaded algorithm is the level at which a problem is divided into subcomponents p. The number of components is positively correlated with the number of synchronizations required. If we consider the cost of executing an algorithm on a component p to be $E(p)$ and the cost of synchronization to be $S(p)$, then the ratio of $\xi_g = \frac{E_p}{E_p+S_p}$ would be a better way to quantify efficiency of a multi-thread algorithm for a specific granularity level.

A suitable granularity level would be the one that maximizes ξ_g, which can clearly be achieved by reducing the synchronization time S_p and increasing the execution time E_p through efficient multi-threaded implementation, which is hindered by the fact that the synchronization and execution times are positively correlated. Further discussion on fine-grained parallelism is presented in § 7.1.3

Chapter 5

Performance Evaluation of Sparse Matrix-Vector Multiplication on Multi-Core Processors

Contents

5.1 Introduction

Sparse matrix-vector multiplication (SMVM) is an essential and widely used kernel arising in many scientific computational problems. It is of particular importance for the computational electromagnetic community, since it serves as a building block for many iterative solvers of sparse systems of equations such as those that are based on the Conjugate Gradient (CG) Method [95, 96]. Algorithm 5.1 shows a basic implementation of a CG algorithm, where during each iteration, one SMVM (line 7, Ad), two inner-products (line 8 and 12) and three vector updates (line 9, 10 and 14) are executed.

Algorithm 5.1 Conjugate Gradient Method

```
 1: $i \Leftarrow 0$
 2: $r \Leftarrow b - Ax$
 3: $d \Leftarrow r$
 4: $\delta_{new} \Leftarrow r^T r$
 5: $\delta_{old} \Leftarrow \delta_{new}$
 6: while $\|r\|/\|b\| < error\_tolerance$ do
 7:     $q \leftarrow Ad$
 8:     $\alpha \leftarrow \frac{\delta_{new}}{d_T q}$
 9:     $x \leftarrow x + \alpha d$
10:     $r \leftarrow r - \alpha q$
11:     $\delta_{old} \leftarrow \delta_{new}$
12:     $\delta_{new} \leftarrow r^T r$
13:     $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
14:     $d \leftarrow r + \beta d$
15: end while
```

For large realistic problems, especially those which produce systems of equations that are badly conditioned, the number of iterations required by an iterative solver to converge to the solution is high. Within each iteration, sparse matrix-vector multiplication is the most time consuming step relative to the other kernels (i.e. inner-products and vector updates), hence a large amount of research has been directed towards optimizing this kernel.

It is well established that matrix-vector multiplication ($Y = A \times X$) exhibits a low floating-point operations (FLOP) count to memory access ratio, regardless of whether A is dense or sparse [97, 98]. The number of floating-point operations for SMVM is 2 × number of non-zeros (NNZ) in A; each non-zero in A is multiplied by the corresponding entry in X and the result is added to the corresponding entry in Y. The number of memory operations for each non-zero is three. This corresponds to 12 bytes of memory transfer when single floating-point precision is used and 24 bytes when double-floating precision is used. This low ratio of FLOP/BYTE makes SMVM a memory bandwidth limited problem requiring the use of optimization techniques which efficiently use the memory hierarchy system (main memory, caches and registers).

The experiments conducted and presented in this chapter aim at analyzing both the effectiveness and the limitation of the commonly used SMVM optimization techniques when applied on the matrix set described in Chapter 3. These matrices were obtained by applying different finite element formulations used in FEM analysis for typical problems.

A brief overview on SMVM optimization techniques and the attained performance results will be presented in § 5.3. § 5.4 will present the sparse storage schemes used in this chapter. Next, the effect of cache misses on SMVM performance when accessing the vector X in $Y = A \times X$ will be analyzed in § 5.5. Furthermore, since the matrices resulting from using FEM have low number of non-zeros per row, the effect of such a property on the loop setup overhead in SMVM is reported in § 5.6. Finally, experiments on parallel SMVM and the impact of using multiple threads on cache performance is shown in § 5.8.

5.2 Complexity Analysis of Conjugate Gradient Kernels

The runtime complexities of kernels executed within a single CG iteration are shown in Table 5.1. Those complexities assume that p processors are available, where $p \gg N$ (in the case of SMVM) and $p \geq N$ (in the case of vector update and inner-product kernels). N being the number of the degrees of freedom of a system of equations $Ax = b$.

Table 5.1: Complexity and parallel complexity of CG kernels.

Kernel	Sequential	Parallel
Vector update	$O(N)$	$O(1)$
Inner-product	$O(N)$	$O(\log_2 N)$
SMVM	$O(N)$	$O(\log_2 N)$

The runtime complexities presented above are not applicable to multi-core based processors where the number of processor cores or threads is less than the number of degrees of freedom (i.e. $p \ll N$). In this case, parallel vector update has a complexity of $O(\frac{N}{p})$ and parallel inner-product and SMVM complexities will be derived as such:

Inner-Product The inner-product of two vectors each of size N can be completed in N sequential steps (i.e. $O(N)$). When $p \geq N$, the inner-product can be performed in $O(\log_2 N)$ steps since each of these "reduction" steps, as shown in Figure 5.1, can be executed in parallel. However, when $p \ll N$, each reduction step will require a number, SS, of sub-steps; thus the total number of runtime steps is calculated according to Equation 5.1

$$\sum_{k=0}^{\log_2(N)} SS \tag{5.1}$$

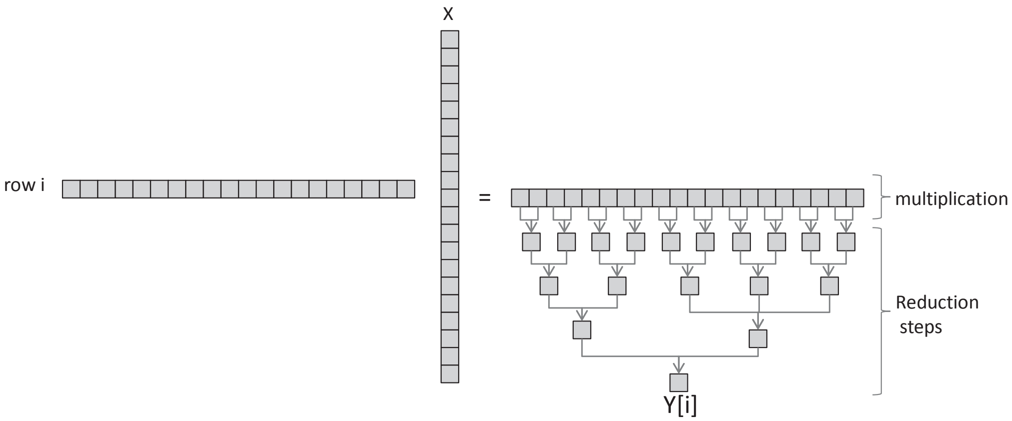


Figure 5.1: Parallel inner-product.

The number of operations in each sub-step, SS, is related to the number of threads, p, and the "reduction" step, k, according to Equation 5.2

$$SS = \tfrac{N/p}{2^k} \tag{5.2}$$

Replacing Equation 5.2 into Equation 5.1 gives rise to Equation 5.3.

$$\sum_{k=0}^{\log_2(N)} \tfrac{N/p}{2^k} = \tfrac{N}{p} \sum_{k=0}^{\log_2(N)} \tfrac{1}{2^k} \tag{5.3}$$

Given that a geometric series of the form $\sum\limits_{k=0}^{\infty} r^{-k}$ is equal to $\frac{1}{1-r}$, the worse case scenario of Equation 5.3 can be re-written as

$$\tfrac{N}{p} \sum_{k=0}^{\log_2(N)} \tfrac{1}{2^k} = \tfrac{N}{p} \sum_{k=0}^{\log_2(N)} 2^{-k} \simeq 2 \times \tfrac{N}{p} \tag{5.4}$$

Hence, the complexity of parallel inner-product on p processors, where $p \ll N$ is $O\left({}^{N}\!/_{p}\right)$.

SMVM SMVM requires $O(\eta(A))$ operations, where $\eta(A)$ is the number of non-zeros of A. In the case of the *Finite Element Method*, the number of non-zeros can be roughly related to the average number of non-zeros per row ϑ and the degrees of freedom N. In this case, SMVM complexity is $O(\vartheta N)$. Parallel SMVM can be easily parallelized into $O(\frac{N}{p} log_2\ \vartheta)$ where p is the number of processors. Note that it is assumed in here that the inner-product of ϑ can be completed in parallel.

5.3 SMVM Optimization Techniques on Multi-core Processors

SMVM optimization techniques work on reducing the number of main memory fetches by reordering the computations and using more condensed storage techniques to avoid cache misses. For example, register blocking and cache blocking [99, 98] are two important optimization techniques which increase the temporal and spatial locality of using processor registers and cache memory respectively. Computational re-ordering also increases contiguous and regular memory accesses which are useful for compiler and low level (processor) tuning techniques such as vectorization (using SIMD instructions), loop unrolling and memory prefetching [69, 71].

The extent of the success of these techniques varies depending upon the matrix structure and processor architectures [100, 71]. Matrix structure refers to the set of matrix properties, such as the non-zeros distribution, the average number of non-zeros per row and sparsity. These properties affect the choice of sparse storage and thus the SMVM algorithm to be used. On a multi-core based processor, the effectiveness of SMVM optimization varies greatly, mainly due to the architectural difference between all available multi-core processors. The difference stems from using different processing cores (Cell processor vs Intel/AMD processors) and different topologies in building a shared memory hierarchy (Intel vs AMD). Many studies have investigated the performance of SMVM on different multi-core architectures [69, 101, 100, 71]. They attempted to qualitatively understand the obtained performance on each of the processors and with the large set of test matrices used in the experiments. Their empirical results provided some optimization guidelines on multi-core and shared cache based processors. For instance, matrices with a small average number of non-zeros per row - a case that occurs in matrices obtained from FEM - showed little performance gain on both an AMD quad-core Opteron 2356 (Barcelona) and an Intel quad-core Xeon E5354 (Clovertown) architecture. This observation was consistent on a GPU architecture (GeForce GTX 280).

To further understand the effect of each of the known limitations in SMVM on multi-core based processors, many studies performed experiments by singling out each of the factors that are believed to be limiting the performance of SMVM (e.g. indirect memory access, short row length, matrix structure). For instance, reference [101], by removing the effect of irregular memory access on X, found that only 30% of the test matrices exhibited more than a 10% speedup; this means that the irregular access is not the prevailing bottleneck. However, the irregular access can have a greater impact when the matrix used has short row lengths; this is due to the backward jumps in memory fetches from the vector X (short row lengths also degrade performance due to loop overheads). Using a similar technique, the effect of indirect memory references has been found to pose the greatest performance degradation (i.e. a 50% performance degradation when compared a dense matrix-vector multiplication) [102].

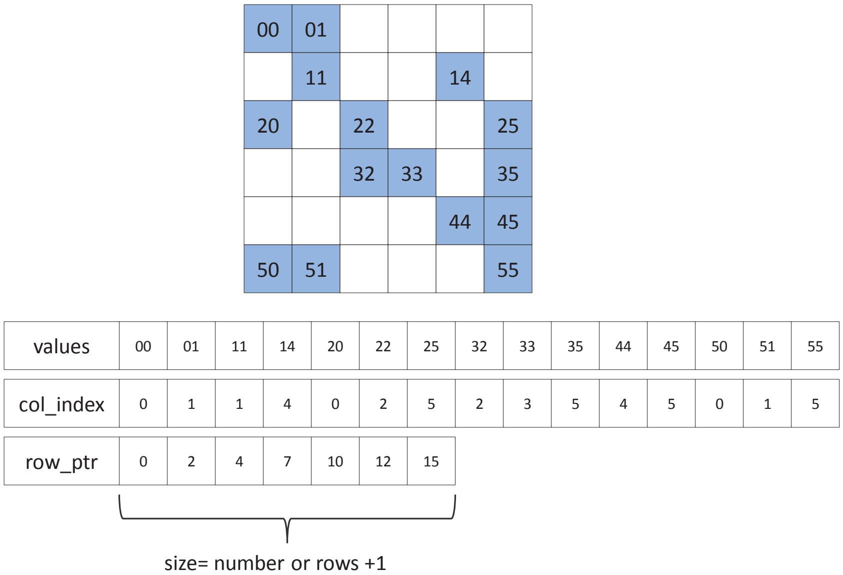
5.4 SMVM using CSR and NVIDIA's HYB Sparse Storage

Many storage schemes have been investigated with the goal to enhance the performance of SMVM, each proved to be efficient for certain types of problems (i.e. matrix structure) or processor architecture (for a review on sparse matrix storage schemes, refer to [103, 104]). Of particular interest when working with SMVM is the CSR and NVIDIA's hybrid (HYB) sparse storage schemes.

5.4.1 SMVM Using CSR Sparse Storage

The Compressed Sparse Row storage format (CSR), as explained in Chapter 3 § 3.3, is a simple storage format that can store any matrix regardless of its structure. Figure 5.2

and Algorithm 5.2 illustrate examples of the Compressed Sparse Row storage and a naive implementation of SMVM using CSR, respectively.



00	01				
	11			14	
20		22			25
		32	33		35
				44	45
50	51				55

values	00	01	11	14	20	22	25	32	33	35	44	45	50	51	55
col_index	0	1	1	4	0	2	5	2	3	5	4	5	0	1	5
row_ptr	0	2	4	7	10	12	15								

Figure 5.2: Compressed Sparse Row Storage (CSR)

The CSR storage format does not make any assumptions about the structure of the matrix and hence it is widely used when the matrix structure is not known a priori, as it balances efficiency and applicability.

Algorithm 5.2 Naive implementation of a SMVM using CSR sparse storage.

```
Input: Matrix $A$ stored in CSR format
Input: Vector $X$
Output: Vector $Y = A \times X$
N $\Leftarrow$ degrees of freedom of matrix $A$.
for i = 1 to N do
    $row\_start = row\_ptr[i]$
    $row\_end = row\_ptr[i+1]$
    for $j = row\_start \rightarrow row\_end$ do
        $column = col\_index[j]$
        $sum \leftarrow sum + X[column] \times values[j]$
    end for
end for
```

5.4.2 SMVM Using NVIDIA's Hybrid Sparse Storage

NVIDIA's hybrid sparse storage scheme (HYB) is a combination of the ELLPACK [60] sparse storage format and the Coordinate list format (COO). As explained in Chapter 3, the ELLPACK sparse storage scheme stores a $N \times N$ sparse matrix into two $N \times W$ dense data structures (*ELL_values* and *ELL_column_ind*) as shown in Figure 5.3. The size W corresponds to the maximum number of non-zeros per row. When the number of non-zeros per row is less than W, zeros are padded to fill the remaining locations. *ELL_values* stores the values of non-zeros in each row in a condensed form and pads the remaining spaces with zeros. *ELL_column_ind* stores the column index of each corresponding non-zero in the *ELL_values* and "-1" for the padded non-zeros. The ELLPACK format is well suited for vector processing (GPU) since the rows have a fixed length which permit warps (groups of 32 threads on a GPU that execute the same kernel simultaneously) to be aligned with the data structure. For matrices that have high discrepancy in their number of non-zeros per row, the number of padded zeros can be large; thus inducing additional useless storage space, floating-point and memory operations. To overcome this disadvantage of the ELLPACK for these type of matrices, NVIDIA's implementation of SMVM on a GPU (cusp-library) [105] uses a hybrid matrix storage format (HYB), which is a combination of the ELLPACK format and the Coordinates list format (COO), as illustrated in Figure 5.4. The advantage of the HYB format over the ELLPACK is that it reduces the amount of padded zeros; it does so by using the COO storage to handle the non-zeros that do not fit in the optimal ELLPACK storage.

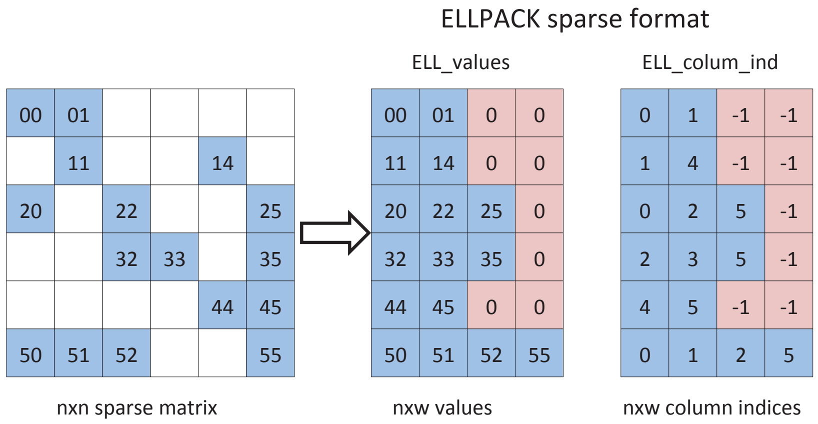


Figure 5.3: The ELLPACK sparse storage format.

A COO format stores a matrix into three separate arrays. One array is used to store the matrix's non-zeros and the other two arrays store the row index and the column index of each non-zero respectively. This format is rarely used since the operation to randomly access and retrieve a non-zero entry has a complexity of $O(n)$ steps. However, in the case of HYB storage, COO is used to store a very small number of non-zeros of the matrix. An example of this storage is shown in Figure 5.4 where only few non-zeros of the matrix where stored using COO.

Preliminary investigations have been conducted to compare the performance of SMVM when using the Hybrid and CSR storage format. In our implementation of HYB on a CPU, the “-1” value in *ELL_column_ind*, which is used to denote a padded zero in *ELL_values*, has been replaced by a valid column number (i.e. “0”) in order to avoid the use of a conditional *if* statement which checks for a ”-1” column in the inner loop of the SMVM implementation as shown in Algorithm 5.3 line 10. The results in Figure 5.5 shows that the performance of SMVM using HYB storage and SMVM using CSR storage are similar for FEM matrices. HYB was more advantageous than CSR for small matrices generated from using first-order nodal FEM (2D and 3D) (i.e. ET-1 and BDC-1 respectively). For the set of miscellaneous matrices, CSR's performance was superior to that of HYB, mainly due to the large difference in the number of non-zeros per row in these matrices.

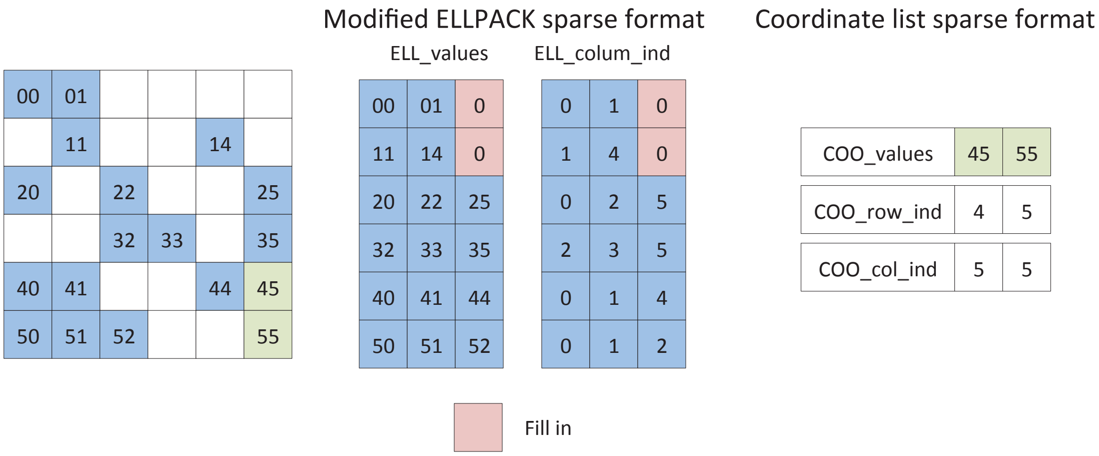


Figure 5.4: **Modified Nvidia's Hybrid sparse format.** Nvidia's Hybrid sparse format stores entries in an ELLPACK storage and COO. In this thesis, the original ELLPACK storage scheme has been modified by setting the column index of fill-ins as "0" instead of "-1".

Algorithm 5.3 SMVM using NVIDIA's Hybrid (HYB) sparse storage.

```
Input: Matrix $A$ stored in NVIDIA's hybrid storage
Input: Vector $X$
Output: Vector $Y = A \times X$
$N$: degrees of freedom of matrix $A$.
$ELLvals$: $N \times W$ matrix.
$ELLcols$: $N \times W$ matrix.
$nCOO$: the number of elements in the COO storage
$row\_width$: the maximum row width of
for $r = 1 \to N$ do
    $sum = 0$
    for $c = 1 \to W$ do
        $column = ELL\_column\_ind[r][c]$
        if $column$ not equal $-1$ then
            $sum \leftarrow sum + ELL\_values[r][c] \times X[column]$
        end if
    end for
    $Y[r] \leftarrow sum$
end for
for $i = 1 \to nCOO$ do
    $value \leftarrow COOvals[i]$
    $rowind \leftarrow COOrow[i]$
    $colind \leftarrow COOcol[i]$
    $Y[rowind] \leftarrow Y[rowind] + value \times X[colin]$
end for
```

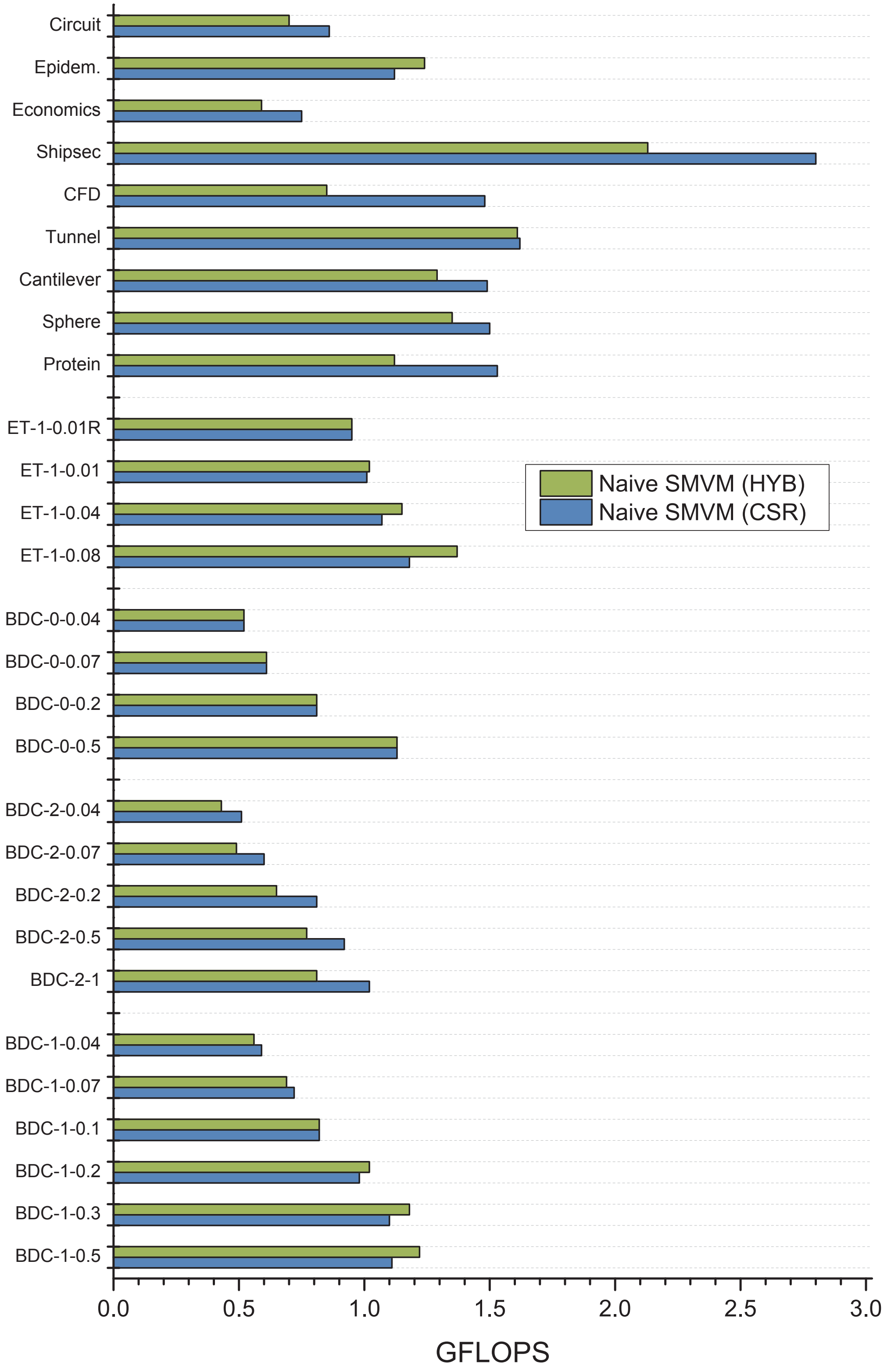


Figure 5.5: Performance comparison (GFLOPS) of SMVM using CSR and HYB sparse storage (single-precision floating-point).

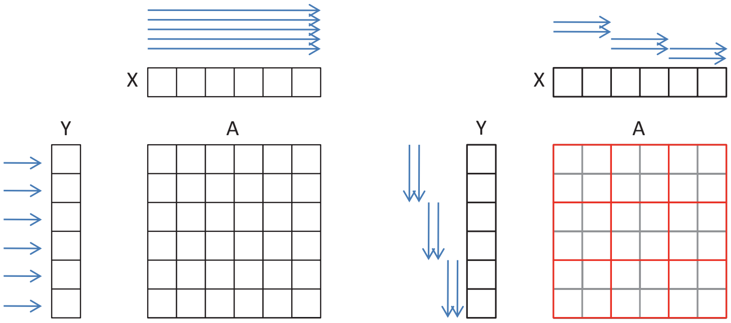
5.5 Cache Blocking Optimization

5.5.1 Cache Blocking Using Dense Storage

The goal of a cache blocking technique in matrix-vector multiplication is to reduce the cache misses incurred when reading from the X vector and writing to the Y vector. First, the source for cache misses will be illustrated for the case when A is a dense matrix. For a naive implementation of a matrix-vector multiplication, where A is a dense $N \times N$ matrix (Algorithm 5.4), the memory access pattern on X is shown in Figure 5.6a.

Algorithm 5.4 Naive implementation of dense matrix-vector multiplication.

```
$N$: degrees of freedom of matrix $A$.
$Y = A \times X$: system of equations of size $N$.
for $i = 1 \to N$ do
    $sum \Leftarrow 0$
    for $j = 1 \to N$ do
        $sum \leftarrow sum + A[i][j] \times X[j]$
    end for
    $Y[i] \leftarrow sum$
end for
```



(a) Without cache blocking (b) With cache blocking

Figure 5.6: **Memory access pattern on X and Y in SMVM (where A is a dense matrix).** a) With cache blocking and b) Without cache blocking.

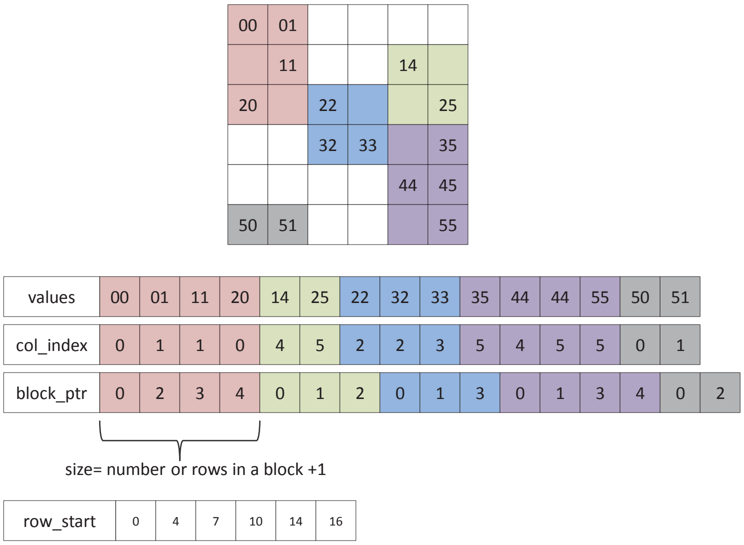
In this naive implementation, X is traversed for each outer loop iteration. This memory access pattern has low temporal locality of access on X. Each element of X

is accessed once by the end of each outer iteration. This non optimized data locality access will cause many cache misses to occur especially when the size of the system (A, Y and X) is larger than the size of the cache memory. As illustrated in Figure 5.6b, reordering the computations by dividing the matrix into blocks that fit the cache will induce an alteration in the access pattern on X so that elements are reused multiple times during each outer loop before they are evicted from the cache. The same logic applies to Y.

5.5.2 Cache Blocking Using Sparse Storage

When a memory location is fetched from the main memory into the cache, the memory system actually retrieves a cache line which contains the data in memory spaces contiguous to that of the requested datum. This fact is advantageous when the matrix A is dense due to the spatial locality of memory accesses on X. In the case of a sparse A, especially for unstructured matrices, the spatial locality of memory accesses to X is less than that when the matrix is dense. Furthermore the predictable access pattern of X, in the case of a dense A, allows for the prefetching feature of most current processors to retrieve data that are likely to be needed. In the case of a sparse A, pre-fetching will not be accurate due to indirect addressing.

One approach to using cache block optimization is reordering the non-zeros and storing the matrix as a logical block by using the Block Compressed Sparse Row storage (BCSR), which is an extension of CSR, as shown in Figure 5.7. Figure 5.8 demonstrates the changes occurring in the access pattern on X when using cache block optimization for this particular example.



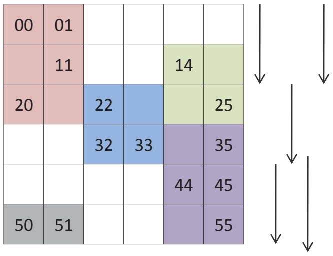
00	01				
	11			14	
20		22			25
		32	33		35
				44	45
50	51				55

values	00	01	11	20	14	25	22	32	33	35	44	44	55	50	51	
col_index	0	1	1	0	4	5	2	2	3	5	4	5	5	0	1	
block_ptr	0	2	3	4	0	1	2	0	1	3	0	1	3	4	0	2

size= number or rows in a block +1

row_start	0	4	7	10	14	16

Figure 5.7: **Block Compressed Sparse Row Storage (BCSR).**



00	01				
	11			14	
20		22			25
		32	33		35
				44	45
50	51				55

Figure 5.8: **SMVM memory access pattern when using block CSR (BCSR) sparse storage.**

5.5.3 Analyzing the Effect of Cache Blocking Techniques on SMVM Performance

In order to evaluate the magnitude of the impact exerted by accessing X on SMVM performance, the multiplication by $X[column]$ was replaced by $X[i]$ (Algorithm 5.5, line 7). Although this multiplication yielded an incorrect result, the aim was to show an upper bound on performance gain in cache blocking (i.e. no cache misses on X). Since the variable *column* (line 6) is not used elsewhere in the code, the compiler will not execute it. However, in order to force the compiler to execute this statement (line 6) a simple print statement of *column* at the end of SMVM (line 10) can force the compiler to evaluate line 6.

Algorithm 5.5 Modified SMVM to eliminate the effect of cache misses on X.

```
 1: $N$: degrees of freedom of matrix $A$.
 2: for $i = 1 \rightarrow N$ do
 3:     $row\_start \leftarrow row\_ptr[i]$
 4:     $row\_end \leftarrow row\_ptr[i+1]$
 5:     for $j = row\_start \rightarrow row\_end$ do
 6:         $column \leftarrow col\_index[j]$
 7:         $sum \leftarrow sum + X[i] \times values[j]$
 8:     end for
 9: end for
10: print $column$
```

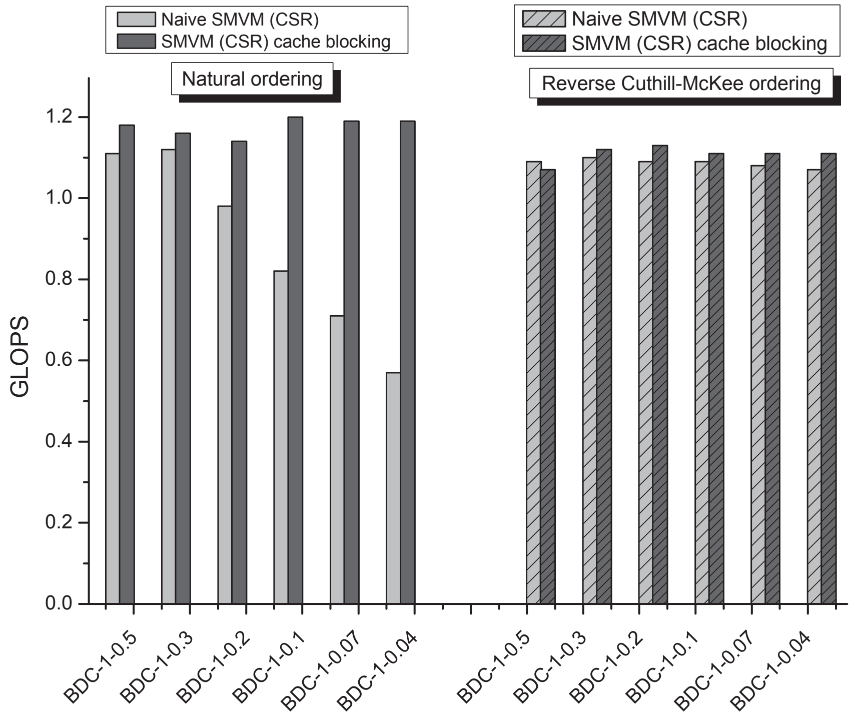


Figure 5.9: BDC-1: SMVM performance when using cache blocking the access on X (single-precision floating-point).

Eliminating the cache misses of X has increased the performance of SMVM significantly (as anticipated) when the matrix was unstructured (i.e. BDC-1) as shown in Figure 5.9 (Natural ordering). To further validate the results, the set of matrices in Table 3.3 (BDC-1) were reordered to reduced their bandwidth using the Reverse Cuthill-McKee (RCM) technique [106] (e.g. Figure 5.10 shows BDC-1-1 reordered using RCM). When the matrices were reordered using RCM, the performance of SMVM using cache blocking was close to the performance of SMVM without cache blocking (Figure 5.9), since cache misses were less due to the ordered access pattern on X.

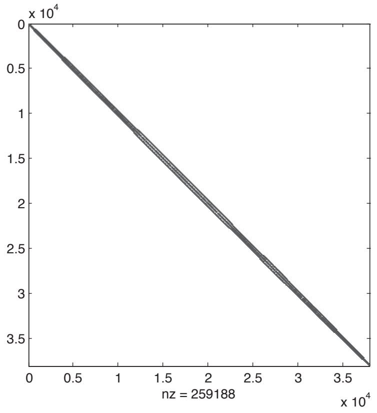


Figure 5.10: Reordered BDC-1-1 using Reverse Cuthill-McKee.

5.6 Loop Setup Overhead and Loop Unrolling

As discussed in the introduction, the matrix structure affects the performance of an SMVM kernel and the attainable GFLOPS. One of the factors that has been argued to be contributing to reducing the performance of SMVM is the low number of non-zeros per row [71]. For each row of the matrix A, the inner loop of the SMVM code, whether using the CSR storage or using the HYB storage as shown in Algorithm 5.2 and Algorithm 5.3 respectively, iterates over the row's non-zeros and multiplies them by the correspond entries in X. When only a few non-zeros are present, the inner loop setup overhead time would dominate the calculation time and would not be able to be amortized over the short calculation time of a few non-zeros. Since the set of FEM matrices used in this work falls within this category (i.e. low NNZ per row) a test examining the degradation of the SMVM performance due to the inner loop setup overhead has been carried out by replacing the inner loop of SMVM with a set of instructions which explicitly multiply each element of A by its corresponding element in X; this technique is often referred to as "loop unrolling". "Loop unrolling" has been made possible by the use of the ELLPACK (or Hybrid) sparse format since the number of non-zeros per row is fixed, hence the number of times an inner loop executes its inner instruction is fixed. In such a case, the inner loop can be eliminated and the instruction within the inner loop can be replaced by explicitly writing the set of instructions that would have been executed by the inner loop. Algorithm 5.6 illustrates a sparse matrix-vector multiplication using the ELLPACK storage. Assuming that the width W of the ELLPACK storage is 7, the inner loop which multiplies the non-zeros of a row by the corresponding locations in X is replaced by seven instructions. The effect of this technique on the performance of SMVM when applied on BDC-1 matrix test set is shown in Figure 5.11. It can be seen that while loop unrolling did increase SMVM performance, it was not as significant as the performance gain obtained from eliminating cache misses on X.

Algorithm 5.6 SMVM loop unrolling using NVIDIA's Hybrid (HYB) sparse storage.

```
Input: Matrix $A$ stored in NVIDIA's hybrid storage
Input: Vector $X$
Output: Vector $Y = A \times X$
$N$: degrees of freedom of matrix $A$.
$ELLvals$: $N \times 7$ matrix.
$ELLcols$: $N \times 7$ matrix.
for $r = 1 \to N$ do
    $sum = 0$
    $sum \leftarrow sum + ELL\_values[r][1] \times X[ELL\_column\_ind[r][1]]$
    $sum \leftarrow sum + ELL\_values[r][2] \times X[ELL\_column\_ind[r][2]]$
    $sum \leftarrow sum + ELL\_values[r][3] \times X[ELL\_column\_ind[r][3]]$
    $sum \leftarrow sum + ELL\_values[r][4] \times X[ELL\_column\_ind[r][4]]$
    $sum \leftarrow sum + ELL\_values[r][5] \times X[ELL\_column\_ind[r][5]]$
    $sum \leftarrow sum + ELL\_values[r][6] \times X[ELL\_column\_ind[r][6]]$
    $sum \leftarrow sum + ELL\_values[r][7] \times X[ELL\_column\_ind[r][7]]$
    $Y[r] \leftarrow sum$
end for
```

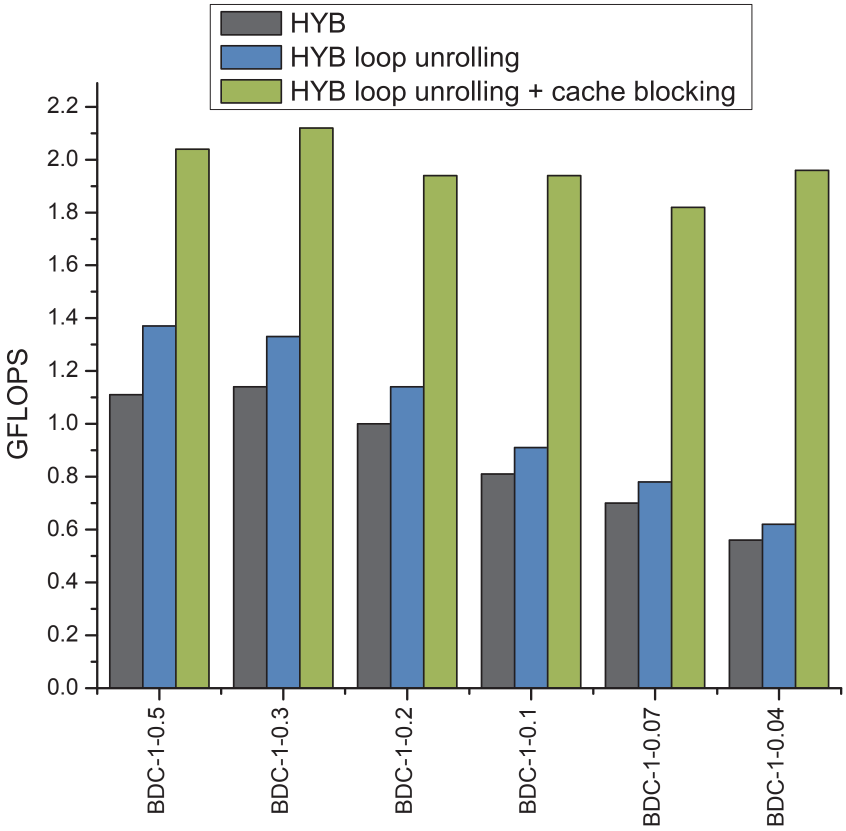


Figure 5.11: BDC-1 Loop unrolling and cache blocking (single-precision floating-point).

5.7 Memory Bandwidth

In order to test the performance of SMVM in relation to the processor's capabilities, the results obtained from the STREAM benchmark [107] were compared to the attainable bandwidth (MB/s) of SMVM when applied on the BDC-1 matrix set.

5.7.1 The STREAM Benchmark

The theoretical peak performance of a computer's memory subsystem mainly depends on two factors. The first is the transfer rate between the memory and the processor, the second is the number of bits that can be transmitted during each transfer cycle (i.e. bus width). Roughly, this is given by Equation 5.5. Additional technological enhancements can boost such performance, by increasing the number of interfaces between the memory and the processor and increasing the number of memory transfers per clock cycle. In this case, the theoretical memory bandwidth is given by Equation 5.6. This peak performance only measures the transfer rate of the physical layer of the memory's subsystem and assumes that data are exchanged between the processor and the memory in burst mode, that is, data is always available to be transferred and no other factors that are interfering, such as waiting for other operations.

$$BW(Mbit/s) = Memory\,Clock\,Frequency\,(Mhz) \;\times\; Bus\,Width\,(bits) \tag{5.5}$$

$$\begin{aligned} BW(Mbit/s) = Memory\,Clock\,Frequency \;\times\; (transfers\,per\,cycle) \\ \times\; (Bus\,Width\,) \;\times\; (number\,of\,channels) \end{aligned} \tag{5.6}$$

Alternatively, the STREAM benchmark [107] has been widely used as a realistic measure of the sustainable memory bandwidth. It works by applying four kernels on an array of data that do not fit in the cache. Table 5.2 shows the sustainable memory bandwidth of the processors used in this thesis's experiments.

Table 5.2: STREAM benchmark kernel operations.

Name	Kernel	Bytes/ iteration	FLOPS/ iteration	Intel i7-860 (MB/s)	AMD Opteron 2241 (MB/s)
COPY	a(i) = b(i)	16	0	10165.69	2963.51
SCALE	a(i) = q * b(i)	16	1	8878.59	2925.34
SUM	a(i) = b(i) + c(i)	24	1	8968.57	2400.48
TRIAD	a(i) = b(i) +q * c(i)	24	2	9675.90	3140.33

5.7.2 SMVM Memory Bandwidth

Memory bandwidth of four SMVM optimization techniques (i.e. loop unrolling - natural ordering, loop unrolling and cache blocking - natural ordering, loop unrolling - Reverse Cuthill-Mckee ordering, loop unrolling and cache blocking - Reverse Cuthill-Mckee ordering) are shown in Figure 5.12 when executed on an Intel i7-860 processor and in Figure 5.13 when executed on an AMD Opteron processor. In general, a version of SMVM that does utilize cache blocking optimization would work well below 50% of the STREAM benchmark sustained memory bandwidth, while an optimized SMVM (with cache blocking) only attained 70% of the STREAM benchmarks.

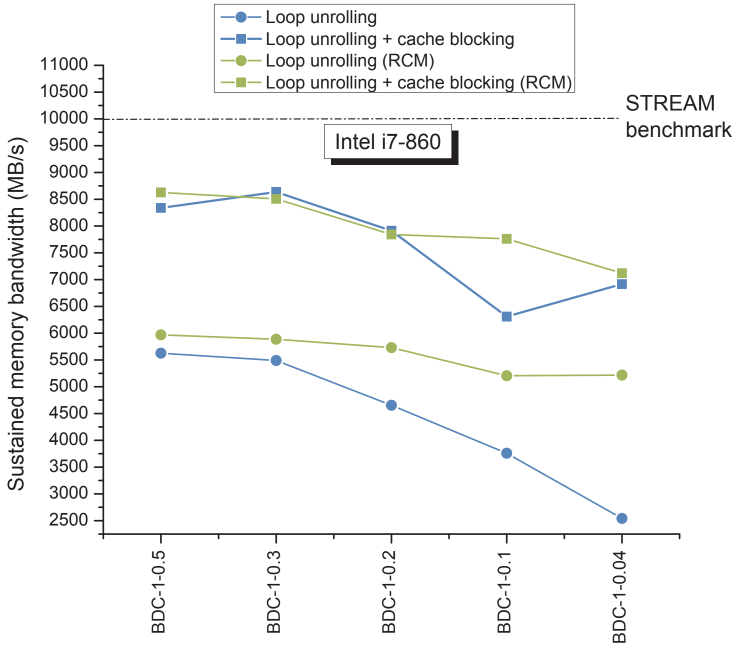


Figure 5.12: **BDC-1 sustainable memory bandwidth (MB/s) on Intel i7-860 processor**.

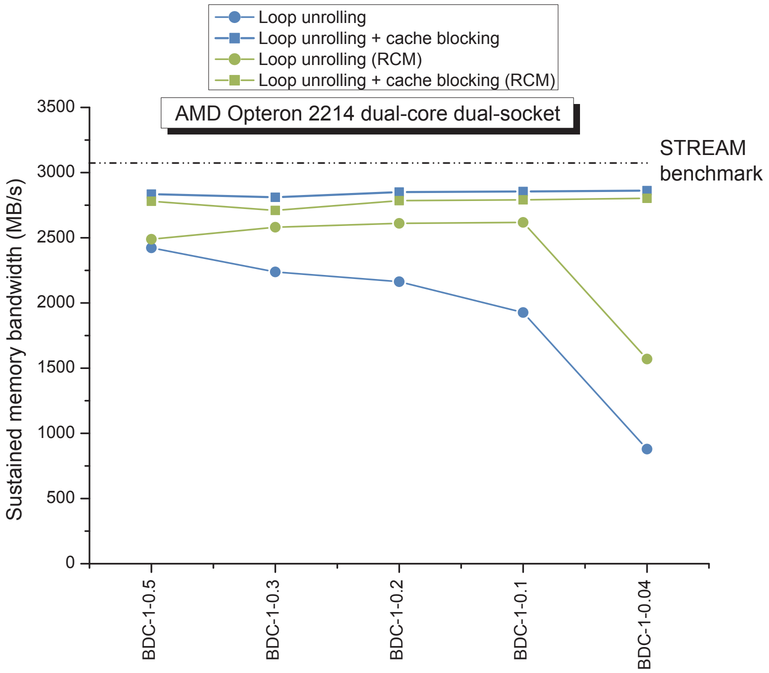
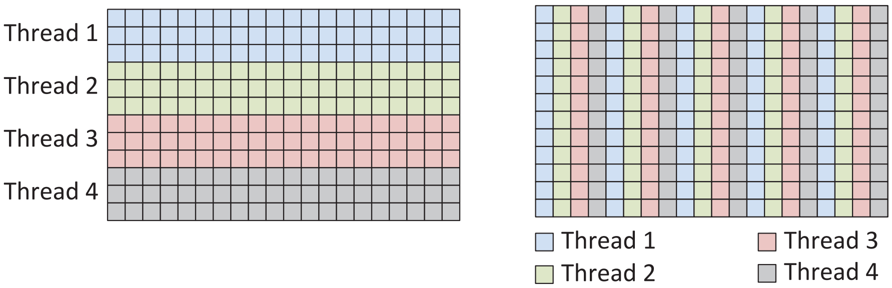


Figure 5.13: **BDC-1 sustainable memory bandwidth (MB/s) on an AMD Opteron 2241 dual-socket dual-core processor.**

5.8 Parallel SMVM

Sparse matrix-vector multiplication operations can be easily parallelized since each row of the matrix can be multiplied by X independently of the remaining rows. Let T be the number of threads where $T \ll n$ (n is the number of rows in the matrix A), each thread can be assigned a block of rows of the matrix to be multiplied by X as shown in Figure 5.14a. This thread distribution technique is referred to as "scalar" SMVM. One drawback of "scalar" SMVM on a multi-core processor is that threads can create cache contention and cache conflicts especially if all threads are executing simultaneously on a large data set. On a GPU, the limitation of this thread distribution is that threads (warps) are not aligned to access contiguous memory locations. A different distribution of threads, known as "vector" SMVM, assigns a group of threads to perform computations within the same row as shown in Figure 5.14b. This method however also has many drawbacks, one of which is that the inner-product between a row and X requires parallel reduction to add all multiplied entries. The reduction will render some threads idle and synchronization is required at each reduction step. The "vector" SMVM is suited for execution on a GPU due to the coalescent access on a GPU. In general, "vector" SMVM has been the method of choice on GPU, while "scalar" SMVM is used on a CPU.

On a multi-core processor, assigning one thread per matrix row (i.e. "scalar" SMVM) is a better choice than assigning multiple threads per row for matrices arising from FEM due to the low number of non-zeros per row. The small number of non-zeros are usually fetched into 1 cache line. Having multiple threads working on the same cache line (i.e. "vector" SMVM) leads to false cache sharing among threads which causes cache line invalidation; thus causing cache misses.



(a) Scalar parallel threads

(b) Vector parallel threads

Figure 5.14: **Parallel SMVM: Scalar and Vector thread distribution techniques.** Showing different techniques to map threads onto a matrix data set.

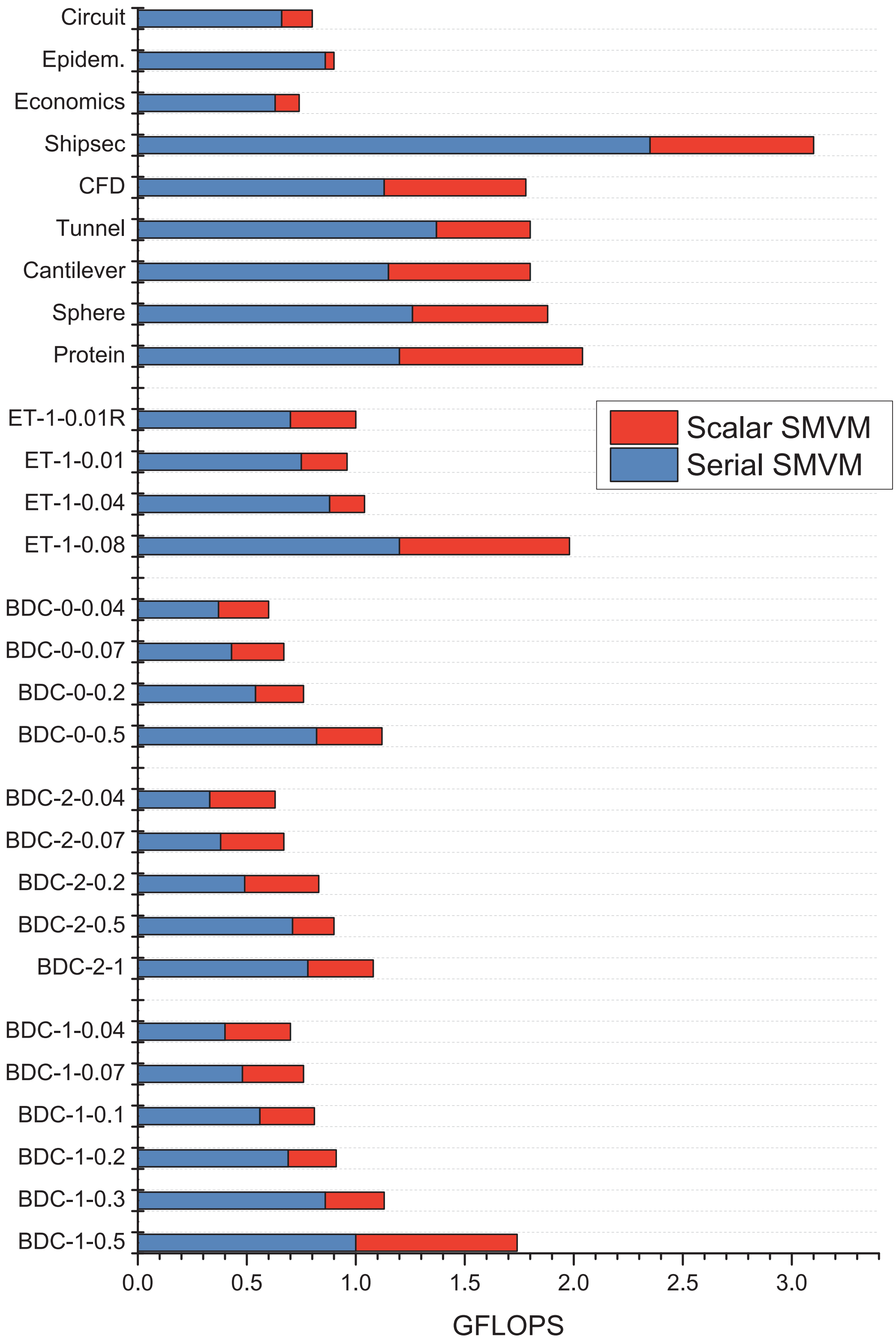


Figure 5.15: Parallel SMVM using HYB storage (double-precision floating-point). Performance increase due to using multi-threads is stacked on top of the performance using 1 thread.

The runtime results, expressed in GFLOPS, of sequential SMVM and parallel "scalar" SMVM for a large set of matrices stored using the HYB sparse storage scheme are shown in Figure 5.15. In general, matrices that are structured (i.e. banded), such as most of those obtained from *matrix market*, had higher GFLOPs than the unstructured matrices devised in Chapter 3 by applying FEM on EM problems (BDC-0, BDC-1, BDC-2 and ET matrices). An exception to this is matrices *Circuit* and *Economics* which had low GFLOPs mainly due to the large amount of fill-ins caused by using HYB storage. On the other hand, the *Epidem* matrix, like the matrices obtained from applying edge finite element formulation (BDC-0), attained low GFLOPs due to the low number of non-zeros per row. Those results re-confirm that optimizing SMVM to reduce the cache misses that occur when accessing the X vector in $Y = A \times X$ has the greatest effect on increasing SMVM performance.

More analysis has been carried on BDC-1-0.04 (DOF=3,125,216 NNZ=22,000,128 CSR=263.7962 MB) and ET-0.01R1 (DOF=2,666,039 NNZ=39,535,927 CSR=462.62 MB) to understand the attained GFLOPs when running sequential and parallel SMVM on each. Despite the fact that BDC-1-0.04 matrix has a smaller storage size and more efficient HYB storage (i.e. less fill-ins) than ET-0.01R, the latter exhibited higher GFLOPs. The percentage of execution cycles that were halted due to misses on the last level cache (L3) was larger than 60% in the case of BDC-1-0.04 and close to 35% in the case of ET-0.01R1. The access pattern to the vector x in ET-0.01R1 was more coalesced and exhibited more locality than the access pattern to the vector X in BDC-1-0.04.

In both cases, the number of execution cycles that were halted due to cache misses was reduced to 10% when using multiple threads (i.e. 4 threads). When a thread fetches data from the main memory (causing L3 cache misses), subsequent threads can take advantage of the data and reuse it if their temporal access to the datum is high; such is the case of the access pattern on X. This "true cache sharing" among threads did not cause invalidation on the low level private caches (L1 and L2) mainly because X is a read-only vector. This observation implies that read-only data that are frequently accessed in multi-threaded applications enhance cache performance when compared to a single threaded application.

5.9 Concluding Remarks

SMVM is an important kernel in Krylov based iterative solvers. Its main limitation is that it is a memory bandwidth limited problem since it has a low floating-point operations (FLOP) count to memory access ratio. Hence, in order to increase SMVM performance, it is important to resort to optimization techniques that enhance cache performance. Particularly, it has been re-confirmed in this chapter that optimization techniques which aim at reducing the number of cache misses, especially when accessing X in $Y = A \times X$, have been found to have the greatest effect on increasing the performance of SMVM. This can be accomplished by either re-ordering SMVM computations by using sparse storage techniques such as blocked CSR (BCSR) or re-ordering the matrix A using RCM to reduce its bandwidth. However, as it will be discussed in the next chapter, RCM decreases the convergence rate of preconditioned conjugate gradient (PCG) and reduces the degree of parallelism of the backward/forward solve within each PCG iteration.

SMVM optimization techniques on multi-core and GPU processors mainly focus on devising storage techniques which fit the processor's architecture. Basically, most research devises sparse storage where rows are combined and padded with zeros to have a block size that fits a processor's hardware specifications such as cache line size and warp size (in a GPU). Ideally, a matrix where the number of non-zeros in each of its rows are closely clustered is more amenable to be transformed into a suitable sparse storage; such is the case of matrices obtained from FEM problems. Although, the work in this chapter did not demonstrate the power of ELLPACK or similar storage schemes (except by using loop unrolling), its purpose was to show that it was possible to use such storage without incurring a great amount of overhead.

The SMVM kernel generally took advantage of using multi-threads which reduced the amount of cache misses due to the frequent access to a read-only shared data structure (i.e. the X vector). All threads (processor cores) were efficiently utilized leading to a reduction in SMVM runtime although the attainable GFLOPs was small relative to a processor's theoretical peak performance. However, many arising challenges are still present mainly due to the fact that SMVM is executed within each CG iteration along other kernels such as the inner-product and backward/forward solve (in the case of the Preconditioned Conjugate Gradient method). Those kernels have less degree of parallelism than SMVM and do not take advantage from using all processor cores. Hence reserving all resources for SMVM might not benefit the whole PCG process, even in the presence of an efficient SMVM implementation.

Chapter 6

Parallel Preconditioning Techniques on Multi-Core Processors

Contents

6.1 Introduction

Preconditioning of a system of linear equations $Ax = b$ roughly speaking is the introduction of a new matrix M such that the resulting system $M^{-1}Ax = M^{-1}b$ has a better "condition number[1]" than the original system. In a different way, the quality of a preconditioner is how well M^{-1} approximates A^{-1}. In the cases we deal with, the preconditioner is usually implicitly applied as part of the iteration of the Conjugate Gradient Method by backward/forward solving a system of the form $Ms = r$ (i.e. $s = M^{-1}r$) within each iteration of a Preconditioned Conjugate Gradient (PCG) algorithm (line 13 of Algorithm 6.1). PCG aims at reducing the number of iterations.

[1] The condition number of a matrix is the ratio of its largest eigenvalue to its smallest eigenvalue.

However, this is not enough to reduce the solver time if the cost to set up the preconditioner (line 1 of Algorithm 6.1) and to apply it (line 13 of Algorithm 6.1) within each iteration is high. Hence, a preconditioner must be fast to set up, fast to apply and must reduce the number of iterations.

Algorithm 6.1 Preconditioned Conjugate Gradient Method.

```
1: $M = \hat{L}\hat{U}$: Preconditioner setup
2: $\varepsilon$: Error tolerance
3: $i \Leftarrow 0$
4: $r \Leftarrow b - Ax$
5: $d \Leftarrow M^{-1}r$
6: $\delta_{new} \Leftarrow r^T r$
7: $\delta_{old} \Leftarrow \delta_{new}$
8: while $\|r\|/\|b\| < \varepsilon$ do
9:     $q := Ad$
10:    $\alpha := \frac{\delta_{new}}{d_T q}$
11:    $x := x + \alpha d$
12:    $r := r - \alpha q$
13:    $\boxed{s := M^{-1}r}$
14:    $\delta_{old} := \delta_{new}$
15:    $\delta_{new} := r^T s$
16:    $\beta := \frac{\delta_{new}}{\delta_{old}}$
17:    $d := s + \beta d$
18: end while
```

There are many classes of preconditioners each of which can be subdivided into many sub variants. Polynomial preconditioners for Krylov subspace methods were popular for some applications in the early stages of the preconditioning studies [108]. Preconditioners based on Jacobi, SOR (or SSOR for symmetric matrices) and block Jacobi are old techniques that are still in use [109, 103]. Currently, preconditioners that are based on approximate inverse and incomplete LU decomposition (ILU) are a popular choice. This chapter focuses on ILU based preconditioners, mainly due to the fact that the incomplete Cholesky preconditioner IC(0), which is a special case of ILU, has been the state-of-the-art preconditioner for iterative solvers running on desktop computers.

First, in § 6.2, incomplete Cholesky IC(0), will be applied on a set of matrices selected from Chapter 3. The aim of this step is to show the efficiency of setting up IC(0) preconditioner using an "in-place" factorization algorithm. It has been possible to use this algorithm in the case of IC(0) since the sparsity of the resulting preconditioner is priori-known. When devising an ILU preconditioner, the sparsity of the final preconditioner cannot be a priori-known unless "symbolic factorization" is applied first which is only applicable for ILU preconditioner technique that drops fill-ins based on their location. In this case, the bottleneck to rapidly generating a preconditioner lies in the "symbolic factorization" step. This step will be investigated in section § 6.4.

Another essential and time consuming step when using preconditioning techniques is the cost of applying a preconditioner M within each PCG iteration by backward/forward solving a system of the form $s = M \times r$. The degree of parallelism that can be obtained in this step will be examined in § 6.5.

The remainder of this section § 6.1.1.1 will overview LU factorization which is used in direct solvers and from which the idea of incomplete LU stems. Furthermore, the impact of matrix re-ordering before applying LU or ILU factorization will also be discussed in § 6.1.1.2.

6.1.1 Overview

In theory, a Conjugate Gradient Method converges in at most N iterations in the absence of round-off errors (i.e. on an infinite precision machine), where N is the number of degrees of freedom of the system of linear equations $Ax = b$. However in, practice, it is desired to perform a number of iterations i so that the ratio of the norm of the solution error at the i^{th} iteration $\|e_i\|$ and the norm of the initial solution error $\|e_0\|$ is small, i.e. $\|e_i\| \leq \varepsilon \|e_0\|$. The maximum number of iterations i required to reduce the norm of the error by ε is given by Equation 6.1 [110]. This number of iterations is related to the condition number κ of the matrix A which is defined as the ratio of the largest eigenvalue λ_{max} to the smallest eigenvalue λ_{min}, i.e. $\kappa = \lambda_{max}/\lambda_{min}$.

$$i \leq \left\lceil \frac{1}{2}\sqrt{\kappa}\left(\frac{2}{\varepsilon}\right)\right\rceil \tag{6.1}$$

The error of a solution at iteration i, $e_i = x_i - x_{solution}$ cannot be calculated since it requires the solution of the system of equations, $x_{solution}$. Instead, one way to measure the progress of the convergence of a CG iteration is to use the norm of the residual r instead of the norm of the error e since the residual of a solution $r_i = b - Ax_i$ is related to the error $r_i = -Ae_i$. In this case, $\|r_i\| \leq \varepsilon \|r_0\| \leftrightarrow \|r_i\| \leq \varepsilon \|Ax_0 - b\|$. When the initial solution vector x_0 of a CG is set to zeros, the stopping criterion of a CG becomes $\|r_i\| / \|b\| \leq \varepsilon$ (CG Algorithm 6.1 - line 6) .

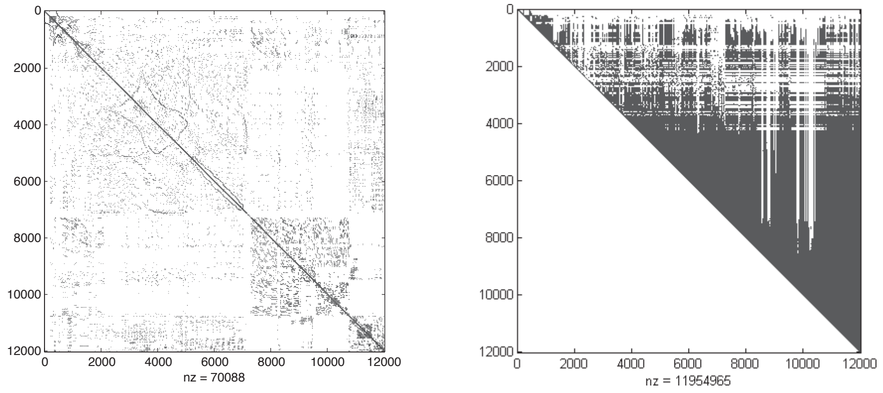
A condition number of 1 minimizes the upper limit of the number of iterations in Equation 6.1, hence the closer a matrix's condition number to 1, the fewer iterations it requires to converge. Furthermore, CG converges faster when the eigenvalues are clustered within the limits $[\lambda_{\min}, \lambda_{\max}]$ than when they are irregularly distributed.

Matrices obtained from applying FEM on low frequency EM problems are usually large and ill-conditioned ($\kappa \gg 1$) mainly due to the disparity in the magnitude of the matrix entries which are related to the physical material properties. An ill-conditioned system of linear equations has slow convergence, hence preconditioning the system of equations becomes essential to reduce the number of iterations.

6.1.1.1 Incomplete LU Factorization Overview

The incomplete factorization methods were first introduced by Varga [111], then later used as a preconditioner for the Conjugate Gradient Method [112]. They are a variation of the factorization technique that is commonly used in direct solvers during which a matrix A is decomposed into a lower L and an upper triangular matrix U such that $A = LU$ or $A = LL^T$ for symmetric matrices. In this case (direct solvers), the exact decomposition of A into LU results in a system of the form $LUx = b$ which is then solved in two steps: forward substitution $Ly = b$ and backward substitution $Ux = y$. Despite the fact that direct solvers that are based on this method are robust, they are not useful for large systems since the triangular matrices L and U lose their sparsity, as zero entries in the coefficient matrix A turn into non-zero entries (referred to as fill-ins) in L and U which require a large amount of operations and storage. Figure 6.1 shows a matrix A and the decomposed factor U that has a large number of non-zeros. Algorithm 6.2 shows a sample code of LU factorization assuming that matrix A is stored in a dense data structure.

Symbolic factorization is usually performed prior to LU decomposition (or ILU decomposition) so that the locations of fill-ins are determined to pre-allocate memory. This significantly reduces the factorization time. Symbolic factorization is usually carried by parsing an elimination tree, which is a data structure that depicts the order of factorization and dependency between the matrix's rows. Hence its usage is also important to explore parallelism in the factorization stage [113].



(a) Matrix A

(b) Upper triangle factor U

Figure 6.1: **LU factorization.** (a) Symmetric matrix NNZ=70,088 and (b) Upper triangular factor U obtained from decomposing A into an LU factors (NNZ=11,954,965).

Algorithm 6.2 Fan-out down-looking LU factorization using dense storage.

Input: U: the upper triangle to be factored
Output: U: factored upper triangle

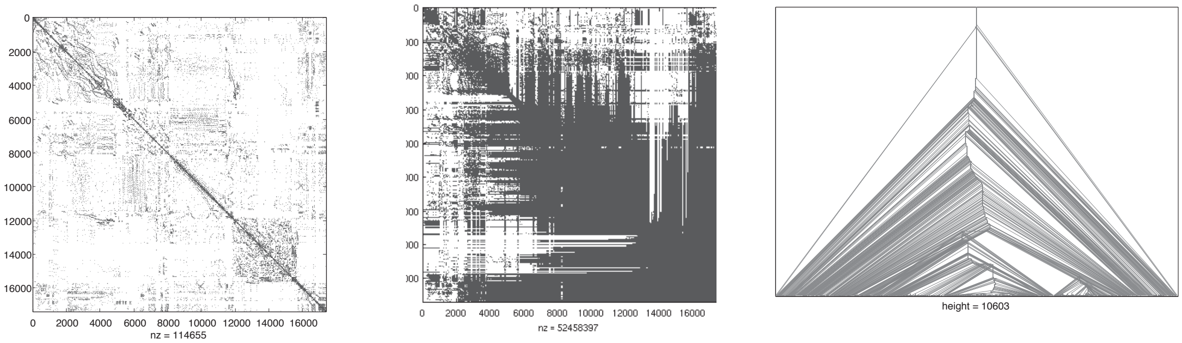
```
# Iterate over each row in the upper triangle
for $k = 0 \rightarrow N - 1$ do
    # Square root the diagonal entry of row $k$
    $U_{kk} := \sqrt{U_{kk}}$
    # Modify each entry of row $k$ other than the diagonal entry
    for $i = k + 1 \rightarrow N - 1$ do
        $U_{ki} := U_{ki} / U_{kk}$
    end for
    # Modify each entry of the remaining rows $i < k$
    for $i = k + 1 \rightarrow N - 1$ do
        for $j = i \rightarrow N - 1$ do
            $U_{ij} := U_{ij} - U_{kj} \times U_{ki}$
        end for
    end for
end for
```

In the case of iterative solvers, the decomposed system is used as a preconditioner, hence the decomposition of matrix A into a product of a lower triangular matrix and an upper triangular matrix need not be exact, that is $A = \hat{L}\hat{U}$, where $\hat{L}$ and $\hat{U}$ are approximations of L and U respectively obtained by allowing many fill-ins to be dropped. Many strategies exist to control the number and locations of fill-ins. For dropping strategies which eliminate fill-ins based on their locations, symbolic factorization is used to pre-determine their locations before factorization.

There is a commonality between LU decomposition used in direct solvers and ILU used as a preconditioner in iterative methods. Many parallelization techniques used in LU factorization are used in incomplete LU factorization. However, it is important to draw the differences between the two. In the case of a direct solver, the LU factorization step is the most time consuming step. It has a complexity of $O(n^3)$ as opposed to $O(n^2)$ for backward/forward solve (in the case of a dense matrix). Since each of these steps is executed once, a plethora of research has been geared to optimize and parallelize the more expensive stage (i.e. the factorization stage). It has been established that the factorization stage is more amenable to parallelization than the solve stage. On a multi-core processor, Hogg [114] established the fact that despite the solve stage taking fewer operations than the factorization, it has higher memory traffic and reported that 98% of the floating-point operations were performed by the factorize phase, but 20%-40% of the Level-2 cache misses were in the solve phase. Those results could have a more negative impact on the performance of PCG when using ILU as a preconditioner since the backward/solve is executed within each PCG iteration.

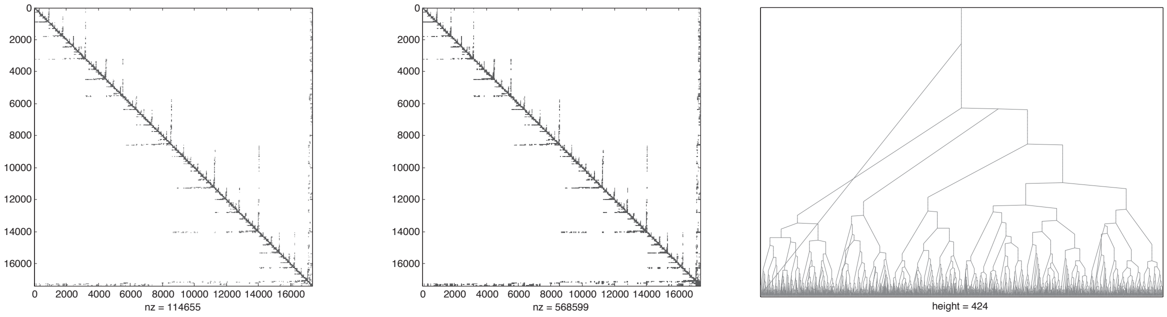
6.1.1.2 Ordering Techniques, Convergence and Parallelism

Minimum degree (MD) and Nested dissection (ND) orderings represent classes of ordering techniques that aim at reducing the number of fill-ins when factoring a matrix A into LU. MD reorders matrix columns in such a way that columns with the fewest number of non-zeros are eliminated first. Multiple Minimum degree (MMD) [115] and Approximate Minimum Degree (AMD) [116] are examples of the state-of-the-art heuristics of the MD-ordering technique. On the other hand, Nested dissection (ND) based ordering uses the *divide-and-conquer* approach by recursively subdividing the matrix's adjacency graph into disjoint subgraphs. One such heuristic is the multilevel nested dissection [117] implemented in the METIS [118] library. Another important advantage of these orderings is that they increase the degree of parallelism of the elimination process (i.e. LU factorization) since they increase the amount of independent nodes. The degree of parallelism can be depicted by constructing the elimination tree of the factorization process. Tree nodes of the same level can be eliminated simultaneously, hence it is desired to have an elimination tree that has a small height (i.e. less elimination steps) and larger number of leaves on each level (i.e. degree of parallelism). For instance, Figure 6.2 shows the sparsity of matrix BDC-1-1 before ordering and its corresponding elimination tree. The height of this tree is 10,603. After reordering BDC-1-1 using Minimum Degree (MD) and Nested dissection (ND) ordering, the heights of the corresponding elimination trees for each ordering were reduced to 424 and 311 as shown in Figure 6.3 and Figure 6.4 respectively.



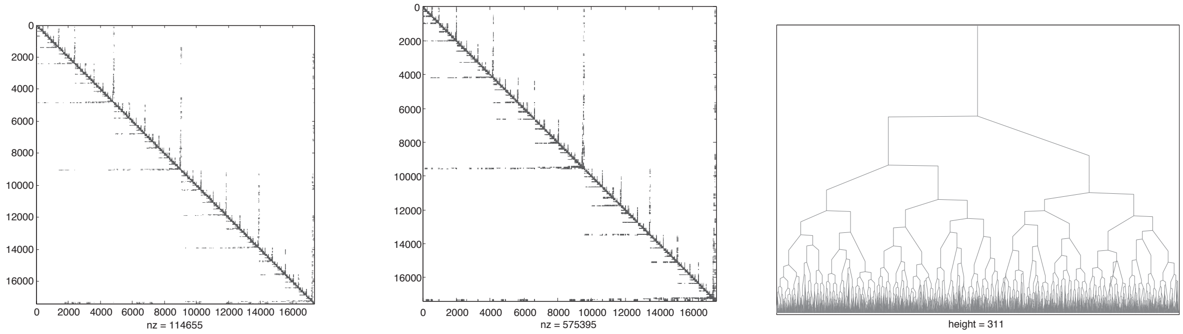
(a) Matrix A (b) $A = LU$ factorization (c) Elimination tree

Figure 6.2: **Natural ordering and LU factorization.** (a) Symmetric matrix NNZ=114,655. (b) $A = LU$ factorization, NNZ=52,458,397. (c) Elimination tree with height=10,603.



(a) Matrix A (b) $A = LU$ factorization (c) Elimination tree

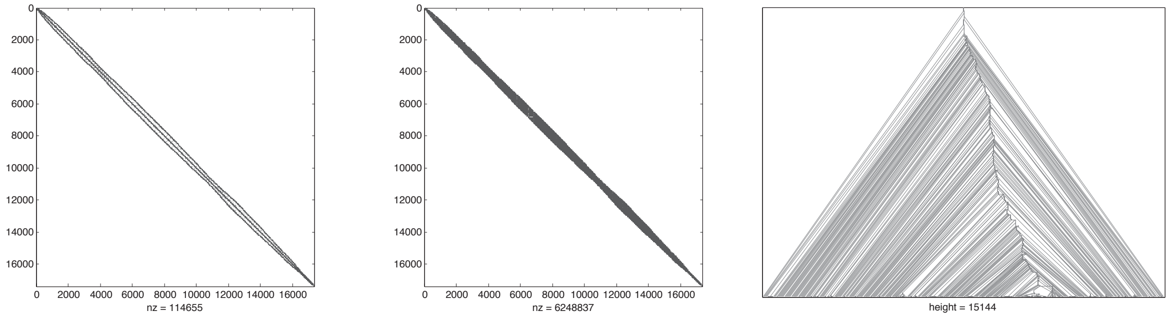
Figure 6.3: **Approximate Minimum degree ordering and LU factorization.** (a) Symmetric matrix NNZ=114,655. (b) $A = LU$ factorization, NNZ=568,599. (c) Elimination tree with height=424.



(a) Matrix A (b) $A = LU$ factorization (c) Elimination tree

Figure 6.4: **Nested Dissection ordering and LU factorization.** (a) Symmetric matrix NNZ=114,655. (b) $A = LU$ factorization, NNZ=575,395. (c) Elimination tree with height=311.

Reverse Cutill-McKee ordering (RCM) described in Chapter 5 reduces the amount of fill-ins by reducing the matrix's bandwidth as shown in Figure 6.5. Despite its advantage in reducing the amount of cache misses in SMVM, it does not lead to an elimination tree with minimum height. Little parallelism can be extracted in both the factorization and backward/forward solve using RCM.



(a) Matrix A (b) $A = LU$ factorization (c) Elimination tree

Figure 6.5: **Reverse Cuthill-McKee ordering and LU factorization.** (a) Symmetric matrix NNZ=114,655. (b) $A = LU$ factorization, NNZ=6,248,837. (c) Elimination tree with height=15,144.

The advantages of the ordering techniques discussed above in both reducing fill-ins and increasing parallelism are essential to LU factorization in direct solvers. However, in the case of iterative solvers (i.e. PCG), reordering A before ILU factorization leads to a preconditioner $\hat{L}'\hat{U}'$ that has different effect on the convergence of PCG than that obtained without ordering (i.e. $\hat{L}\hat{U}$). It has been established that ordering to reduce fill-ins or increase parallelism reduces the quality of the preconditioner and leads to more PCG iterations. Duff and Meurant [119] were the first to investigate such an effect by exhaustively performing experiments using many ordering techniques on the incomplete Cholesky - Conjugate Gradient Method (ICCG)[2]. They concluded that ordering increases the number of iterations, however, this was not related to the fact that ordering reduced the number of fill-ins, but rather to the norm of the "remainder" matrix $R = M - A$, where $M = \hat{L}\hat{U}$. This provides a good measure of convergence. However Chow and Saad [120] and Benzi et al. [121] have shown that for non-symmetric problems R may be an insufficient characterization of the convergence of the preconditioned iterative methods [122]. A large of body of research has since then attempted to understand the trade-off between convergence and parallelism [119, 123, 124, 125, 126, 127, 128]. Most results found an overall gain in reducing the solver's runtime by optimizing the ILU parallel implementation to compensate for the increased number of iterations. Saad [129] found that ILUT converges better using RCM than any other ordering.

6.2 Incomplete Cholesky Factorization

One of the dropping strategies during ILU factorization is to drop all fill-ins so that the sparsity of $\hat{L}$ and $\hat{U}$ matches that of the original matrix A. This dropping rule gives rise to an ILU(0) or IC(0) (incomplete Cholesky in the case of symmetric matrices) preconditioner [112], where the zero denotes that no fill-ins are allowed. Incomplete Cholesky with no fill-ins has been the preconditioner of choice on a desktop computer mainly due to its ability to reduce the number of iterations of a PCG while being inexpensive to produce and to compute on a desktop computer. The structures of the

[2] ICCG: a Preconditioned Conjugate Gradient Method which uses IC(0) as a preconditioner.

factors $\hat{L}$ and $\hat{L}^T$ are a priori known, making it easy to pre-allocate the storage requirement, without the need for symbolic factorization. An efficient implementation would be to duplicate the lower part of A and then perform an in-place factorization by going in an ordered manner over the entries of each row. The most efficient implementation is given by Algorithm 6.3, which is also found in SparseLib++ library [130].

Algorithm 6.3 Fan-out in-place incomplete Cholesky factorization with no fill-ins IC(0)

Input: The upper triangle matrix stored in CSR (*values*, *col_ptr* and *row_ptr*)
Output: The factored upper triangle matrix stored in CSR

```
# Iterate over each row in the upper triangle
for $k = 0 \rightarrow N - 1$ do
    # Square root the diagonal entry of row $k$
    $d := \mathit{row\_ptr}[k]$
    $z := \sqrt{\mathit{values}[d]}$
    # Modify each non-zero entry of row $k$ other than the diagonal entry
    for $i = d + 1 \rightarrow \mathit{row\_ptr}[k + 1]$ do
        $\mathit{values}[d] := \mathit{values}[d]/z$
    end for
    # For each non-zero entry of row $k$, get its column index and jump to row $j$=colmn index
    for $i = d + 1 \rightarrow \mathit{row\_ptr}[k + 1]$ do
        $z := \mathit{values}[i]$
        $h := \mathit{col\_index}[i]$
        $g = i$
        for $j = \mathit{row\_ptr}[h] \rightarrow \mathit{row\_ptr}[h + 1]$ do
            # For each non-zero entry of row $j$ with column index $c$, find the entry in row $k$ that has the same column index
            while $g < \mathit{rowPtr}[k + 1]$ and $\mathit{col\_index}[g] \leq \mathit{col\_index}[j]$ do
                if $\mathit{col\_index}[g] == \mathit{col\_index}[j]$ then
                    $\mathit{values}[j] := \mathit{values}[j] - z * \mathit{values}[g]$
                end if
                $g := g + 1$
            end while
        end for
    end for
end for
```

The results of Table 6.1 shows the execution times of incomplete Cholesky when applied on matrices obtained from 2D and 3D problems. The execution times are relativity small compared to a PCG solver execution time.

Table 6.1: **Incomplete Cholesky factorization timings on an Intel i7-860 processor.** Execution times of incomplete Cholesky when applied on matrices obtained from 2D and 3D problems. The execution times are relativity small compared to a PCG solver execution time.

Matrix	Degrees of freedom	Upper triangle NNZ	CSR storage size (MB)	IC(0) time on i7-860 (sec.)
BDC-1-0.5	38,084	147,636	1.846	0.0275
BDC-1-0.1	632,883	2,521,428	31.2697	0.4987
BDC-1-0.04	3,152,216	12,576,171	155.94	2.594
ET-0.08	38,234	293,643	3.5	0.1359
ET-0.04	409,531	3,204,372	38.233	1.554
ET-0.01R	2,666,039	21,100,983	251.65	10.3244

6.3 Complexity Analysis

The complexity of a zero fill-in incomplete LU is calculated as follows. For each matrix's row i, $\eta(i)$ operations are performed on that row where $\eta(i)$ represent the number of non-zeros of the i^{th} row. Using the fan-out algorithm, the row being eliminated causes $\eta(i)$ number of fan-outs (i.e. causes $\eta(i)$ rows to be modified), hence the number of operations for each row elimination is $\eta(i) + \eta(i) \times 2 * \eta(i) = O(\eta(i)^2)$. The total complexity to eliminate all matrix rows is $O(N * \eta(i)^2)$.

There are two types of parallelism that can be explored in ILU. The first is to perform the operations when eliminating a specific row in parallel (fine-grained parallelism). The degree of parallelism in this case is limited to the average number of non-zeros of the matrix factor $\eta(i)$. ILU complexity becomes $O(N\eta(i))$. Additional parallelism can be explored by eliminating many rows simultaneously (coarse-grained parallelism). In this case, the maximum attainable degree of parallelism is equal of the number of leaves in the elimination tree. This is maximized when the elimination tree is balanced and has a small height h (using MD ordering). The best case parallel time complexity provided $p \geq$ tree leafs is $\Omega(\eta(i)h)$ where p is the number of available processors or threads. For small number of threads, a good scheduling strategy would be to use threads at the beginning of the elimination process for coarse-grained parallelism (multiple rows elimination) and switch to fine-grained parallelism (parallelism within a row) as the elimination progresses.

6.4 Incomplete Factorization With Fill-ins

In order to improve the convergence rate of PCG beyond that provided by using the IC(0) preconditioner, much research has focused on extending the idea of the incomplete Cholesky preconditioner by allowing fill-ins to occur. There are two heuristics used to control the amount of fill-in. The first is based on a drop tolerance criterion, known as the Incomplete LU Threshold (ILUT) through which entries are dropped if their values are below a preset threshold. The second is based on the level of fill-in known as ILU(ℓ), where symbolic factorization, using graph theory, is carried out to identify the locations of the fill-ins and their level in the graph. The fill-in entries that exceed a given level are dropped. Matrix elements are assigned a level 0, hence IC(0) discards all fill-ins and the resulting factorized matrix has the same sparsity pattern as the original matrix.

ILUT algorithms perform row-by-row, upward-looking factorizations, discarding elements that are smaller than a given value. Perhaps the most popular formulation is ILUT(τ,p) [129], which employs a dual-dropping strategy. The first parameter τ is the dropping threshold, while the second parameter, p, limits the amount of fill in any row of the matrix. If a row in the original matrix A has r nonzero entries, then a maximum of $r+p$ entries are permitted in the corresponding row of the preconditioner. The limit p is sometimes applied separately to upper and lower triangular portions of the row. As with other ILU algorithms, both symmetric and non-symmetric variants exist.

6.4.1 Parallel Symbolic Factorization of ILU(ℓ)

ILU(ℓ) is an incomplete factorization technique in which a fill-in is discarded when its level is larger than a pre-specified value ℓ. The level of a fill-in (e.g. $level(U_{ij})$) depends on the level of the entries that caused it (e.g. $level(U_{kj})$ and $level(U_{ki})$), that is, $level(U_{ij}) = f(level(U_{kj}), level(U_{ki}))$. Hysom [131] presented many rules that can be used to determine such dependency. One such is the "sum" rule, in which the level of an element is calculated according to Equation 6.2. It simply states that the level of a new fill-in is the sum of the levels of the causal elements incremented by one. Since a fill-in may be updated many times, its level is always the smallest number of all calculated levels. The ultimate goal of this rule is to discard fill-ins that are small.

$$level(i, j) = \min_{1 \leq k < \min\{i,j\}} \{level(i, k) + level(k, j) + 1\} \tag{6.2}$$

The choice in this thesis to adopt the "sum" rule is because it gives rise to a symbolic factorization method that can be applied on each row independently. This can be done according to Hysom's method described in [131] and shown here in Algorithm 6.4. This algorithm takes as an input the adjacency list of the matrix A. An adjacency list $adj(i)$ of a matrix's row i contains the rows numbers dependent on i. Since the CSR storage is used to store the matrix A, then the adjacency list of A is the *col_index* of the CSR. The output of Hysom's algorithm is the adjacency list adj' which contains the locations of the entries in each row of the ILU preconditioner. The dynamic CSR storage was used in this thesis to store adj' of ILU as illustrated in Figure 6.4. The choice to use dynamic CSR was to allow each thread to apply the algorithm on each row and update the row's structure independently. When all threads finish parallel symbolic factorization, the adjacency of each row is then combined to create the CSR storage of the preconditioner ILU in $O(N)$ steps. By the end of this step, the *values* of ILU would all be set to zeros. Finally, the upper triangle A entries are copied to ILU followed by executing an efficient in-place Cholesky factorization.

Algorithm 6.4 Hysom row-structure symbolic factorization

Input: The adjacency list $adj(A)$ of A, the row number i and the level of fill-in ℓ
Output: The adjacency list of the upper factored matrix $adj'(i)$.

```
$Enqueue(Q, i)$
$length[i] := 0$
$visited[i] := i$
while $Q \neq \emptyset$ do
  $h := Dequeue(Q)$
  for $t \in adj(h)$ with $visited[t] \neq i$ do
    $visisted[t] := i$
    if $t < i$ and $length[h] < \ell$ then
      $Enqueue(Q, t)$
      $length[t] := length[h] + 1$
    else if $t > i$ then
      insert $t$ in $adj'(i)$
    end if
  end for
end while
```

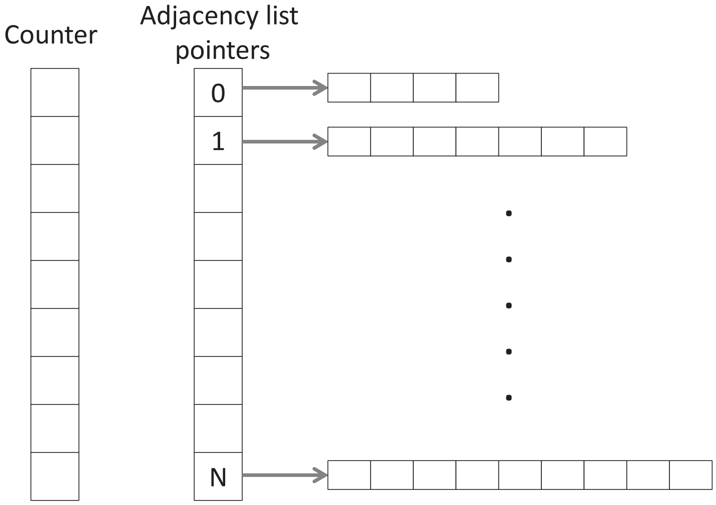


Figure 6.6: **Dynamic CSR adjacency list.** The adjacency list adj' is stored using dynamic CSR storage where each row can be dynamically allocated and expanded with little overhead.

Table 6.2 illustrates the amount of incurred fill-ins when creating ILU(1), ILU(2) and ILU(3) preconditioners. The amount of fill-ins when using natural ordering is high. One important observation is that the time it took to create the symbolic factorization for different ILU levels of the same problem was very close despite the difference in the number of fill-ins. This means that the DOF of a problem has the greatest impact on the execution time and not the operations involved in adding entries into the dynamic CSR. Such results might suggest that a higher level of ILU should be used to have a better preconditioner. However, problems might arise due to the decrease of parallelism when forward/backward solving the preconditioner, as will be investigated in the next section.

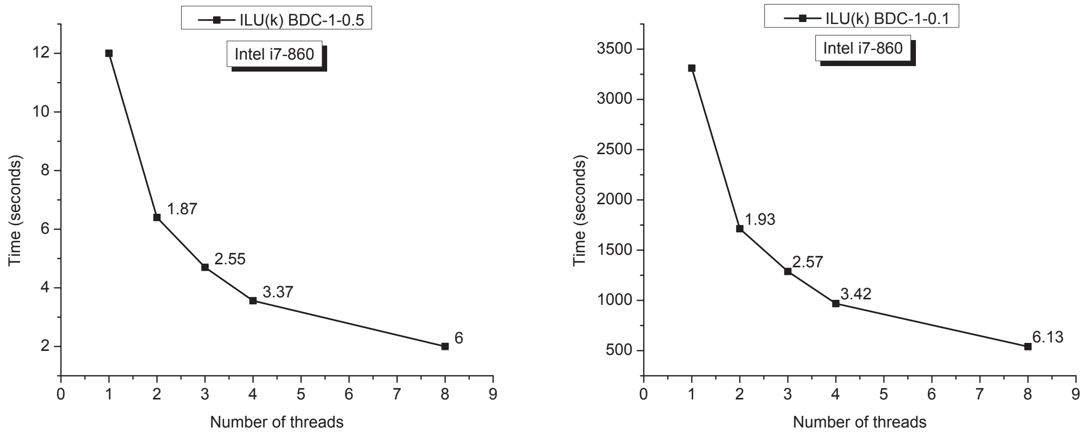
Table 6.2: **Percentage of fill-ins of ILU(1), ILU(2) and ILU(3).** The percentage of fill-ins relative to the number of non-zeros of the upper triangular matrix.

Matrix	IC(0)	ILU(1)	ILU(2)	ILU(3)
BDC-1-0.5	148,636	225,244 (51%)	321,101 (116%)	429,566 (189%)
BDC-1-0.1	2,521,428	3,848,266 (52%)	5,544,448 (120%)	7,380,933 (193%)
ET-0.08	293,643	700,914 (139%)	1,368,930 (366%)	2,555,737 (770%)
ET-0.04	3,204,372	7,927,979 (147%)	15,963,746 (398%)	30,758,154 (860%)

The multi-threaded implementation of Hysom's algorithm in this thesis also exhibited high scalability as shown in Figure 6.7. The fact that a speedup of 6 was attained beyond 4 threads (i.e. 8 threads) means that this algorithm is not memory bound but CPU cycles bound; the more execution threads can run in parallel, the more speedup can be attained. These results were consistent regardless of whether threads were assigned to different blocks of the matrix's rows (Figure 6.8a), or whether working threads executed on contiguous rows (Figure 6.8b).

6.5 Parallel Backward Solve

Backward/forward solve is an inherently sequential step in the case of dense matrices. Fortunately, this is not the case when the matrix is sparse due to less dependency between rows (or columns). When using ILU as a preconditioner, the backward/forward solve is executed once per PCG iteration. Investigating parallelism in this step is



(a) BDC-1-0.5 (DOF=38,084 NNZ=259,188 CSR=3.1MB)

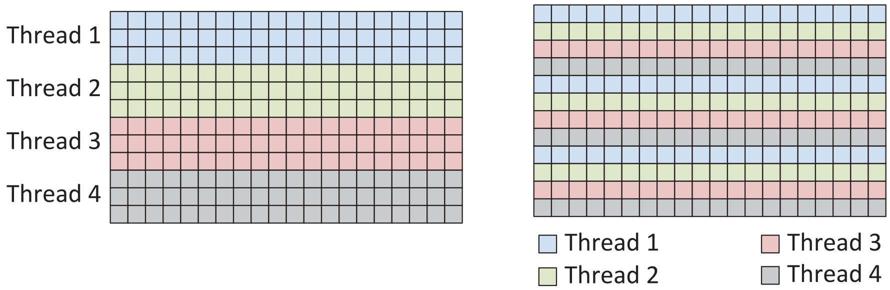
(b) BDC-1-0.1 (DOF=632,883 NNZ=4,409,973 CSR=52.88MB)

Figure 6.7: **Execution times of parallel symbolic factorization of BDC-1-0.5 and BDC-1-0.1.** The results demonstrate that the multi-threaded implementation of Hysom's algorithm is highly scalable.

important. In this thesis, a method (Algorithm 6.5) has been devised to investigate the dependency between rows. In this approach, rows of a matrix are assigned levels so that those that have the same level can be solved simultaneously. This approach is very similar to the "level scheduling" methods described in [132, 133]. Figure 6.9 illustrates an example of an upper triangular preconditioner $\hat{U}$ and the corresponding row dependency levels.

Without loss of generality, this section focuses on the backward substitution phase since the result of Algorithm 6.3 is an upper triangular matrix. Backward substitution is used to solve $Ux = y$, where y is obtained during the forward solve $Ly = b$. Equation 6.3 shows the operations in the backward solve. This is similar to one iteration of Gauss-Seidel.

$$\begin{gathered} For\, i = N,\, N-1, ..., 1 \\ x_i = \frac{b_i - \sum\limits_{k=1}^{i-1} U_{i,k} x_k}{U_{i,i}} \end{gathered} \tag{6.3}$$



(a) Assigning threads to blocks of rows

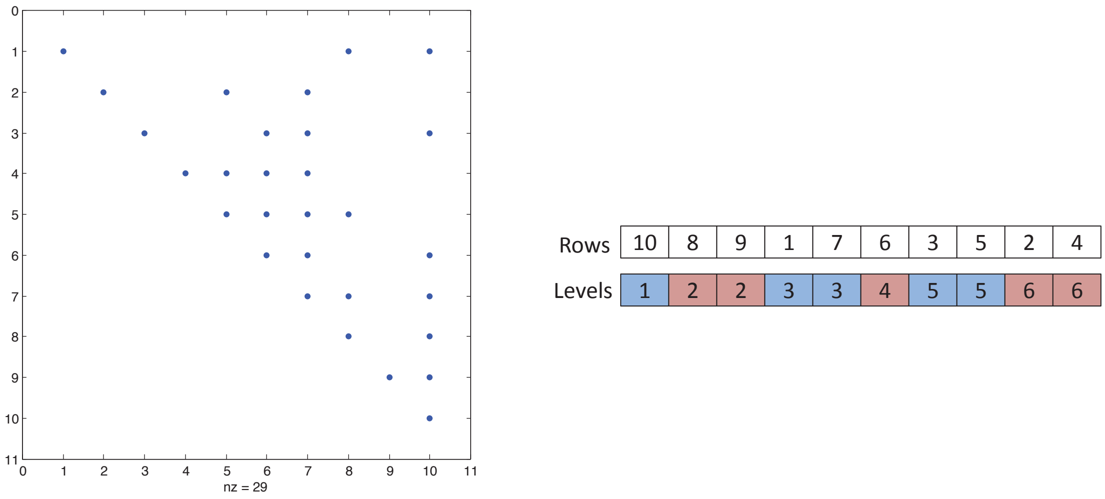
(b) Assigning threads to interleaving rows

Figure 6.8: **Threads assignment on each of the matrix rows in Hysom's algorithm.** Both thread assignment techniques resulted in a similar performance suggesting that the multi-threaded implementation is CPU cycles bound and not memory bound.

Algorithm 6.5 Backward solve dependency list

```
Input: $\hat{U}$: upper triangular incomplete Cholesky
Input: N: degrees of freedom
Output: $Rows$ and $Levels$: lists rows numbers in $Rows$ and their corresponding level
    in $Levels$
# Initialize $Levels$ to 1
$Levels[N] := 1$
# Initialize $Rows$ from 1 to N
$Rows[N] := 1...N$
for $r = N \rightarrow 1$ do
    for $p = rptr(r) \rightarrow rptr(r+1)$ do
        $j = col(p)$
        # If diagonal entry skip current loop
        if $r == j$ then
            continue
        end if
        if $Levels(j) \geq Levels(r)$ then
            $Levels(r) = Levels(j) + 1$
        end if
    end for
end for
Sort($Rows$,$Levels$) by ascending order of $Levels$.
```

To illustrate how rows of a matrix are assigned levels, Figure 6.9 shows an upper triangular matrix and its corresponding scheduling levels. Since it is a backward solve, the first row to be solved is 10. Then both rows 8 and 9 can be solved simultaneously as both depend on row 10 but are independent of each other. Row 10 has been assigned "level 1" and rows 8 and 9 have been assigned "level 2". The same logic applies to the remaining rows leading to the *rows dependency list* shown in Figure 6.9b. The fact that the number of levels assigned in this example is 6, means that the backward solve can be achieved in 6 steps. The maximum degree of parallelism attainable is 2 since at any solve step, a maximum of 2 rows can be solved simultaneously.



Rows	10	8	9	1	7	6	3	5	2	4
Levels	1	2	2	3	3	4	5	5	6	6

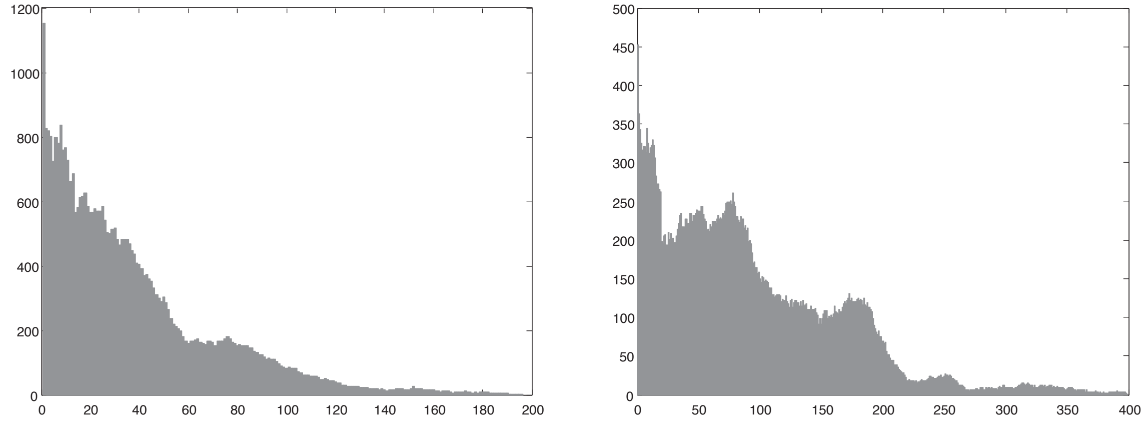
(a) Upper triangular matrix

(b) Rows dependency list

Figure 6.9: ***Rows dependency lists* of a backward solve.** (a) Upper triangular factor U. (b) The corresponding *rows dependency lists*.

The next step is to investigate the degree of parallelism that can be attained when backward solving a preconditioner obtained from 2D and 3D finite element problems. Particularly, the *rows dependency lists* will be calculated for ILU(1) and ILU(3) of the BDC-1-0.5 (a 2D finite element problem) and the ET-0.08 (a 3D finite element problem) matrices. Since it is not possible to draw the *rows dependency list* for ILU(1) and ILU(3), instead, a histogram will be used to depict the maximum degree of parallelism and the number of steps required to backward solve each of the preconditioners. The x-axis shows the number of steps required to backward solve a matrix, and the y-axis of the histogram shows the number of rows that can be solved simultaneously at a given step.

Figure 6.10 shows the rows dependency histograms of ILU(1) and ILU(3) of the matrix BDC-1-0.5. This matrix which was obtained from a 2D first-order nodal finite element formulation exhibited a high degree of parallelism. The maximum degree of parallelism of ILU(1) was 1,151 and the number of steps required to backward solve the preconditioner was 196. On the other hand, the ILU(3) preconditioner of the same problem had a maximum degree of parallelism equal to 453 and 399 steps were required to backward solve it. As anticipated, the more fill-ins that existed in a preconditioner, the less parallelism could be exploited.

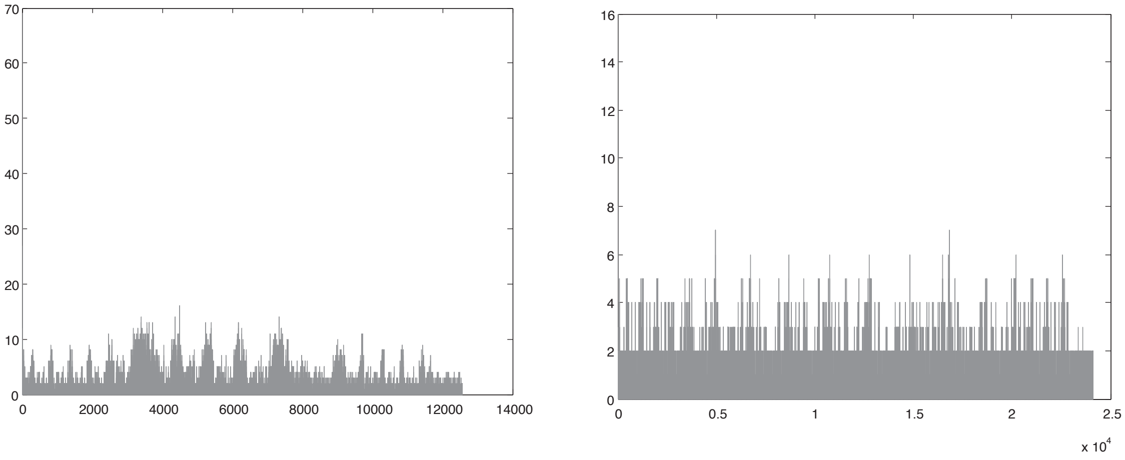


(a) BDC-1-0.5 - Degree of parallelism of ILU(1)

(b) BDC-1-0.5 - Degree of parallelism of ILU(3)

Figure 6.10: **BDC-1-0.5 - Degree of parallelism of backward solving ILU(1) and ILU(3) preconditioners.** The maximum degree of parallelism of ILU(1) was 1,151 and the number of steps required to backward solve is 196. On the other hand the ILU(3) preconditioner of the same problem had a maximum degree of parallelism = 453 and it required 399 steps to complete the backward solve.

Backward/Froward solving a preconditioner obtained from a 3D problem is less amenable to parallelism than that obtained from a 2D problem. For instance, an ILU(1) preconditioner obtained from ET-0.08 (3D transformer problem) can be solved in 12,559 steps where the maximum number of rows that could be solved simultaneously is only 16 and an ILU(3) preconditioner of the same problem can be solved in 24,125 steps where the maximum attainable degree of parallelism is only 16. As investigated in the previous section, a 2D problem that has the same number of degrees of freedom as ET-0.08 (i.e. BDC-1-0.5) was more amenable to parallelism.



(a) ET-0.08 - Degree of parallelism of ILU(1) (b) ET-0.08 - Degree of parallelism of ILU(1)

Figure 6.11: **ET-0.08 (Natural ordering) - Degree of parallelism of backward solving ILU(1) and ILU(3) preconditioners.** The maximum degree of parallelism of ILU(1) was 16 and the number of steps required to backward solve is 12,559. On the other hand, the ILU(3) preconditioner of the same problem had a maximum degree of parallelism = 16 and it required 24,559 steps to complete the backward solve.

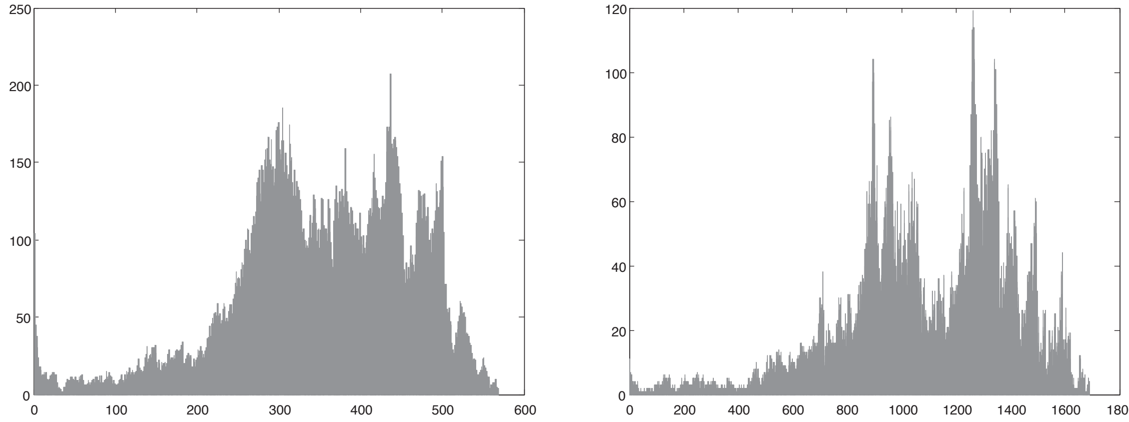
Since the preconditioner obtained from the 3D transformer problem exhibited a low degree of parallelism, approximate minimum degree (AMD) and Reverse Cuthill–McKee (RCM) orderings were first applied on the ET-0.08 matrix before generating the ILU(1) and ILU(3) preconditioners. The number of non-zeros in the upper triangle preconditioner, the maximum degree of parallelism and the backward solving steps of both preconditioners ILU(1) and ILU(3) using different orderings are summarized in Table 6.3 and Table 6.4 respectively. The histogram of the degree of parallelism is also shown in Figure 6.12 for ILU(1) and in Figure 6.13 for ILU(3). As anticipated, the AMD ordering resulted in a relatively higher parallelizable backward solver than the Natural and RCM orderings. However, as discussed before, this ordering might degrade PCG convergence. On the other hand, RCM ordering exhibited a similar degree of parallelism to that of the Natural ordering but required less solve steps. This implies that there is a balance in the degree of parallelism among steps, which will translate into a balanced threads utilization.

Table 6.3: **ILU(1) of ET-0.08-Degree of parallelism of backward/forward solve using different orderings.** Showing the effect of ordering on the number of non-zeros, degree of parallelism and the number of steps to perform backward/forward solve.

	NNZ	Max. degree of parallelism	Solving steps
ILU(1)-Natural ordering	700,914	69	12,559
ILU(1)-AMD ordering	673,511	207	569
ILU(1)-RCM ordering	585,646	36	3,401

Table 6.4: **ILU(3) of ET-0.08-Degree of parallelism of backward/forward solve using different orderings.** Showing the effect of ordering on the number of non-zeros, degree of parallelism and the number of steps to perform backward/forward solve.

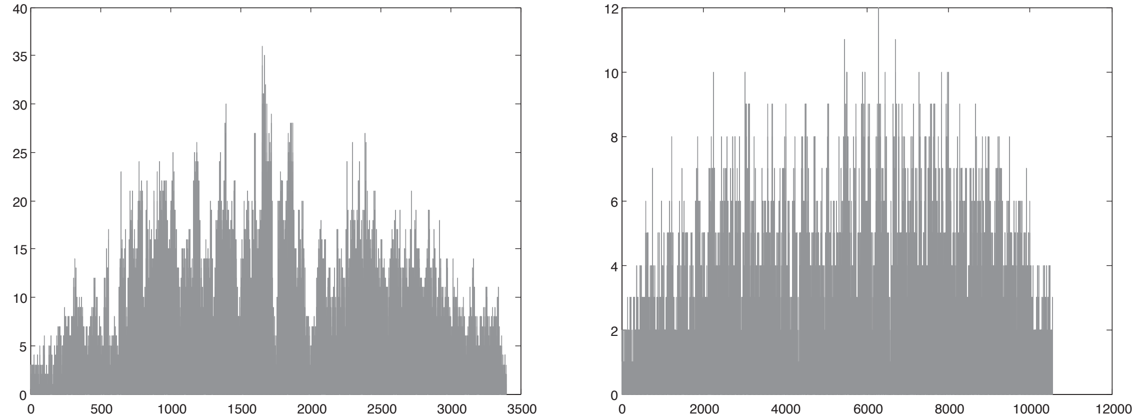
	NNZ	Max. degree of parallelism	Solving steps
ILU(3)-Natural ordering	2,555,737	16	24,125
ILU(3)-AMD ordering	1,937,345	119	1,690
ILU(3)-RCM ordering	1,984,093	12	10,548



(a) ET-0.08 (AMD) - Degree of parallelism of ILU(1)

(b) ET-0.08 (AMD) - Degree of parallelism of ILU(3)

Figure 6.12: **ET-0.08 (AMD ordering) - Degree of parallelism of backward solving with ILU(1) and ILU(3) preconditioners.** The maximum degree of parallelism of ILU(1) was 207 and the number of steps required to backward solve is 569. On the other hand, the ILU(3) preconditioner of the same problem had a maximum degree of parallelism = 119 and it required 1,690 steps to complete the backward solve.



(a) ET-0.08 (RCM) - Degree of parallelism of ILU(1)

(b) ET-0.08 (RCM) - Degree of parallelism of ILU(3)

Figure 6.13: **ET-0.08 (RCM ordering) - Degree of parallelism of backward solving with ILU(1) and ILU(3) preconditioners.** The maximum degree of parallelism of ILU(1) was 36 and the number of steps required to backward solve is 3,401. On the other hand, the ILU(3) preconditioner for the same problem had a maximum degree of parallelism = 12 and it required 10,548 steps to complete the backward solve.

6.6 Concluding Remarks

Applying an ILU preconditioner in a PCG method incur two types of overheads. The first occurs during the preconditioner setup stage where an incomplete factorization is performed on the matrix A, where A is the coefficient matrix of a system of equations $Ax = b$. The second overhead occurs when backward/forward solving the preconditioner obtained in the first stage withing each PCG iteration.

The incomplete Choleksy IC(0) preconditioner has been the state-of-the-art preconditioner, hence, the work in this thesis considers that a factorization problem that can be reduced to that of an IC(0) to be efficient. Subsequently, the focus was to elucidate a parallel symbolic factorization technique to know the locations of non-zeros in the preconditioner so that efficient in-place incomplete Cholesky factorization can be preformed. The parallel symbolic factorization implemented has been shown to be scalable and its execution time to be dependent of the problem size, but not on the level of fill-ins of a preconditioner. However, the overall time to setup an ILU preconditioner remains high relative to a PCG iterative solver. More efficient multi-threaded implementation is required to optimize this process.

The preconditioner solve stage within each PCG iteration has been shown to be a bottleneck as the problem size increases where in such a case, less parallelism can be explored. Among the ordering techniques investigated, Reverse Cuthill-McKee (RCM) has been shown to be a good overall choice to decrease the execution time of a PCG method. Although it does not increase the degree of parallelism in the preconditioner solve stage, it does however balance the parallel backward/forward solve workload in a multi-threaded application. That in addition to its advantage over other ordering techniques as it increases the convergence rate of PCG and decreases the cache misses in SMVM.

Chapter 7

Conclusion and Future Work

7.1 Discussion and Conclusion

7.1.1 Inter-Compatibility of FEM Kernel Optimization Techniques

This thesis predominantly emphasizes speedup analyses of a given kernel in terms of the effect it exerts on the whole analysis process. It thereby places considerable focus on investigating the possibility of utilizing the same sparse storage scheme in all analyzed kernels and methods.

Given the dependency of the sparse storage upon the problem structure, it is important to devise matrix test sets that are relevant to the problem domain (i.e. low frequency electromagnetic analysis using the Finite Element Method). The fact that matrices resulting from FEM do not have a large discrepancy in the number of non-zeros per row makes it possible, as has been previously shown, to use the HYB storage (ELLPACK + COO storage) in most kernels – with the exception of the ILU(ℓ) symbolic factorization cases.

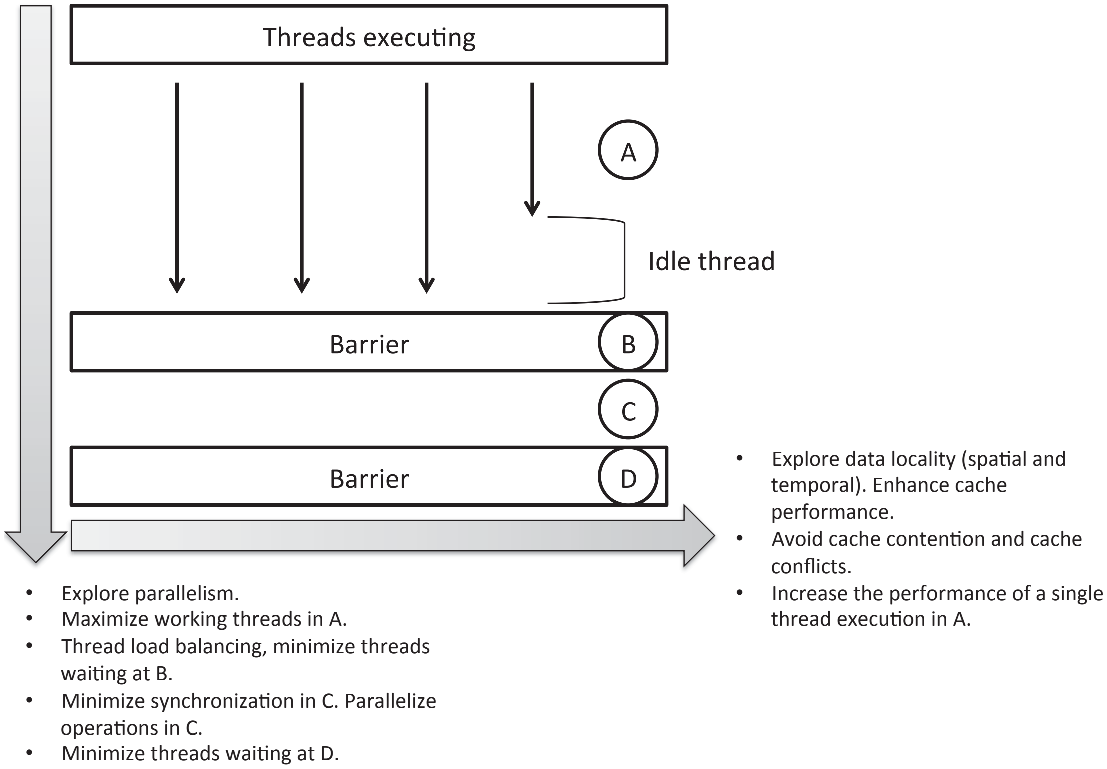
The time duration required for a kernel in the analysis stage (SMVM, matrix assembly, mesh generation, etc.) to complete run is typically short (a few seconds). However, problems arise during the analysis of a design, when these kernels are called for execution more than once. For instance, a typical field analysis software refines a mesh and solves it, using iterative solvers such as CG, a number of times during a single call to a design analysis. Furthermore, the CG algorithm iterates a large number of times over kernels such as SMVM, vector update and inner-products. This fact (i.e. short kernels runtime) renders kernel optimization (by altering the sparse data structure or thread re-distribution) an ineffective process since it incurs a large overhead.

7.1.2 Different Types and Degree of Parallelism

7.1.3 Issues in Fine-Grained Parallelism on Multi-Core Processors

Finite Element Method sequential algorithms and their implementations serve as the basic building block of multi-threaded algorithms being optimized for multi-core processors. As such, the optimization of these sequential algorithms is of great importance when enhanced performance on multi-core processors is desired.

Assuming that sequential algorithms used in FEM are amenable to parallelization that exhibits a large degree of parallelism, which is in fact the case for most kernels with the exception of the preconditioner and backward/forward solve, two important challenges should be overcome in order to achieve performance gain in multi-threaded algorithms. The first challenge is synchronization cost and the second is the effect of running multiple threads on cache performance. Those two concerns are depicted in Figure 7.1 which is a typical execution flow of a fine-grained multi-threaded algorithm - such as the Sliding-Window Gauss-Seidel method.



- Explore parallelism.
- Maximize working threads in A.
- Thread load balancing, minimize threads waiting at B.
- Minimize synchronization in C. Parallelize operations in C.
- Minimize threads waiting at D.

- Explore data locality (spatial and temporal). Enhance cache performance.
- Avoid cache contention and cache conflicts.
- Increase the performance of a single thread execution in A.

Figure 7.1: **Exploring and Exploiting Parallelism in a typical fine-grained multi-threaded algorithm.**

The effect of synchronization on the performance of multi-threaded algorithms can be further divided into two aspects: the first is the number of times a thread has to wait at a barrier or wait to enter a *critical section* (i.e. stages “B” and “D” of Figure 7.1), and the second is the amount of work involved in synchronizing data between threads (i.e. stage “C”). The latter is less critical for multi-core processors than for multi-processors due to cache sharing. The former, on the other hand, has been shown to degrade performance particularity when the number of synchronization points increases with the problem size and the execution runtime of threads (stage “A”) is small relative to the synchronization stage (stage “D”).

Cache sharing among threads is advantageous in that it allows fast synchronization among threads, however, in the case where multiple threads modify a read-write data structure simultaneously (i.e. matrix assembly and symbolic factorization), true or false cache sharing can occur. On the other hand, cache misses have been reduced when multiple threads shared a read-only data structure; such as the case of the SMVM kernel (i.e. accessing the read-only matrix A and vector X in $Y = A \times X$).

7.2 Future Work

7.2.1 Design Optimization Using Coarse-Grained Parallelism

The design process is an optimization process wherein a multi-dimensional search space is explored to yield improved performance of a multi-objective function. Any instance of a design – or point in the search space - requires a call to a “field analysis” tool; a rather time-consuming step. Reducing the number of calls has been widely investigated employing many strategies that use either a stochastic (i.e. evolutionary strategy [134, 135], neuro-fuzzy [136], surrogate modeling [137, 138, 139, 140], etc) or deterministic approach (i.e. minimal function calls) [141, 142].

The work in this thesis has demonstrated that in the attempt to gain speed in every kernel of the field analysis, various challenges are encountered before an overall gain in the design process is accomplished. This raises a question of whether it would be more productive - in terms of reducing the design process time - for a design engineer to use coarse-grained parallelism through which multiple design instances are investigated simultaneously (i.e. exploring the design space simultaneously). In such computational model, many design models can share a read-only structure, such as part of a device or material properties which are common among many designs.

7.2.2 Analysis Post-Processing and Visualization

This thesis investigates many kernel operations in the “processing” stage of FEM; a stage that remains the most time consuming one in field analysis. The “post-processing’ stage is beyond the scope of this thesis but this should not imply that it is less important that its precedent; it involves inverse mapping the solution obtained from the “processing” stage and plotting the results on the design model and it is relatively time efficient when one field analysis is considered at a time. However, since the work in this thesis, and the conclusion, advocate for the simulation of multiple designs simultaneously during which a design engineer could be visualizing the result of some designs, while other analysis are executing in the background, it would be important to analyze the computational resources required by the “post-processing” stage involving running multiple design result evaluations and visualizations.

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, *et al.*, "The landscape of parallel computing research: A view from Berkeley," *EECS Department University of California Berkeley Tech Rep UCBEECS2006183*, vol. 18, no. UCB/EECS-2006-183, 2006.

[2] T. Tortschanoff, "Survey of numerical methods in field calculations," vol. MAG-20, pp. 1912–1917, 1984.

[3] L. Finkelstein and A. C. W. Finkelstein, "Review of design methodology," *IEE Proceedings A: Physical Science. Measurement and Instrumentation. Management and Education. Reviews*, vol. 130, pp. 213–222, 1983.

[4] G. B. R. Feilden, S. H. Grylls, and M. C. De Malherbe, "The Feilden's report on egnineering design," tech. rep., Her Majesty's Stationary Office, 1963.

[5] N. F. O. Erbuomwan, S. Sivaloganathan, and A. Jebb, "A survey of design philosophies, models, methods and systems," *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, vol. 210, no. 4, pp. 301–320, 1996.

[6] J. Clarkson and C. Eckert, *Design process improvement: a review of current practice.* Springer Verlag, 2005.

[7] V. V. Kryssanov, H. Tamaki, and S. Kitamura, "Understanding design fundamentals: How synthesis and analysis drive creativity, resulting in emergence," *Artificial Intelligence in Engineering*, vol. 15, no. 4, pp. 329–342, 2001.

[8] V. Goel, "Ill-structured representations for ill-structured problems," in *Proceedings of the Fourteenth Annual Conference of the cognitive Science Society*, vol. 14, pp. 130–135, Lawrence Erlbaum, 1992.

[9] A. Macnee, "An electronic differential analyzer," *Proceedings of the IRE*, vol. 37, no. 11, pp. 1315–1324, 1949.

[10] D. E. Weisberg, *The Engineering Design Revolution: The People, Companies and Computer Systems That Changed Forever the Practice of Engineering.* 2008.

[11] S. Nash, *A History of scientific computing.* ACM New York, NY, USA, 1990.

[12] T. Oden, "Some historic comments on finite elements," in *Proceedings of the ACM conference on History of scientific and numeric computation*, pp. 125–130, ACM, 1987.

[13] H. Goldstine, "Remembrance of things past," in *A history of scientific computing*, pp. 5–16, ACM, 1990.

[14] G. David, "The eniac, the verb "to program" and the emergence of digital computers," vol. 18, pp. 51–55, 1996. IEEE Annals of the History of Computing.

[15] J. Fjortoft and J. von Neumann, "Numerical integration of the barotropic vorticity equation," *Tellus*, vol. 2, pp. 237–254, 1950.

[16] H. El-Rewini and M. Abd-El-Barr, *Advanced computer architecture and parallel processing*. Wiley-Interscience, 2005.

[17] M. M. Gold, "Time-sharing and batch-processing: an experimental comparison of their values in a problem-solving situation," *Communications of the ACM*, vol. 12, no. 5, pp. 249–259, 1969.

[18] R. Bousquet and D. Yates, "A low cost interactive graphics system for large scale finite element analyses," *Computers & Structures*, vol. 3, no. 6, pp. 1321–1330, 1973.

[19] W. Woodward and J. Morris, "Improving productivity in finite element analysis through interactive processing," *Finite elements in analysis and design*, vol. 1, no. 1, pp. 35–48, 1985.

[20] J. Shewchuk, "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator," *Applied Computational Geometry Towards Geometric Engineering*, pp. 203–222, 1996.

[21] Intel, "Intel VTune Amplifier XE 2011." http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/, 2011.

[22] Infolytica, "MagNet: 2D/3D Electromagnetic Field Simulation Software (version 7)." http://www.infolytica.com/en/products/magnet/, 1978-2012.

[23] J. Shewchuk, "Lecture notes on Delaunay mesh generation," 1999.

[24] B. Delaunay, "Sur la sphere vide," *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, pp. 793–800, 1934.

[25] D. Lee and B. Schachter, "Two algorithms for constructing a Delaunay triangulation," *International Journal of Parallel Programming*, vol. 9, no. 3, pp. 219–242, 1980.

[26] S. Fortune, "A sweepline algorithm for voronoi diagrams," *Algorithmica*, vol. 2, no. 1, pp. 153–174, 1987.

[27] C. Lawson, "Software for c1 interpolation," *Mathematical Software III*, pp. 161–194, 1977.

[28] A. Bowyer, "Computing dirichlet tessellations," *The Computer Journal*, vol. 24, no. 2, p. 162, 1981.

[29] D. Watson, "Computing the n-dimensional Delaunay tessellation with application to voronoi polytopes," *The Computer Journal*, vol. 24, no. 2, p. 167, 1981.

[30] L. Guibas, D. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, no. 1, pp. 381–413, 1992.

[31] M. Shamos, *Computational geometry.* PhD thesis, 1978.

[32] K. Clarkson and P. Shor, "Applications of random sampling in computational geometry, II," *Discrete & Computational Geometry*, vol. 4, no. 1, pp. 387–421, 1989.

[33] F. Razafindrazaka, "Delaunay triangulation algorithm and application to terrain generation," Master's thesis, United Nations University, Macao, 2009.

[34] J. R. Shewchuk, "Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator (version 1.6)." http://www.cs.cmu.edu/~quake/triangle.html/, 1996–2005.

[35] L. Chew, "Guaranteed-quality triangular meshes," Tech. Rep. TR-89-983, Department of Computer Science, Cornell University, 1989.

[36] L. Chew, "Guaranteed-quality mesh generation for curved surfaces," in *Proceedings of the ninth annual symposium on Computational geometry*, pp. 274–280, ACM, 1993.

[37] J. Ruppert, "A Delaunay refinement algorithm for quality 2-dimensional mesh generation," *Journal of Algorithms*, vol. 18, no. 3, pp. 548–585, 1995.

[38] J. Shewchuk, *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, 1997.

[39] J. Boissonnat, O. Devillers, and S. Hornus, "Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension," in *Proceedings of the 25th annual symposium on Computational geometry*, pp. 208–216, ACM, 2009.

[40] C. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. Nikolopoulos, and N. Chrisochoides, "Multigrain parallel Delaunay mesh generation: challenges and opportunities for multithreaded architectures," in *Proceedings of the 19th annual international conference on Supercomputing*, pp. 367–376, ACM, 2005.

[41] T. Tu, *A Scalable Database Approach to Computing Delaunay Triangulations.* PhD thesis, June 2008.

[42] N. Chrisochoides, "A survey of parallel mesh generation methods," *Numerical Solutions of Partial Differential Equations on Parallel Computers*, 2005.

[43] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains," in *Proceedgins of the 18th Symposium on Computational Geomtry*, pp. 135–144, 2002.

[44] M. Kulkarni, L. Chew, and K. Pingali, "Using transactions in Delaunay mesh generation," in *Workshop on Transactional Memory Workloads*, 2006.

[45] V. Batista, D. Millman, S. Pion, and J. Singler, "Parallel geometric algorithms for multi-core computers," *Computational Geometry*, vol. 43, no. 8, pp. 663–677, 2010.

[46] S. Pion and M. Teillaud, "3D triangulations," *CGAL Editorial Board, editor, CGAL-3.4 User and Reference Manual*, 2008.

[47] N. Chrisochoides and D. Nave, "Parallel Delaunay mesh generation kernel," *International Journal for Numerical Methods in Engineering*, vol. 58, no. 2, pp. 161–176, 2003.

[48] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains," *Journal of computational geomtry*, no. 28, pp. 191–215, 2004.

[49] J. Galtier and P. George, "Prepartitioning as a way to mesh subdomains in parallel," in *Proceedings of the 5th International Meshing Roundtable*, pp. 107–121, 1996.

[50] L. Linardakis, *Decoupling Method for Parallel Delaunay 2D Mesh Generation.* PhD thesis, The College of William and Mary, 2007.

[51] L. P. Chew, "Constrained Delaunay triangulations," *Algorithmica (New York)*, vol. 4, pp. 97–108, 1989.

[52] A. N. Chernikov and N. P. Chrisochoides, "Algorithm 872: Parallel 2D constrained Delaunay mesh generation," *ACM Transactions on Mathematical Software*, vol. 34, no. 1, 2008.

[53] C. Antonopoulos, F. Blagojevic, A. Chernikov, N. Chrisochoides, and D. Nikolopoulos, "Algorithm, software, and hardware optimizations for Delaunay mesh generation on simultaneous multithreaded architectures," *Journal of Parallel and Distributed Computing*, vol. 69, no. 7, pp. 601–612, 2009.

[54] D. Spielman, S. Teng, and A. Üngör, "Parallel Delaunay refinement: Algorithms and analyses.," in *Proceedings of the International Meshing Raoundtable*, pp. 205–217, 2002.

[55] D. Spielman, S. Teng, and A. Üngör, "Time complexity of practical parallel steiner point insertion algorithms," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, p. 268, ACM, 2004.

[56] D. Spielman, S. Teng, A. Ungor, and M. Goodrich, "Parallel Delaunay refinement: Algorithms and analyses," *International Journal of Computational Geometry and Applications*, vol. 17, no. 1, pp. 1–30, 2007.

[57] C. B. Chapman and M. Pinfold, "The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure," *Advances in Engineering Software*, vol. 32, no. 12, pp. 903 – 912, 2001.

[58] J. Jin, *The finite element method in electromagnetics.* Wiley New York, 2 ed., 2002.

[59] A. C. Polycarpou, *Intrdocution to the finite element method in eletromagntics.* 1 ed., 2006.

[60] R. Grimes, D. Young, and D. Kincaid, "Itpack 2.0: User's guide," Tech. Rep. CNA-150, Center for Numerical Analysis,University of Texas, Austin, Texas, August 1979.

[61] A. Buchau, S. M. Tsafak, W. Hafla, and W. M. Rucker, "Parallelization of a fast multipole boundary element method with cluster openmp," *Magnetics, IEEE Transactions on*, vol. 44, no. 6, pp. 1338–1341, 2008.

[62] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[63] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-and many-core architectures," *International Journal for Numerical Methods in Fluids*, 2011.

[64] R. Natarajan, "Finite element applications on a shared-memory multiprocessor: Algorithms and experimental results," *Journal of Computational Physics*, vol. 94, no. 2, pp. 352–381, 1991.

[65] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451–460, 2009.

[66] E. Tejada and T. Ertl, "Large steps in GPU-based deformable bodies simulation," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 703–715, 2005.

[67] D. Fernández, M. Dehnavi, W. Gross, and D. Giannacopoulos, "Alternate parallel processing approach for FEM," *Magnetics, IEEE Transactions on*, vol. 48, no. 2, pp. 399–402, 2012.

[68] B. Smith, P. Bjørstad, and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 2004.

[69] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, November 2007.*

[70] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," *NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004*, 2008.

[71] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[72] D. Fernandez, D. Giannacopoulos, and W. Gross, "Efficient multicore sparse matrix-vector multiplication for FE electromagnetics," *Magnetics, IEEE Transactions on*, vol. 45, no. 3, pp. 1392–1395, 2009.

[73] M. Dehnavi, D. Fernández, and D. Giannacopoulos, "Finite-element sparse matrix vector multiplication on graphic processing units," *Magnetics, IEEE Transactions on*, vol. 46, no. 8, pp. 2982–2985, 2010.

[74] T. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[75] G. Golub and D. O'Leary, "Some history of the conjugate gradient and Lanczos algorithms: 1948-1976," *SIAM review*, vol. 31, no. 1, pp. 50–102, 1989.

[76] G. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[77] P. Wesseling, "Introduction to multigrid methods," 1995.

[78] J. Ouyang and D. Lowther, "A hybrid design model for electromagnetic devices," *Magnetics, IEEE Transactions on*, vol. 45, no. 3, pp. 1442–1445, 2009.

[79] J. Ouyang and D. Lowther, "The use of case-based reasoning in creating a prototype for electromagnetic device optimization," *Magnetics, IEEE Transactions on*, vol. 46, no. 8, pp. 3377–3380, 2010.

[80] J. Ouyang and D. Lowther, "Towards a case-based computational model for the creative design of electromagnetic devices," in *Computation in Electromagnetics (CEM 2011), IET 8th International Conference on*, pp. 1 –2, april 2011.

[81] C. Weiss, W. Karl, M. Kowarschik, and U. Rude, "Memory characteristics of iterative methods," in *Supercomputing, ACM/IEEE 1999 Conference*, pp. 31–31, 1999.

[82] Q. Huang, J. Xue, and X. Vera, "Code tiling for improving the cache performance of PDE solvers," *ICPP'03: Proceedings of the International Conference on Parallel Processing*, 2003.

[83] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," *Proceedings of the 2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC 2006*, pp. 51–60, 2006.

[84] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[85] J. Treibig, G. Wellein, and G. Hager, “Efficient multicore-aware parallelization strategies for iterative stencil computations,” *Journal of Computational Science*, vol. 2, no. 2, pp. 130–137, 2011.

[86] L. Hayes, *Comparative analysis of iterative techniques for solving Laplace’s equation on the unit square on a parallel processor*. PhD thesis, 1974.

[87] J. Lambiotte, *The solution of linear systems of equations on a vector computer*. PhD thesis, 1975.

[88] J. Adams and A. J. Ortega, “A multicolor SOR method for parallel computation,” *Proceedings of the International Conference on Parallel Procession*, pp. 53–56, 1982.

[89] G. Baudet, “Asynchronous iterative methods for multiprocessors,” *Journal of the ACM (JACM)*, vol. 25, no. 2, pp. 226–244, 1978.

[90] N. Patel and H. Jordan, “Parallelized point rowwise successive over-relaxation method on a multiprocessor,” *Parallel Computing*, vol. 1, no. 3, pp. 207–222, 1984.

[91] J. Bonomo and W. Dyksen, “Pipelined iterative methods for shared memory machines,” *Parallel Computing*, vol. 11, no. 2, pp. 187–199, 1989.

[92] J. M. Ortega and R. G. Voigt, “Solution of partial differential equations on vector and parallel computers,” *SIAM review*, vol. 27, no. 2, pp. 149–240, 1985.

[93] B. Dolwithayakul, C. Chantrapornchai, and N. Chumchob, “GPU-based total variation image restoration using Sliding Window Gauss-Seidel algorithm,” in *Intelligent Signal Processing and Communications Systems (ISPACS), 2011 International Symposium on*, pp. 1–6, IEEE, 2011.

[94] J. Zhang and J. Yuan, “The strategy for optimizing the performance of Gauss-Seidel iterative algorithm based on register reuse,” *Advanced Science Letters*, vol. 7, no. 1, pp. 73–77, 2012.

[95] M. R. Hestenes and E. Stiefel, “Methods of conjugate gradients for solving linear systems,” *J. Res. Nat. Bur. Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[96] M. R. Hestenes, “Iterative methods for solving linear equations,” *Journal of Optimization Theory and Applications*, vol. 11, no. 4, pp. 323–334, 1973.

[97] S. Toledo, “Improving the memory-system performance of sparse-matrix vector multiplication,” *IBM Journal of research and development*, vol. 41, no. 6, pp. 711–725, 1997.

[98] R. Vuduc, *Automatic performance tuning of sparse matrix kernels*. PhD thesis, 2003.

[99] E. Im, *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, 2000.

[100] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *Journal of Supercomputing*, vol. 50, no. 1, pp. 36–77, 2009.

[101] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2008*, pp. 283–292, 2008.

[102] G. Schubert, G. Hager, and H. Fehske, "Performance limitations for sparse matrix-vector multiplications on current multi-core environments," in *High Performance Computing in Science and Engineering, Garching/Munich 2009* (S. Wagner, M. Steinmetz, A. Bode, and M. M. Müller, eds.), pp. 13–26, Springer Berlin Heidelberg, 2010.

[103] Y. Saad, *Iterative methods for sparse linear systems.* Society for Industrial Mathematics, 2003.

[104] R. Shahnaz, A. Usman, and I. R. Chughtai, "Review of storage techniques for sparse matrices," in *9th International Multitopic Conference, IEEE INMIC 2005*, pp. 1–7, 2005.

[105] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2011.

[106] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, pp. 157–172, ACM, 1969.

[107] 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, 1995.

[108] O. Johnson, C. Micchelli, and G. Paul, "Polynomial preconditioners for conjugate gradient calculations," *SIAM Journal on Numerical Analysis*, pp. 362–376, 1983.

[109] M. DeLong, *SOR as a preconditioner.* PhD thesis, University of Virginia, 1997.

[110] J. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[111] R. Varga, "Factorization and normalized iterative methods," tech. rep., Westinghouse Electric Corp. Bettis Plant, Pittsburgh, 1959.

[112] J. Meijerink and H. van der Vorst, "An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix," *Mathematics of Computation*, pp. 148–162, 1977.

[113] J. Liu, “The role of elimination trees in sparse factorization,” *SIAM Journal on Matrix Analysis and Applications*, vol. 11, p. 134, 1990.

[114] J. Hogg and J. Scott, “A note on the solve phase of a multicore solver,” tech. rep., RAL-TR-2010-007, 2010.

[115] A. George and J. Liu, “The evolution of the minimum degree ordering algorithm,” *Siam review*, pp. 1–19, 1989.

[116] P. Amestoy, T. Davis, I. Duff, *et al.*, “An approximate minimum degree ordering algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[117] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[118] G. Karypis, *METIS 5.0: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, August 2011.

[119] I. S. Duff and G. A. Meurant, “The effect of ordering on preconditioned conjugate gradients,” *BIT*, vol. 29, no. 4, pp. 635–657, 1989.

[120] E. Chow and Y. Saad, “Experimental study of ilu preconditioners for indefinite matrices,” *Journal of Computational and Applied Mathematics*, vol. 86, no. 2, pp. 387–414, 1997.

[121] M. Benzi, D. Szyld, A. Van Duin, *et al.*, “Orderings for incomplete factorization preconditioning of nonsymmetric problems,” *SIAM Journal on Scientific Computing*, vol. 20, no. 5, pp. 1652–1670, 1999.

[122] M. Benzi, W. Joubert, and G. Mateescu, “Numerical experiments with parallel orderings for ILU preconditioners,” *Electronic Transactions on Numerical Analysis*, vol. 8, pp. 88–114, 1999.

[123] H. Elman and E. Agron, “Ordering techniques for the preconditioned conjugate gradient method on parallel computers,” *Computer Physics Communications*, vol. 53, no. 1-3, pp. 253–269, 1989.

[124] S. Doi and A. Lichnewsky, “A graph-theory approach for analyzing the effects of ordering on ILU preconditioning,” *Rapports de recherche- INRIA*, 1991.

[125] E. D’Azevedo, P. Forsyth, and W. Tang, “Ordering methods for preconditioned conjugate gradient methods applied to unstructured grid problems,” *SIAM Journal on Matrix Analysis and Applications*, vol. 13, p. 944, 1992.

[126] S. Stotland and J. Ortega, “Orderings for parallel conjugate gradient preconditioners,” *SIAM Journal on Scientific Computing*, vol. 18, p. 854, 1997.

[127] S. Doi and T. Washio, "Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations," *Parallel Computing*, vol. 25, no. 13-14, pp. 1995–2014, 1999.

[128] T. Iwashita and M. Shimasaki, "Block red-black ordering method for parallel processing of ICCG solver," in *High Performance Computing*, pp. 297–300, Springer, 2006.

[129] Y. Saad, "ILUT: A dual threshold incomplete LU factorization," *Numer. Linear Algebra Appl*, vol. 1, no. 4, pp. 387–402, 1994.

[130] R. Pozo, K. Remington, and A. Lumsdaine, "Sparselib++ v. 1.5, sparse matrix class library, reference guide," *SparseLib++ Homepage: http://gams. nist. gov/acmd/Staff/RPozo/sparselib++. html*, University of Notre Dame, 1996.

[131] D. Hysom and A. Pothen, "Level-based incomplete LU factorization: Graph model and algorithms," *Preprint UCRL-JC-150789, US Department of Energy*, p. 17, 2002.

[132] H. A. Van Der Vorst, "High performance preconditioning," *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 6, pp. 1174–1185, 1989.

[133] M. Pakzad, J. L. Lloyd, and C. Phillips, "Independent columns: A new parallel ILU preconditioner for the PCG method," *Parallel Computing*, vol. 23, no. 6, pp. 637–647, 1997.

[134] A. Khan and O. Mohammed, "Parameter optimization for sensorless position and speed control of permanent magnet motor at low speed using genetic algorithm," in *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*, pp. 1–5, IEEE, 2009.

[135] A. Sarikhani and O. Mohammed, "Multiobjective design optimization of coupled PM synchronous motor-drive using physics-based modeling approach," *Magnetics, IEEE Transactions on*, vol. 47, no. 5, pp. 1266–1269, 2011.

[136] K. Rashid, J. Ramírez, and E. Freeman, "Optimization of electromagnetic devices using sensitivity information from clustered neuro-fuzzy models," *Magnetics, IEEE Transactions on*, vol. 37, no. 5, pp. 3575–3578, 2001.

[137] G. Hawe and J. Sykulski, "A hybrid one-then-two stage algorithm for computationally expensive electromagnetic design optimization," *COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, vol. 26, no. 2, pp. 236–246, 2007.

[138] A. Forrester, A. KEANE, and A. SOBESTER, "Engineering design via surrogate modelling: a practical approach," *Recherche*, vol. 67, p. 02, 2008.

[139] A. Forrester and A. Keane, "Recent advances in surrogate-based optimization," *Progress in Aerospace Sciences*, vol. 45, no. 1-3, pp. 50–79, 2009.

[140] S. Xiao, M. Rotaru, and J. Sykulski, "Exploration versus exploitation using kriging surrogate modelling in electromagnetic design," 2011.

[141] J. Sykulski, "Reducing computational effort in field optimisation problems," *COMPEL: Int J for Computation and Maths. in Electrical and Electronic Eng.*, vol. 23, no. 1, pp. 159–172, 2004.

[142] J. Sykulski, "Computational electromagnetics for design optimisation: the state of the art and conjectures for the future," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 57, no. 2, pp. 123–131, 2009.