Czech Technical University in Prague

Faculty of Civil Engineering
Department of Mechanics

EFFICIENT PARALLEL COMPUTING ON HETEROGENEOUS SYSTEMS IN STRUCTURAL MECHANICS

Doctoral Thesis

MICHAL BOŠANSKÝ

Prague, September 2019

Ph.D. Programme: (P3604) Civil Engineering
Branch of study: (3607V009) Building and Structural Engineering

SUPERVISOR: PROF. DR. ING. BOŘEK PATZÁK

AFFIDAVIT

Name of Ph.D. student: Michal Bošanský

Title of Ph.D. thesis: Efficient Parallel Computing on Heterogeneous Systems in Structural Mechanics

I hereby declare that this thesis has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Prague September 2019 signature

ABSTRACT

Developments in computer hardware are currently bringing new opportunities for numerical modelling. The current trend in technology is parallel processing and making use of multiple processing units simultaneously to solve a given problem. Many engineering problems lead to extensive and time consuming computational problems. Solutions to these problems using parallel computing can significantly reduce computational time by using the available hardware more efficiently. Parallel techniques using modern computers with a distributed memory model also enable large and complex problems to be solved. This thesis evaluates different parallelization strategies in finite element software. The first part of the thesis examines assembly evaluation of different parallelization strategies in assembly operations for right-hand side vectors and left-hand side system matrices, which are one of the critical operations in any finite element software. The finite element method leads to a set of algebraic equations whose components are assembled from individual element contributions. Different strategies for assembling right-hand side vectors and left-hand side matrices using systems with shared memory models are proposed and evaluated. The principal issue in parallel assembly is to prevent the race conditions where the same memory location is updated by multiple threads. The aim of the next section of the thesis is to evaluate the performance of existing serial and parallel linear equation solvers in solving a large-scale, sparse, non-symmetric system of linear equations as a part of the solution in finite element software. In this section, the differences between a sequential and parallel solution using different equation solver types are studied. The parallel method uses different memory model types that are represented as shared or distributed memory models. The final section of the thesis deals with tuning a parallel load balancing framework as a part of the parallel solving process in finite element software. The parallel framework was based on a domain decomposition paradigm. The capabilities, efficiency, and performance of all implemented parallel methods are tested on problems in solid mechanics, and the obtained results are discussed. The results showed that the performance of implemented parallel algorithms is comparable to or better than serial computations previously designed by other researchers using algorithms with the non-optimally performing hardware available at the time.

ABSTRAKT

Současný vývoj v oblasti počítačového hardwaru přináší nové možnosti v numerickém modelování. Současným technologickým trendem v paralelním processingu, které je založeno na současném využívání paralelních více-procesorových jednotek k řešení daného problému. Mnoho inženýrských problémů vede k výpočetně velkým a časově náročným výpočetným problémům. Řešení těchto problémů s využitím paralelního počitaní může výrazně znížit čas řešení úlohy s efektivnějším využitím dostupného hadwaru. Práce se zabývá hodnocením různých paralelních strategií v konečně prvkovém softwaru. První část práce se zabýva hodnocením rešení a implementace různých paralelních strategií pro sestavovací operace pro vektory pravých stran a matic, které jsou jednou z kritických operací v jakým koliv konečně prvkovém programu. Metoda konešných prvků vede k souboru algebraických rovnic, které jsou složeny z jednotlivých elementárních příspěvků. Jsou navrženy a vyhodnoceny různé strategie pro sestavování vektorů pravých stran a matic a jsou založené na systémech s modelem sdílené paměti. Hlavním problémem v paralelním řešení je zabránit jevu "race-conditions", kde stejné místo paměti má být aktualizováno více podprocesy. Cílem následující části doktorské práce je zhodnotit výkonnost existujících sériových a parallelních rešiců soustav lineárních rovnic, jako další část řešení v konečně prvkovém programu. V této části studujeme rozdíly mezi sériovým a paralelním řešením s využitím různých typů řešení rovnic. Paralelní přístup využívající různé typy paměťových modelů, které jsou reprezentovány jako sdílený nebo distribuovaný paměťový model. Poslední část práce se zabývá laděním paralelního frameworku pro vyvažování zátěže jako součást procesu paralelního řešení v softwaru konečných prvků. Paralelní rámec je založen na paradigmatu rozkladu domén. Schopnost, efektivita a výkonnost všech implementovaných paralelních přístupů je testována na problémech z mechaniky tuhých teles a získané výsledky jsou diskutovány v dizertační práci. Ukázalo se, že výkon implementovaných paralelních algoritmů je srovnatelný nebo lepší než sériové výpočty, které dříve navrhli jiní výzkumníci s využitím algoritmů se zřetelem na optimálním výkonem dostupného hardwaru. Paralelní techniky využívající moderní počítače s distribuovaným paměťovým modelem navíc umožňují řešení velkých a složitých problémů.

PUBLICATIONS

Publications in journals with impact factor

- M. Bosansky, B. Patzak: Parallelization of assembly operation in Finite Element Mmethod. Acta Polytechnica, Journal of Advanced Engineering. CTU in Prague, Under review.

Publication in peer-reviewed journals and conference papers

- M. Bosansky, B. Patzak On tuning of finite element load balancing framework. In: Engineering Mechanics2019- Book of full text. 25nd International Conference on Engineering Mechanics, Svratka (2019). ISBN: 978-80-87012-71-0, pp.61–64. ISSN: 1805-8248. doi: 10.21495/71-0-61. url: https://doi.org/10.21495%2F71-0-61.
- M. Bosansky, B. Patzak Performance Evaluation Of Different Linear Equation Solvers For Solving Nonlinear FE Problems On Multicore Architectures. In: Acta Polytechnica CTU Proceedings15(Dec.2018),pp.6–11. ISSN:1805-8248. doi: 10.14311/app.2018.15.0006. url: https://doi.org/10.14311%2Fapp.2018.15.0006..
- M. Bosansky, B. Patzak Parallel Approach To Solve Of The Direct Solution Of Large Sparse Systems Of Linear Equations. In: Acta Polytechnica CTU Proceedings13(Nov.2017), pp.50–53.ISSN: 978-80-01-06346-0. doi: 10.14311/app.2017.13.0016. url: https://doi.org/10.14311%2Fapp.2017.13.0016.
- M. Bosansky, B. Patzak On Parallelization Of Linear System Equation Solver In Finite Element Method. In: Proceeding of Engineering Mechanics 2017. Engineering Mechanics2017, Svratka,2017. Brno: Brno University of Technology(2017), pp.198–201.ISSN: 1805-8248.
- M. Bosansky, B. Patzak Evaluation of Different Approaches to Solution of the Direct of Large, Sparse Systems of Linear Equations. In: Advanced Materials Research 1144(Mar.2017), pp.97–101. ISSN: 1022-6680. ISBN: 978-3-0357-1092-2. doi: 10.4028/www.scientific.net/amr.1144.97. url: https://doi.org/10.4028%2Fwww.scientific.net%2Famr.1144.97.[1.
- M. Bosansky, B. Patzak On Parallelization Of Assembly Operations In Finite Element Method. In Engineering Mechanics 2016 - Book of full text. 22nd International Conference on Engineering Mechanics. Svratka. 2016. Praha: Institute of Thermodynamics, AS CR, v.v.i. 2016. p. 198-201, ISSN: 1805-8248, ISBN: 978-80-97012-59-8.

- M. Bosansky, B. Patzak Different Approaches to Parallelization of Vector Assembly. In: Applied Mechanics and Materials 821 (Jan.2016), pp.341–348. ISSN: 1662-7482. doi: 10.4028/www.scientific.net/amm.821.341. url: https://doi.org/10.4028%2Fwww.scientific.net%2Famm.821.341.
- M. Bosansky, B. Patzak Different Approaches to Parallelization of Sparse Matrix Assembly Operation. In: Applied Mechanics and Materials 825 (Feb.2016), pp.91–98. ISSN: 1660-9336. ISBN: 978-3-03835-603-5. doi: 10.4028/www.scientific.net/amm.825.91. url: https://doi.org/10.4028%2Fwww.scientific.net%2Famm.825.91
- M. Bosansky, B. Patzak On Parallelization Of Stiffness Matrix Assembly. In 20 th International Conference on Engineering Mechanics 2014. Svratka. 2014. Brno: Brno University of Technology. p. 100-103. ISBN: 978-80-214-4871-1.
- M. Bosansky, B. Patzak Using OpenMP in OOFEM. In Proceedings of 4th Conference Nano & Macro Mechanics. 4th Conference Nano & Macro Mechanics 2013. Prague. 2013. Praha: Czech Technical University in Prague. p. 23-26. ISBN: 978-80-01-05332-4.

"The stars are matter, We're matter, But it doesn't matter."

— Cpt. Beefheart

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Dr. Ing. Bořek Patzák for the opportunity to work in the field of structural engineering, for the continuous support of my Ph.D study and related research, for his patience, motivation and also immense knowledge. His encouraging guidance helped me in all time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

I acknowledge support by the SGS grant. My sincere thanks also go to Czech Technical University in Prague, Faculty of Civil Engineering, Department of mechanics, which gave me access to the office and research materials. Without this precious support, it would not be possible to conduct this research.

My thanks also belong to my family for their help, love and support. Also I thank my friends. In particular, I am grateful to Andrea Šolstésová, Bernard Halás, Martin Šoltés, Dušan Sichrovský, Kamil Barbierik and Jozef Zakucia for their insightful comments and helpful comments.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 MOTIVATION

Computational advancements driven by developments and improvements in the power and pervasiveness of computers and communications have led to a new trend in computational science and engineering. Capitalizing on those advances by developing techniques for modern hardware has enabled large and complex problems to be solved. Traditional code runs sequentially and is limited by the available resources of a single computing unit or single machine. Serial computers only let us run simulation code sequentially and often have limited resources, represented by the single processing unit and available system memory. Only one instruction may be processed at a time, and only when this instruction is complete the next can be executed. The notion of parallel computing suggests dividing a large problem into smaller tasks that are solved simultaneously using available computing resources. Parallelization can significantly reduce the time required to solve a problem by using modern hardware more efficiently. Parallel programs are more difficult to design, because parallelism introduces new sources of complexity. Communication and synchronization between tasks are one of the most important problems in obtaining effective parallel performance. Several parallel programming models are available for parallel programs. An overview of existing platforms and programming models will be given in this work. The work examines the development of strategies and algorithms for optimal solutions to numerical problems in structural engineering based on finite element software. Other contributions focus on tuning the load balancing process, which is based on a domain decomposition paradigm. Load balancing strategies will be designed for problems with evolving complexity and heterogeneous, non-dedicated environments.

1.1 PROBLEM STATEMENT

Finite Element Method (FEM) is one of the most popular method for solving numerous problems in engineering. It actually consists of broad spectrum of methods for finding an approximate solution to boundary value problems in partial differential equations. Numerical solutions for many engineering problems lead to the solution of nonlinear models consisting of very complex geometries and many degrees of freedom, which are large and time consuming computational problems. Large scale problems also exist that cannot be solved on a sequential machine because of the lack of accessible hardware on a serial computer. In situations where large scale problems cannot be solved on available hardware resources, parallel computing based on the decomposition of a computational task using more available parallel hardware resources allows such large scale problems to be solved. This thesis aims to propose and evaluate parallel algorithms for different parts of the overall Finite Element Method solution process.

Different parts of the FEM solution process were selected as the most interesting and time-consuming numerical operations that could be solved in parallel. The assembling processes of vectors and matrices, solution of a linear system of equations and optimization of load balancing framework were selected as processes that could be optimized and parallelized. The different techniques of parallel assembly of vectors and matrices in the FEM were proposed, implemented and evaluated. The interface to direct parallel linear solver is developed and performance evaluated. New load balancing strategy has been proposed and its performance compared to the existing one.

The proposed parallelization methods were validated against serial versions and their efficiency was evaluated by comparing the performance of the parallel algorithm against serial version or previous parallel implementation. The performance has been evaluated on various benchmark problems from structural mechanics.

1.2 OVERVIEW OF THE THESIS

The thesis has eight chapters. The first chapter named *"Motivation"* is describing the motivation for the presented thesis. In the second chapter, the key contributions of the thesis are summarized. The next chapter named *"State of the art* introduces parallel computing, gives the overview of the design of available parallel computers, introduces the scalability of parallel algorithms. Moore's law and new trends in high-performance computing are also introduced and described. The currently available parallel platforms and libraries used in this work are discussed.

Following chapter entitled *"Introduction to the Finite element method* presents a brief review of Finite element method. A sparse matrix and vector assembly operations on serial computers are presented. The methods for solving systems of linear equations are discussed in this chapter as well. Also, the solution of large scale engineering problems using parallel computation based on a parallel load-balancing framework is discussed.

The chapter *"Parallelization of left and right hand side assembly"* presents design and implementation parallel assembly algorithms in finite element software. In this chapter, various strategies are presented and their performance is evaluated.

The parallel algorithms solving systems of large sparse linear equations are discussed in the next chapter. The chapter focuses on comparing the efficiency of different existing libraries for the solutions of large, sparse, non-symmetric systems of linear equations, based on direct or iterative algorithms.

The improved algorithm for determining the processor weights for dynamic load balancing is proposed and evaluated in the next chapter. The performance of static and dynamic load balancing approaches are assessed, and the results are presented and discussed.

Finally, the last chapter contains final conclusions of the thesis.

2 KEY THESIS CONTRIBUTIONS

Several novel ideas and implementations are proposed in the thesis. The first evaluates different parallelization strategies of assembly operations for right-hand side vectors and left-hand system matrices, which are one of the critical operations in any finite element software. Parallelization strategies based on a shared memory model using different shared memory libraries are presented. The assembly process itself is based on localization of individual element contributions into the global stiffness matrix and global force vector according to their connectivity. The principal issue in parallel assembly is to prevent race conditions, when the same memory location is updated by multiple threads. The considered strategies were based on simple synchronization directives, various block locking algorithms, and finally on smart locking free processing based on a colouring algorithm. These parallel methods were implemented in finite element code and compared to a previously implemented sequential method. Some of the results have already been published in papers [5], [6], [7], [8], and [9].

Optimization of the solution process focuses on solving the system of linear equations as a chronological process after the stiffness matrix and force vector assembly. This solution process involves a parallel approach for numerically solving large, sparse, non-symmetric systems of linear equations that can be a part of any finite element software. In general the FEM may results in linear or nonlinear system of equations. In nonlinear mechanics, the problems cannot be solved directly, and an iterative solution algorithm must be used, typically based on different variants of Newton's method. The problems are solved in a series of controlled steps in which the equilibrium state is obtained iteratively. In order to obtain a scalable algorithm, all the steps have to be parallelized, including solving a linear system of equations stage, which can be time-consuming. In this contribution, the differences between the sequential and parallel solutions and the differences between the direct parallel and iterative parallel solvers are presented. The existing libraries of direct linear equation solvers for solving large systems of linear equations in linear or nonlinear problems were connected and compared to existing implementations of iterative linear equation solvers in finite element software (some of the results have already been published in papers [10], [11], [12] and [13].

Finally, the tuning up methods for parallel static and dynamic load balancing frameworks of finite element software were implemented. These methods managed the parallel load balancing framework as a part of the solving process in the finite element method. The idea of a parallel framework is based on a domain decomposition paradigm. The partitioning based on the load balancing process can be affected by many factors. Primarily, the solution process on each sub-domain should be balanced. The upgrade method was based on new method of determining a set of processor weights parameters leading to better load rebalancing. The innovation consists in determining the processor weight automatically using specifically designed benchmarks. This allowed to obtain more appropriate domain decomposition matching actual performance of individual nodes. The capabilities and performance of these methods based on the improved load balancing framework depend on hardware performance and were compared to existing load balance methods. The results are demonstrated in the published paper [14].

3 STATE OF THE ART

The engineering community is facing growing demands for state of the art modelling and realistic predictions in many fields. These demands bring new challenges for computational science as well. The idea of parallel computing is mainly about dividing the large problem into smaller tasks, which can be solved simultaneously using available computing resources. In cases, where large scale problems could not be solved on available hardware resources, parallel computing based on the decomposition of computations task to more available parallel hardware resources can enable the solution of this large scale problems. The parallel algorithms bring new requirements for designing the program, which is for example represented by creating and joining computational threads including the communication and synchronization between computational threads, in comparison with the design process of a sequential program. Parallel programs can be classified, for example, by type of memory architecture.

3.1 MOORE'S LAW

Moore's Law is a computing term which predicts computer development over the decade and it originated around 1970. The simplified version of this law states that processor speed or overall processing power for computers will be doubled every two years. In the figure 3.1 is presented microprocessor clock speed measures the number of pulses per seconds illustrated the frequency of the processor. A quick check among technicians in different computer companies shows that the term is not very popular but the rule is still accepted. The law prediction is based on the number of transistors on an affordable CPU would double every two years, which is essentially the same trend that was mentioned above. The processor speeds from 1970 to 2020 show that the law has reached its limit or is nearing the limit. The processor speeds ranged from 740 kHz to 8 MHz in the 1970s. In the time from 2000 to 2009 there has not really been much of a speed difference. The range is from 1.3 GHz to 2.8 GHz, which present the fact, the speeds have just doubled within the 10-year span. On the other hand, the number of transistors its more accurate to apply the law to transistors than to speed. In 2000 the number of transistors in the CPU numbered 37.5 million, while in 2009 the number went up to 904 million transistors, see Figure 3.2. In the Figure 3.3 is illustrated the transistor count per square millimetre as a different view to the quantity of transistor in the processor.

The limitation of processor speed is based on fact, that the given number of transistors must fit into the processor. The trend illustrated on Figure 3.3

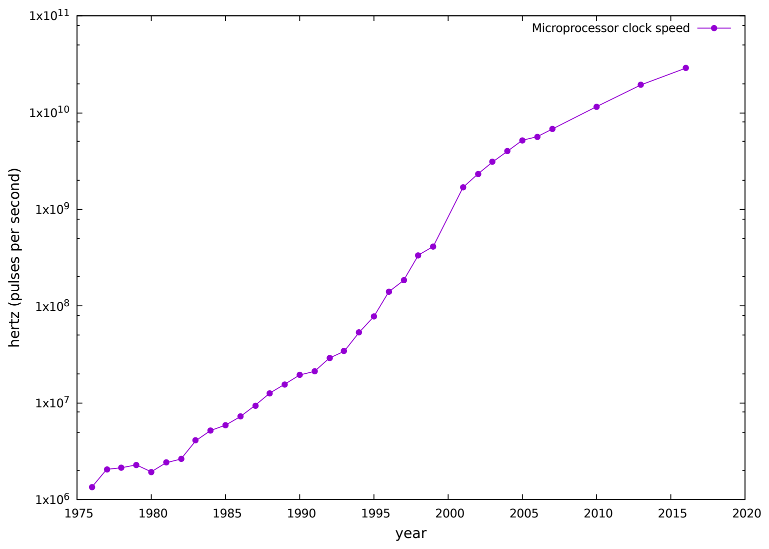


Figure 3.1: Microprocessor clock speed measured in hertz (pulses per second) by the year 1976 - 2016

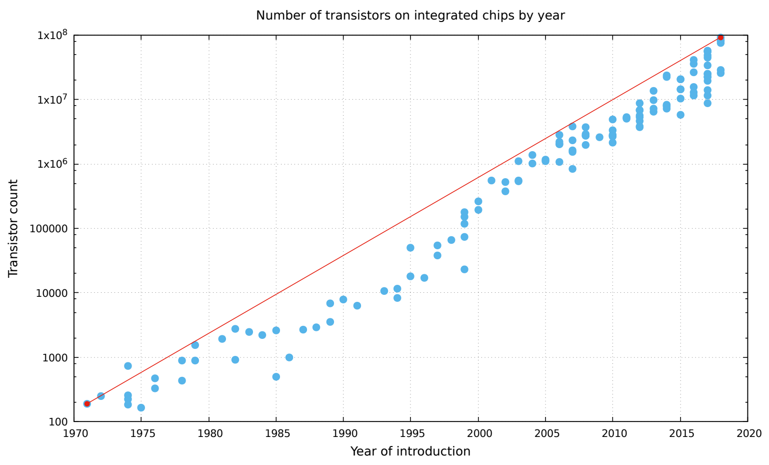


Figure 3.2: Transistor count per processor by the year 1971 - 2018

show the transistor density has been increasing at a fairly even exponential rate, even into year 2018.

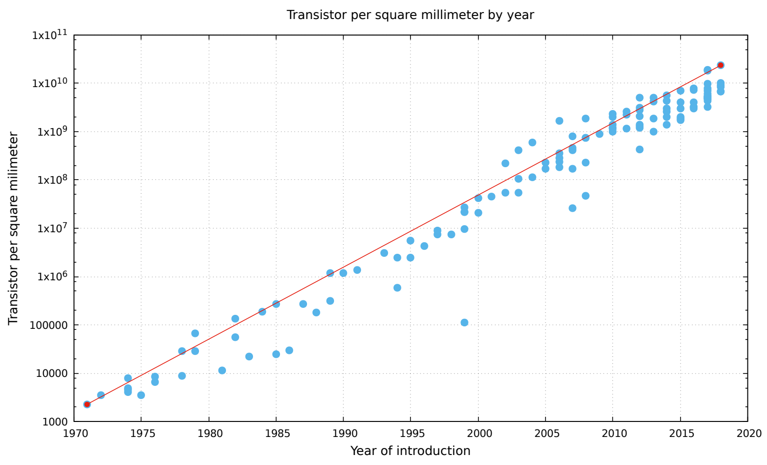


Figure 3.3: Transistor count per square millimeter by the year 1971 - 2018

The limitation which exists is that once transistors will be created with the size of atomic particles, then there will be no more room for growth in the CPU manufacture where speed is concerned.

3.2 high - performance computing trends

In this section, the trends in supercomputing are discussed. The powerful parallel vector computers can deliver a high computational performance only if the application software is adapted to computers architectures. The supercomputer is a designation given about 300 computers installed worldwide with the peak computing power of over 100 megaflops, which represents 100 million of floating point operations per second. These computers have mainly been used for numerical experimentation in various scientific domains such as structural mechanics, fluid mechanics, seismic explorations, quantum mechanics, materials science. The supercomputers are specific with high peak computing power achieved by very rapid clock periods, parallel processors, pipeline architectures leading to two operations (1 added + 1 multiple) per clock period, large system memories with fast and strong network connections to the outside network. In addition, to benefit most from the high computing power, it is necessary to formulate an application in such a way that the maximum of all the computations is executed in parallel.

The scientific and engineering trend is about solving numerically the most realistic physical model available, described for example, by a set of partial differential equations. The numerical solution of (nonlinear) system of PDEs is based on various discretization methods (finite elements, finite volumes, finite differences, etc). To reduce the number of time steps, implicit methods should be used. The very efficient iterative solvers based on the methods of multigrid or conjugate-gradient with preconditioning may be used. To make the best use of the parallel architectures of future supercomputers, algorithms and programming techniques leading to the definition of codes with coarse parallel granularity should be adopted.

For example, the *TOP500* project provides a reliable basis for tracking and detecting trends in high-performance computing [17]. This project started in 1993 and twice a year, a list of the sites operating the 500 most powerful computer systems is assembled and released. The performance measure is based on *Linpack* benchmark for ranking the computer systems. The *Limpack* benchmark was introduced by *Jack Dongarra* and the benchmark used in *Linpack* is to solve a dense system of linear equations. For the *TOP500*, the version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for given supercomputer is used. This performance does not reflect the overall performance of a given system. It reflects the performance of a dedicated system for solving a dense system of linear equations. The actual performance measuring is based on the different problem sizes n, and the user can get not only maximal achieved performance $Rmax$ for the problem size $Nmax$ but also the problem size $N1/2$ where half of the performance $Rmax$ is achieved. These numbers together with the theoretical peak performance $Rpeak$ are the number given in *TOP500*. The *TOP500* list contains a variety of information including the system specifications and its major application areas.

The improving computational power of supercomputers brings new opportunities in computational science.

3.3 PARALLEL COMPUTING

Parallel computing is introduced in this section. The solutions to many engineering problems are extremely demanding, both in terms of time and computational resources. Traditional serial computers only permit simulation codes to be run sequentially on a single processing unit, where only one instruction can be processed at any moment in time. The solution process is translated into a sequence of instructions, which are solved by a sequential processing unit. By contrast, parallel computing is based on the notion of partitioning the work into sets of smaller tasks that can be solved concurrently using multiple computing resources. Parallel processing allows tasks on a single computer to be performed concurrently with multiple processing units or even multiple computers with multiple processing units. Parallelization can significantly reduce computational time by using available hardware more efficiently. It can also enable the solution of large problems that often cannot be processed by a single machine.

3.3.1 *Computer architectures*

Existing computer architectures can be classified using Flynn's Taxonomy [18]. Flynn introduced the concept of instruction and data streams in order to categorize computers. The instruction cycle consists of a sequence of steps needed to execute an instruction in a program. Instructions in a program consist of two parts: the Opcode and Operand. The Operand has two parts: an addressing mode and the operand. The addressing mode specifies the method for determining the addresses of the data on which the operation is to be performed, and the operand is used as an argument by the method determining the address. The control unit of the computer's CPU fetches instructions from a program one at a time. Each fetched instruction is then decoded by the decoder, which is a part of the control unit, and the processor executes the decoder's instructions. The results of execution are temporarily stored in a Memory Buffer Register, also called the Memory Data Register. The normal execution steps are shown in Figure 3.4.

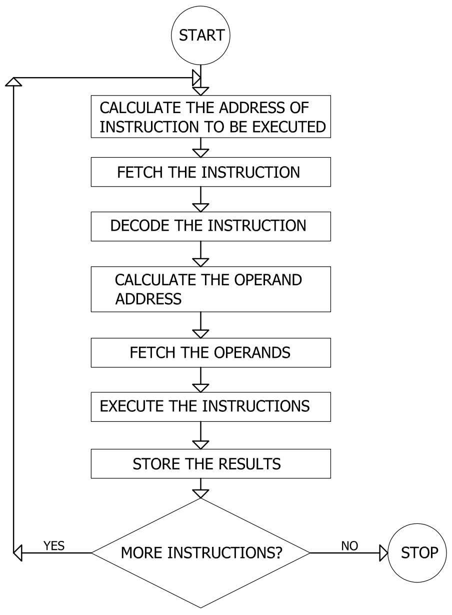


Figure 3.4: Steps for executing an instruction.

Stream refers to the flow of either instructions or data operated on by the computer. The instruction stream is a flow of instructions from the main memory to the CPU. Similarly, there is a bi-directional flow of operands between the processor and memory. This flow of operands is called a data stream. This means that the sequence of instructions executed by the CPU forms the instruction streams and data sequence (Operands) required for executing instructions for the data streams. Flynn's classification is based on the multiplicity of instruction and data streams observed by the CPU during program execution. The minimum number of streams flowing at any point during execution are I (instruction streams) and D (data streams).

Computers can be organized as follows:

- Single Instruction and Single Data stream (SISD)

SISD is an architecture found in a serial (non-parallel) computer with a single processing unit. One instruction stream is acted on by the CPU during any one cycle. Only one data stream is used as input during any one cycle (Fig. 3.5). This architecture represents the traditional type of serial computer.

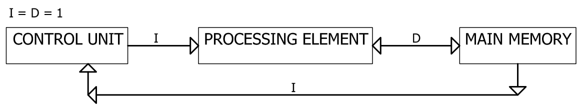


Figure 3.5: Single Instruction and Single Data Stream scheme.

- Single Instruction and Multiple Data stream (SIMD)

In this architecture, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data streams. This represents a type of parallel computing, where all processing units execute the same instruction at any given cycle (Fig. 3.6). Each processing unit operates on a different data element, which creates a multiple data stream.

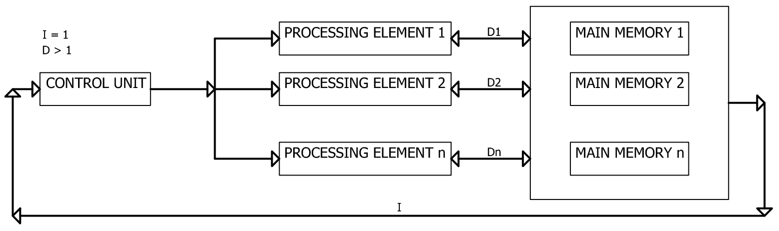


Figure 3.6: Single Instruction and Multiple Data Stream scheme.

- Multiple Instruction and Single Data stream (MISD)

In this architecture, multiple processing elements are organized by multiple control units. Each control unit handles one instruction stream and is processed through its corresponding processing element. Each processing element, however, only processes one data stream at a time (Fig. 3.7). All processing units operate on the same data.

- Multiple Instruction and Multiple Data stream (MIMD)

In this architecture, multiple processing elements and multiple control units are organized, as in MISD, its main difference though being multiple instruction streams operating on multiple data streams (Fig. 3.8). MIMD organization is the most popular choice for a parallel computer.

Parallel computers can also be classified, for example, by the type of memory architecture, such as shared, distributed and hybrid memory systems.

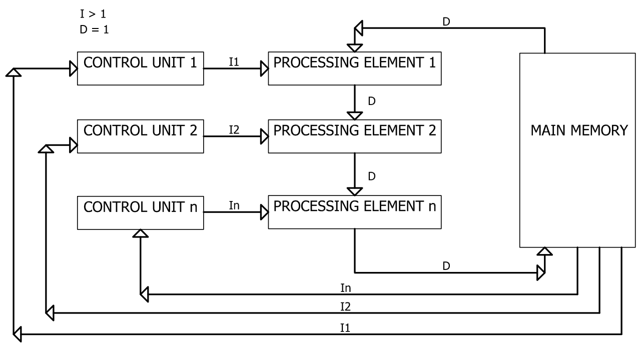


Figure 3.7: Multiple Instruction and Single Data Stream scheme.

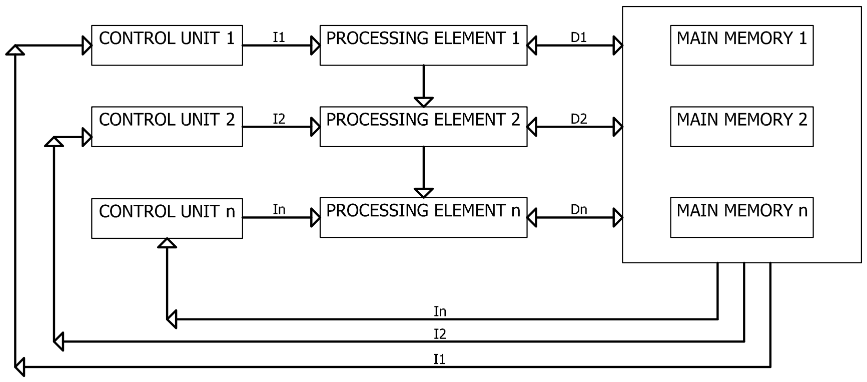


Figure 3.8: Multiple Instruction and Multiple Data Stream scheme.

- Shared memory parallel computers

In a shared memory system, the main memory and global address space are shared between all processing units, which can directly address and access the same logical memory. Global memory significantly eases the design of a parallel program, however, memory bus performance is a limiting factor in scalability as the number of processing units increases. In a shared memory system, multiple processing units can access a single address space that is shared between all processing units. Changes in the logical memory location affected by one processing unit are visible to all the other processing units. Shared memory systems are classified as Uniform Memory Access (UMA), as shown in Figure 3.9 and Non-Uniform Memory Access (NUMA), shown in Figure 3.10, according to memory access times. Uniform Memory Access (UMA) is found today in Symmetric Multiprocessor (SMP) machines. In UMA systems, the processing unit has equal access and access times to memory.

Non-Uniform Memory Access (NUMA) is achieved by physically linking two or more SMPs. One SMP can directly access the memory of another SMP. Memory access across a link between SMPs is slower. Its main advantage is a globally accessible memory, which provides a user-friendly perspective to programming in terms of memory. Its primary disadvantage is a lack of scalability between the memory and processing units. The link between processing units is not scalable as the number of processing units increases.

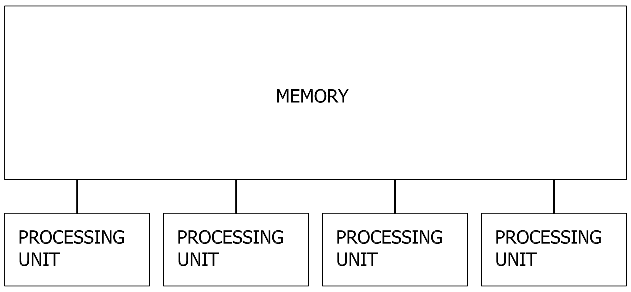


Figure 3.9: Uniform Memory Access.

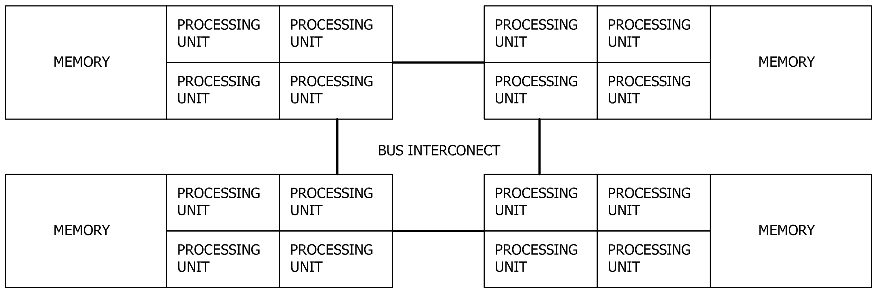


Figure 3.10: Non-Uniform Memory Access.

- Distributed memory parallel computers

In distributed memory systems, memory is physically distributed between individual processing units and has no global address space. When a processor needs to access data on another processor, it is generally the programmer's task to explicitly define how and when data is communicated. Processing units have their own local memory. Distributed memory systems require a communication network to connect the local memories of processing units (Fig. 3.11. Memory addresses in one processing unit do not map to another processing unit, so there is no concept of global address space across all the processing units. Because each processing unit has its own local memory, it operates independently, and the changes made to its memory have no effect on the memory of another processing unit. The cost of communication compared to local memory access can be very high, however, its advantage is that the overall memory is scalable as the number of processors increases. Its disadvantage is a difficult mapping of existing data structures based on global memory into this memory configuration.

- Hybrid distributed-shared memory systems

Hybrid systems combine the features of shared and distributed memory systems, providing global, shared memory for a reasonably small number of processing units that are combined into a distributed memory system. This

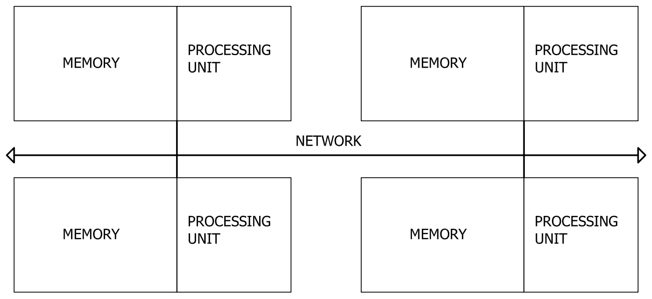


Figure 3.11: Distributed memory parallel computers.

type of system is the largest and fastest parallel computer known at this time. The shared memory component can be the processing unit (CPU) or graphics processing unit (GPU). In this system, communications are required to move data from one machine to another (Fig 3.12). An important advantage in hybrid systems is greater scalability. The disadvantage of the hybrid system is greater programming complexity.

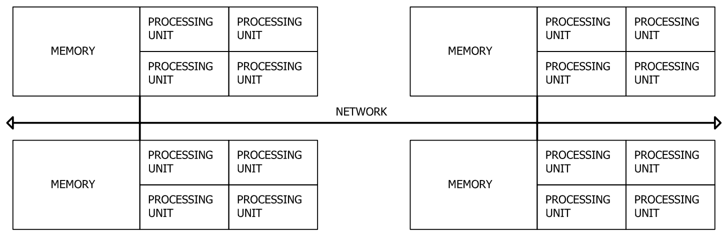


Figure 3.12: Hybrid distributed-shared memory parallel computers.

An equally important fact about memory architecture concerns an aspect of SMP systems. In SMP systems, each processor has a local cache. The processing unit (CPU) has several layers of cache memory between the CPU and main memory (RAM). The cache layers are designed as two or three layers, which are designated L1 cache, L2 cache and L3 cache. The L1 cache is small and very fast and situated right next to the core that uses it. The L2 cache is larger and slower and is also only used by a single core. The L3 cache is more common in modern multi-core machines and is larger and slower than the L2 cache and shared across all cores on a single socket. Finally, the CPU can also access the main memory (RAM), which is shared across all cores and all sockets. Individual items are not stored in the cache as single variables or single pointers. Cache memory stores data in cache lines, typically 64 bytes, and effectively references a location in the main memory (Fig. 3.13). A C++ double integer is 8 bytes, and in a single cache line, eight C++ double integer variables can be stored. A cache line model is a robust tool for iterating over any data structure that is allocated to contiguous blocks in memory. If items in the data structure are not placed next to each other in memory, then the advantage of free cache loading remains unused.

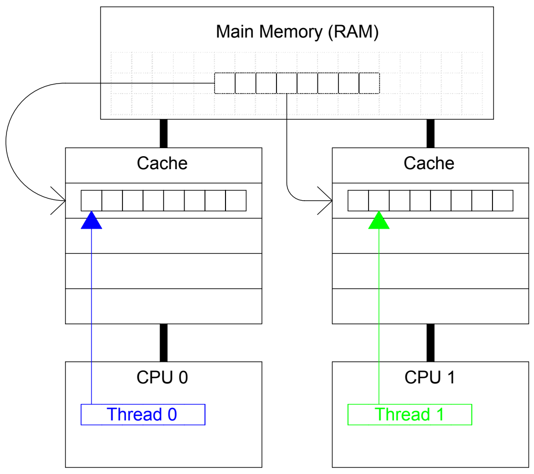


Figure 3.13: Cache layer schema—threads on different processors modify variables residing on the same cache line.

For multiprocessors, the replicated unit is a processor (shared memory systems), while in a multicomputer, the replicated unit is the whole computer (distributed memory systems). Multiprocessors are tightly coupled to a high degree of sharing resources through a high-speed backplane or motherboard. On the other hand, the multicomputer consists of multiple computers, often called nodes, interconnected by a message-passing network. It is loosely coupled since it only has a low degree of resource sharing through a commodity network. Multicomputers are more scalable than multiprocessors. A multicomputer's scalability is multitude scalability. SMPs as a typical example of multiprocessors systems are processor scalable systems, while multiprocessors scale in many components, including the processor, system memory and even I/O devices. In an SMP, the shared memory (and the memory bus) is a bottleneck, while in a multicomputer, there is no memory bottleneck. Multicomputers can provide a much higher aggregate memory bandwidth and reduced memory latency.

3.3.2 *Scalability*

The idea behind parallel algorithms is partitioning the problem into a set of smaller tasks that can be solved simultaneously. One of the important characteristics of the parallel algorithm is its computational scalability, which is the most important goal in parallel computing. The scalable parallel algorithm achieves a reduction in execution time by using more processing units, ideally in a linear trend. Ideal scalability is difficult to obtain because of the overhead costs of the parallel algorithm (synchronization and communication). Almost every parallel algorithm has an overhead cost compared to a sequential version. Individual tasks cannot be executed concurrently without synchronizing and communicating with other tasks. Some parts of the algorithm are also essentially serial and can only be executed by a single thread. In addition to speedup, parallel computing allows large and complex problems to be solved that could not be solved on a single, well-equipped machine.

- Amdahl's law

Amdahl's law is named after Gene Amdahl. This formula is used in parallel computing to predict the theoretical maximum speedup time using multiple processing units. Amdahl's law is a model for the relationship between the excepted speedup of a parallel algorithm relative to the serial algorithm. The law describes the excepted speedup of the parallel code over the serial code when using *n* processing units is dictated by the proportion of the process that can be made parallel, *p*, and the portion of that cannot be parallelized, *(1-p)*. This relationship is shown in figure 3.14.

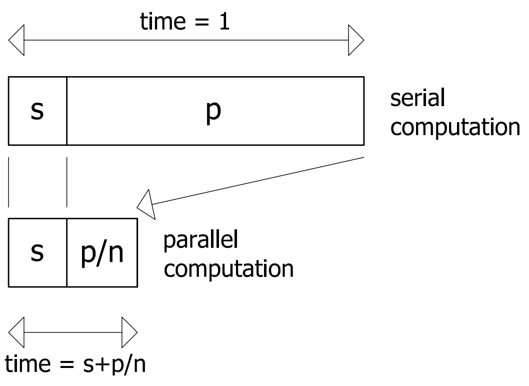


Figure 3.14: Amdahl's law.

- Gustafson's law

Gustafson's law addresses the shortcomings of Amdahl's law, which does not fully exploit the computing power that becomes available as the number of machines increases. Gustafson's Law instead proposes that programmers tend to set the size of problems in order to use the available equipment and solve problems within a practical fixed time. Therefore, if faster (more parallel) equipment is available, larger problems can be solved at the same time.

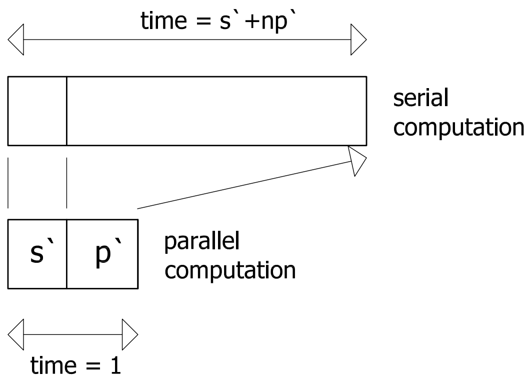


Figure 3.15: Gustafson's law.

3.4 AVAILABLE PARALLEL PLATFORMS AND LIBRARIES

Parallel programming frameworks provide basic infrastructure for developing and executing parallel algorithms. Typically, they are tightly connected to particular parallel computer architecture. One can distinguish between shared memory, distributed memory and hybrid systems.

3.4.1 *Shared memory libraries*

In this section, the general description of parallel libraries which are based on shared memory model with using different frameworks represented by *Open Multi-Processing* (OpenMP), *Portable Operating System Interface (POSIX) Threads, and* C++ 11 Threads programming interface, is presented.

OpenMP is a shared memory programming model that supports multi-platform shared memory multiprocessing programming in C, C++ language and Fortran, on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour.

The POSIX Threads (Pthreads) libraries are standardized thread programming interfaces for C/C++ language. Pthreads allows one to spawn a new concurrent process. Pthreads consists of a set of C/C++ language types and procedure calls. The original POSIX Threads application programming interface (API) synchronization subroutines allow protecting parallel program when multiple threads solve the problem.

Specifically for C++ programming language, the introducing the support for concurrent programming in C++ application that requires concurrent programming. Before C++11 standard the multi-threaded programming was based on platform-specific extensions like OpenMP, POSIX Threads, etc. The C++11 Thread libraries include utilities for creating, managing threads which are standardized for C/C++ language. A thread in C++11 Thread libraries share an address space with other threads which can lead to a basic problem with thread data races, two threads solving a single address space independently and that cause undefined results. The C++11 standard library contains classes for thread manipulation and synchronization, commonly protected data, and low-level atomic operations.

3.4.2 *Distributed memory libraries*

Despite the variety of available platforms, there is a common programming model based on message passing paradigm which is available on most parallel hardware configurations. The most commonly used method of programming distributed-memory systems is message passing or some variant of message passing. In basic message passing, the processes coordinate their activities by explicitly sending and receiving messages. For example, at its most basic, the *Message-Passing Interface* (MPI) provides functions for sending or receiving a message. MPI libraries are highly portable and available on virtually all parallel computing platforms, from shared memory systems to distributed memory systems. The parallel MPI programs are independent of machine architecture and type of network employed to transfer data from one processor to another. The current version of MPI assumes that processes are statically allocated, the number of processes is set at the beginning of program execution, and no additional processes are created during execution.

3.4.3 *Solvers for linear system of equations*

A system of linear equations is called sparse if an only a relatively small number of its matrix elements are non-zero. An efficient algorithm for solving a linear system must exploit this property. There are different schemes for efficient storage of sparse matrices and the solution of the related linear problem. The problem is about solved equation $K * r = F$ where r and F are just vectors, K is a sparse matrix, and even storing a dense matrix of K's size would be prohibitively expensive. There are a number of different libraries out there that solve a sparse linear system of equations, however, in this thesis the performance of only selected solvers is evaluated. This includes serial direct, skyline based solver, serial iterative solver based on IML library, parallel direct solver from SuperLU library and parallel, iterative, Krylov/based solver from PETSc library.

The SuperLU contains a set of sparse direct solvers for solving large sets of linear equations. In our case the matrix K is a square, non-singular $n\ x\ n$ sparse matrix and r and F are dense matrices (vectors). The kernel algorithm in SupeLU is sparse Gaussian elimination. In addition, to complete factorization, SuperLU platform also has limited support for incomplete factorization (ILU) preconditioner which approximately solves $K * r = F$.

The *Portable, Extensible Toolkit for Scientific Computation* (PETSc) has successfully demonstrated that the use of modern programming paradigms can ease the development of large-scale scientific application codes in C++ [34]. The software has evolved into a powerful set of tools for the numerical solution of partial differential equations and related problems on high-performance computers. PETSc consists of a variety of libraries (similar to classes in C++). Each library manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on the objects. The objects and operations in PETSc are derived from our long experiences with scientific computation. Some of the PETSc modules deal with index sets, including permutations, for indexing into vectors, renumbering, vectors, matrices, managing interactions between mesh data structures and vectors and matrices, nonlinear solvers and time steppers for solving time-dependent (nonlinear) partial differential equations including support for differential algebraic equations. Each consists of an abstract interface and one or more implementations using particular data structures. Thus, PETSc provides clean and effective codes for the various phases of solving partial differential equations, with a uniform approach for each class of problems. This design enables easy comparison and use of different algorithms. Hence, PETSc provides a rich environment for modelling scientific applications as well as for rapid algorithm design and prototyping. The libraries enable easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility and separates the issues of parallelism from the choice of algorithms. The PETSc infrastructure creates a foundation for building large-scale applications.

3.4.4 *Mesh partitioning library*

The methods providing effective partitioning are discussed in this paragraph. The mesh partitioning is itself a complex problem. Graph partitioning is effective with using traditional [1], [27] finite element simulations, due, in part, to high-quality serial and parallel graph partitioners. The available serial partitioners include Chaco [24], Jostle [35], METIS [20], Party [33] and Scotch [32]. The parallel graph partitioners include ParMETIS [19] and PJostle [35]. In this thesis, the ParMETIS library is used. The ParMETIS library is an MPI-based parallel library that implements a variety of algorithms for partitioning graphs, meshes, and for computing fill-orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel computations and large scale numerical simulations. The library provides parallel load balancing and rebalancing algorithms for general unstructured graphs and meshes, based on the parallel multilevel k-way graph partitioning.

4 INTRODUCTION TO THE FINITE ELEMENT METHOD (FEM)

4.1 FEM EQUATIONS

The Finite Element Method (FEM) has become a widely used tool for solving problems described by partial differential equations and been widely adopted by engineering and scientific communities as a reliable numerical tool. In mathematics, the FEM is a numerical method for finding a solution to boundary value problems for an ordinary differential equation (ODE) or a partial differential equation (PDE). In the FEM, differential equations are converted into an algebraic system of equations by using variational methods aided by decomposition of the problem domain into sub-domains called elements and an appropriate choice of interpolation functions.

To illustrate, consider the one-dimensional elastic problem shown in Figure 4.1, where a one-dimensional bar occupying body $\Omega = (0, L)$ with boundary $\Gamma = 0, L$. The bar is subject to a distributed volume force $b_{(x)}$ with prescribed displacement at $\Gamma^u (x = 0)$. The bar is characterized by length L, elastic module E and its cross-sectional area A.

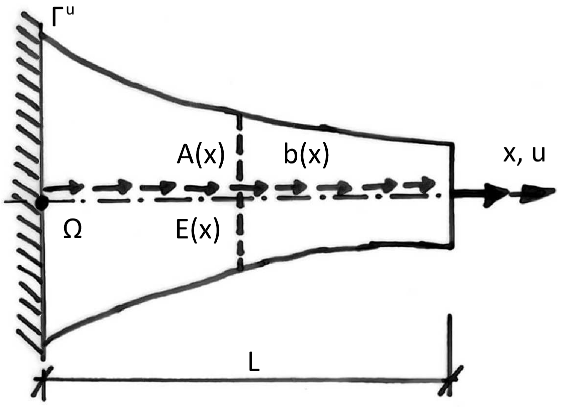


Figure 4.1: One-dimensional elastic problem member.

Displacement is denoted by $u(x)$ and deformation as $\varepsilon(x)$, where $(x \in \Omega)$.

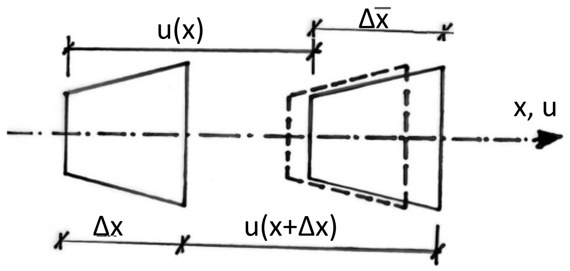


Figure 4.2: Deformation of bar.

From the kinematic considerations, the deformation is given as (see Figure 4.2):

$$\varepsilon(x) = \lim_{\Delta \to 0} \frac{\Delta\bar{x} - \Delta x}{\Delta x} = \lim_{\Delta \to 0} \frac{\Delta x + u(x+\Delta x) - u(x) - \Delta(x)}{\Delta(x)} = \frac{du(x)}{dx} \quad (4.1)$$

In the given problem, essential or kinematic boundary conditions are $u(x) = \bar{u}(x)$ for $(x \in \Gamma_u)$.

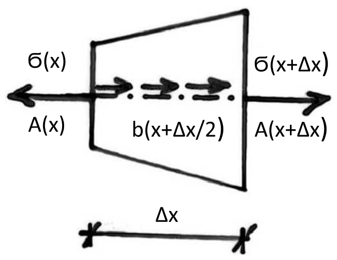


Figure 4.3: Equilibrium condition for infinitesimal element.

The equilibrium condition inside body Ω (for $x \in \Omega$) shown in Figure 4.3 is

$$\begin{aligned} &\rightarrow: -\sigma(x)A(x) + b(x + \frac{\Delta x}{2})\Delta x + \sigma(x+\Delta x)A(x+\Delta x) = 0 \\ &\lim_{dx \to 0} \frac{\sigma(x+\Delta x)A(x+\Delta x) - \sigma(x)A(x)}{\Delta x} + b(x + \frac{\Delta x}{2}) = 0 \\ &\frac{d}{dx}(\sigma(x)A(x)) + b(x) = 0 \end{aligned} \quad (4.2)$$

Hooke's law is assumed relating the stress and strain inside the body (for $x \in \Omega$) has the form $\sigma(x) = E(x)\varepsilon(x)$. After substituting are $\varepsilon(x)$ and $\sigma(x)$ from equations 4.1 and 4.2, we obtain the following differential equation.

$$\begin{aligned}&\varepsilon(x) = \frac{du(x)}{dx}(x) \rightarrow \sigma(x) = \\ &E(x)\frac{du(x)}{dx}(x) \rightarrow \frac{d}{dx}(E(x)\frac{du(x)}{dx}(x)A(x)) + b(x) = 0\end{aligned} \tag{4.3}$$

The boundary conditions for differential equation 4.3 are either kinematic boundary conditions ($u(x) = \bar{u}(x)$ for $(x \in \Gamma_u)$) or static boundary conditions ($\sigma n = \bar{t}$ for $(x \in \Gamma_t)$). The weak form can be obtained using the weighted residual method.

$$\int_\Omega \delta u(x)(\frac{d}{d(x)}(E(x)A(x)\frac{du}{dx}(x)) + b(x))\,\mathrm{d}x = 0 \tag{4.4}$$

where δu is test function satisfying homogeneous kinematic boundary conditions and $\delta u(x)$ is sufficiently integrable, where is $\delta u(x) = 0$ for $(x \in \Gamma^u)$ and its called weight function. Such displacement $u(x)$ is called a weak solution.

Next, we integrate by parts.

$$\int_\Gamma \delta u(s)EA(s)\frac{du}{dx}n(s)ds - \int_\Omega \frac{d\delta u}{dx}(x)EA(x)\frac{du}{dx}(x)dx + \int_\Omega \delta u(x)b(x)dx = 0 \tag{4.5}$$

The approach to solving this problem is based on finding $u(x)$ that is sufficiently integrable and satisfies $u(x) = \bar{u}(x)$ for $(x \in \Omega^u)$.

$$\begin{aligned}&\int_\Omega \frac{d\delta u}{dx}(x)EA(x)\frac{du}{dx}dx = \int_\Omega \delta u(x)b(x)dx \\ &\quad + \int_{\Gamma_u} \underbrace{\delta u(x)}_{=0} E(x)A(x)\frac{du}{dx}(x)n(x)dx \int_{\Gamma_t} \delta u(x)\, \underbrace{E(x)A(x)\frac{du}{dx}(x)n(x)}_{=\bar{t}}\, dx\end{aligned} \tag{4.6}$$

Sufficiently integrable with $u(x) = \delta u(x)$ satisfying for $\forall \delta u(x); \delta u(x) = 0 \in \Gamma_u$.

For a solution, we need to decompose structure Ω into n elements Ω_e. Local approximation of displacement is $u|_{\Omega_e(x)} \approx u^e(x) = N^e(x)r^e$, where r^e is the column matrix of nodal displacements, satisfying $u^e(x) = \bar{u}(x)$ for $(x \in \Gamma^u) \Leftrightarrow r^e_p = \bar{u}(x)$. Similarly, approximation of weight functions is $\delta u|_{\Omega_e(x)} \approx \delta u^e(x) = N^e(x)w^e$, where is $w^e_p = 0$ for $(x \in \Gamma_u)$. From equation 4.6, the left-hand side term can be evaluated.

$$\int_\Omega \frac{d\delta u}{dx}(x)EA(x)\frac{du}{dx}dx = \sum_{e=1}^{n} \int_{\Omega_e} \frac{d\delta u}{dx}(x)EA(x)\frac{du}{dx}dx \tag{4.7}$$

The derivatives can be expressed as

$$u^e(x) = N^e(x)r^e \Rightarrow \frac{du^e(x)}{dx} = \frac{dN^e(x)}{dx}r^e = B^e(x)r^e \tag{4.8}$$

$$\delta u^e(x) = N^e(x)w^e \Rightarrow \frac{d\delta u^e(x)}{dx} = \frac{dN^e(x)}{dx}w^e = B^e(x)w^e \tag{4.9}$$

The left-hand side of 4.6 can be written as

$$\begin{aligned}&\int_{\Omega_e} \frac{d\delta u^e}{dx}(x)EA(x)\frac{du^e}{dx}dx = \int_{\Omega_e}(B^e(x)w^e)^T EA(x)B^e(x)r^e dx\\ &= \int_{\Omega_e} w^{eT}B^{eT}(x)EA(x)B^e(x)r^e dx\\ &= w^{eT}\underbrace{\left(\int_{\Omega_e} B^{eT}(x)EA(x)B^e(x)dx\right)}_{K^e} r^e\end{aligned} \tag{4.10}$$

The right-hand side of 4.6 can be written as

$$\begin{aligned}&\int_{\Omega_e}\delta u(x)b(x)dx + \int_{\Gamma_t}\delta u(x)\bar{t}dx\\ &\approx \int_{\Omega_e}(N^e(x)w^e)^T b(x)dx + \int_{\Gamma_t}(N^e(x)w^e)^T\bar{t}dx =\\ &w^{eT}\left(\underbrace{\int_{\Omega_e}N^{eT}(x)b(x)dx}_{f^e_\Omega} + \underbrace{\int_{\Gamma_t}N^{eT}(x)\bar{t}dx}_{f^e_{\Gamma_t}}\right)\end{aligned} \tag{4.11}$$

By substituting equations 4.10 and 4.11 into equation 4.6, we finally obtain

$$\sum_{e=1}^{n} w^{eT}K^e r^e = \sum_{e=1}^{n} w^{eT}f^e_\Omega = \sum_{e=1}^{n} w^{eT}f^e \tag{4.12}$$

Alternatively, the same discrete system can be obtained by means of a deformation method. The solution consists in expressing nodal forces vector $F^e(x)$ as dependent on nodal displacements $u^e(x)$. The equilibrium conditions relating internal and nodal (end) forces read: conditions of equilibrium between internal forces and nodal forces $(F^e_1, F^e_2) : F^e_1 = -\sigma A, F^e_2 = \sigma A$; By substituting the material relation (Hooke law) and expressing the deformation as relative elongation of the bar we can end up with $\sigma = E\epsilon = E\frac{\Delta l}{l} = E\frac{u^e_2 - u^e_1}{l}$. The end forces (F^e_1, F^e_2) are $F^e_1 = -\sigma A = \frac{EA}{l}(u^e_1 u^e_2), F^e_1 = \sigma A = \frac{EA}{l}(u^e_2 u^e_1)$.

The normal form of equilibrium equations is:

$$\begin{bmatrix} F_1^e \\ F_2^e \end{bmatrix} = \begin{bmatrix} k & -k \\ -k & k \end{bmatrix} * \begin{bmatrix} u_1^e \\ u_2^e \end{bmatrix} k = \frac{EA}{l}$$

In matrix notation:

$$F^e = K^e * r^e$$

Provided $u_1^e = u_2^e$, there will be no internal forces and $F_1^e = F_2^e = 0$. The stiffness matrix is symmetric and singular. The global stiffness matrix and load vector is obtained by assembling the contributions from individual elements.

4.2 LEFT AND RIGHT HAND SIDE ASSEMBLY IN FEM

As demonstrated in previous section, the application of FEM leads to a discrete set of algebraic equations, corresponding to equilibrium equations in nodes. The global stiffness matrix and load vectors are obtained from a local element contributions by means of assembly process. The assembly process relies on the global numbering of discrete equilibrium equations and yields so-called code-numbers assigned to nodes (identifying equilibrium equations assembled in the given node) and elements (identifying equilibrium equation into which the element end-forces contribute). Typical serial implementation of sparse matrix assembly and the force vector involves loop over elements. Inside this loop, individual element matrices and vectors are evaluated and assembled into a global matrix and global vector. Mapping between an element's local degrees of freedom and corresponding global degrees of freedom is described with code numbers. The individual values in element matrices are added to the corresponding global matrix entry, whose row and column indices are determined using element code numbers. Assembling the global vector is similar to assembling the global matrix, the only difference being global vector entry, where only the row indices are determined using the element code numbers.

The typical serial algorithm for vector assembly is as follows in Algorithm 1

Algorithm 1 : Prototype code – Assembly of the element part of the load vector

```
001 for $elem = 1, nelem$
002     $F^e = computeElementVecto(elem)$
003     $Loc^e = giveElementCodeNubmer(elem)$
004     for $i = 1, Size(Loc^e)$
005         $F(Loc^e(i)) + = F^e(i)$
```

The typical serial algorithm for matrix assembly is similar to the algorithm for vector assembly (Algorithm 2).

The assembling process of discrete equilibrium equations as a parallel algorithm for matrix assembly is shown in Figure 4.4.

Algorithm 2 : Prototype code – Assembly of stiffness matrix

```
001 for $elem = 1, nelem$
002     $K^e = computeElementMatrix(elem)$
003     $Loc^e = giveElementCodeNubmer(elem)$
004     for $i = 1, Size(Loc^e)$
005         for $j = 1, Size(Loc^e)$
006             $K(Loc^e(i), Loc^e(j)) += K^e(i, j)$
```

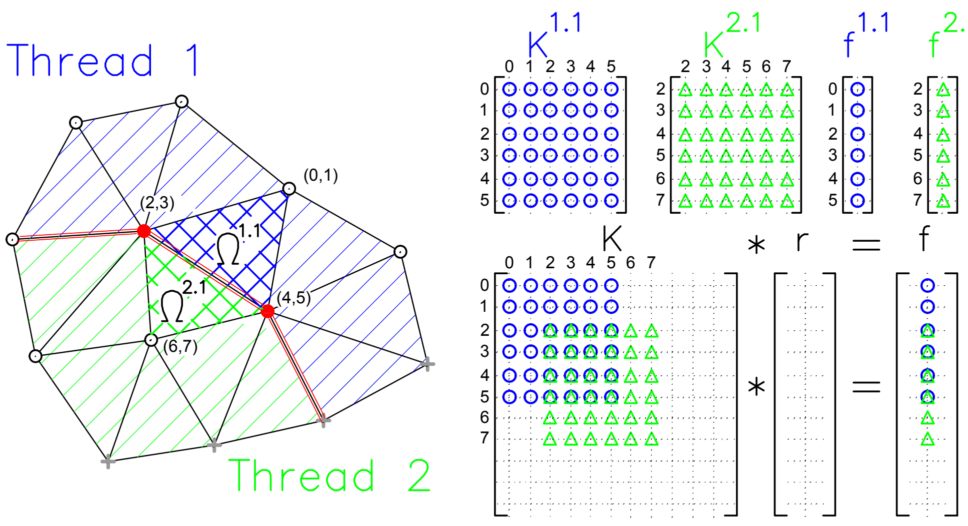


Figure 4.4: Parallel algorithm for Stiffness Matrix assembly.

As already mentioned, assembly operations are one of the typical steps in finite element analysis. In order to obtain a scalable algorithm, all the steps have to be parallelized, including the assembly stage, which can be costly. Especially when solving nonlinear problems, evaluating individual element contributions (tangent stiffness matrix, internal force vector) can be computationally demanding and must be performed for each load increment step and iteration (updating the stiffness matrix depends on a solution algorithm). The parallelization of assembly operations consists generally in splitting the assembly loop over elements into disjoint subsets, which are processed by individual processing threads. Within each thread and for each element in the corresponding subset, the contribution of individual elements is evaluated (this part can be evaluated concurrently), followed by an update of the global matrix/vector value. The key problem is that this step cannot be performed concurrently, as multiple threads may update the same global entry at the same time. It is therefore necessary to make sure the same global entry is not updated at the same time by multiple threads. This is known as a race condition. To prevent a race condition on the update, various techniques can be used. They typically involve using different locking primitives to make sure that either (i) the code performing the update can only be executed by a single thread, (ii) the specific memory location can only be updated by a single thread, or (iii) to order the evaluation of element contributions in such a manner that the conflict does not occur.

4.3 SOLUTION OF LARGE SPARSE SYSTEMS OF EQUATIONS IN THE FEM

The section introduces the numerical solution of large, sparse, non-symmetric systems of linear equations that can be a part of any finite element software. Also, the differences between linear and nonlinear model solutions are highlighted and an introduction to the solution process is given. In the FEM, differential equations are converted into an algebraic system of equations by using variational methods aided by the decomposition of the problem domain into sub-domains called elements and an appropriate choice of interpolation functions.

4.3.1 *Linear models*

In a case of linear problem, the discretization leads directly into the set of linear equations

$$K * r = F \tag{4.13}$$

One of the key features of the FEM is that the stiffness matrix is typically a positive definite symmetric matrix with a sparse structure. A system of linear equations is called sparse only if a relatively small number of stiffness matrix values are non-zero. An efficient algorithm for solving a linear system must exploit the symmetry and sparse structure by saving considerable memory and CPU resources. Several different storage schemes exist for sparse matrices. The problem concerns solving equation 4.13, where r and F are vectors, K is a sparse matrix, and storage of a dense matrix of $K's$ size would be prohibitively expensive. Several different libraries are available that solve a sparse linear system of equations.

4.3.2 *Nonlinear models*

Numerical solutions of many engineering problems lead to the solution of nonlinear models consisting of very complex geometries and many degrees of freedom. Nonlinear problems add complexity. In the nonlinear static case, the error depends on time discretization in history dependent problems, and a part of the error is accumulated in solutions based on an incremental, iterative approach. In structural mechanics, non-linearity can originate from nonlinear geometrical relationships (large deformations), non-linearity of constitutive relationships and from non-linear boundary condition (e.g. follower type of loading). Nonlinear problems are solved incrementally, typically using the Newton–Raphson algorithm. This makes the nonlinear problem solution more demanding than linear problems. A schema of common processes for iteration processes is shown in Figure 4.5.

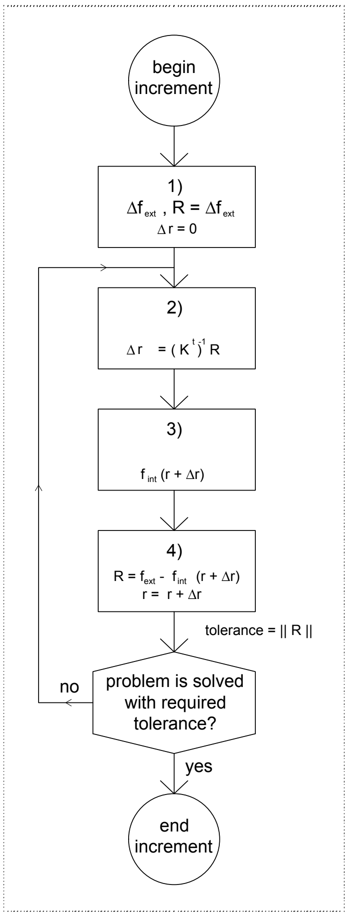


Figure 4.5: Iteration process.

The equilibrium path concept explains the solution process of nonlinear structural analysis. This concept can be graphically represented as response diagrams. The form that is used in this thesis is a load-deflection response diagram. The mechanical behaviour of structures can be characterized by load-deflection or force-displacement response. This response is represented in two dimensions as a representative force quantity **f** against a representative displacement quantity **d**, as illustrated in Figure 4.6.

The term path is a continuous curve in a load deflection diagram, and typically the path is smooth. A smooth path has a continuous tangent, except at exceptional points. The state or configuration of the structure represents each point along the path. The equilibrium path represents configurations in a static equilibrium. Commencement of the response diagram (load is equal to zero, displacement is equal to zero) is characterized by the reference state, this state representing the configuration from which loads and displacements are measured.

In this thesis, material non-linearity is considered as the only source of nonlinear behaviour. Material behaviour depends on the current deformation state and possibly the past history of deformation. Other constitutive variables such as pre-stress, temperature, time, etc. may be involved. The problem representing discrete equilibrium equations at nodes is described by a system of nonlinear algebraic equations

$$\mathbf{f}_{int}(\mathbf{r}) = \mathbf{f}_{ext}, \tag{4.14}$$

where $\mathbf{f}_{int}$ is the internal forces vector depending on unknown displacements $\mathbf{r}$, and $\mathbf{f}_{ext}$ is the external load. The problem can be linearized using a Taylor series expansion of $\mathbf{f}_{int}(\mathbf{r})$. The internal force vector represents nodal equivalent of internal stresses and is defined as

$$\mathbf{f}^{\sigma}_{int}(\mathbf{r}) = B^T\sigma(\varepsilon_{(r)})dV, \tag{4.15}$$

where B is the strain-displacement matrix, σ is stress vector, and ε is strain vector. The Taylor Series expansion of $\mathbf{f}_{int}(\mathbf{r})$ around the point $(\bar{\mathbf{r}})$ is given by

$$\mathbf{f}_{int}(\bar{r}) + \frac{\partial \mathbf{f}_{int}(\mathbf{r})}{\partial \mathbf{r}}\Delta\mathbf{r} = \mathbf{f}_{ext}, \tag{4.16}$$

where $\frac{\partial \mathbf{f}_{int}(\mathbf{r})}{\partial \mathbf{r}}$ represents the tangential stiffness matrix $\mathbf{K}$ of the structure. After substitution, equation 4.14 has the form

$$\mathbf{K}\Delta\mathbf{r} = \mathbf{f}_{ext} - \mathbf{f}_{int}(\bar{r}), \tag{4.17}$$

Newton's method is commonly used to iteratively solve nonlinear systems. Typically, load increment is applied, and the incremental displacement is solved from a linearized, discrete problem. The residual loading is evaluated from the difference of external and internal forces corresponding to achieved displacement, the structure is loaded by residual loading and corresponding incremental change of displacement is evaluated. This iterative process is repeated until the required tolerance is achieved. In this contribution, different variants of the Newton–Raphson method, the modified Newton–Raphson method and the initial stiffness method were considered. The full Newton–Raphson method is based on using the tangent stiffness matrix in each iteration that is formed and only when the direct solver is used in each iteration. The advantage of this method is fast convergence, but it can be computationally expensive for some types of problems.

The modified Newton–Raphson method has the same algorithm, but the stiffness matrix is only updated after a certain number of iterations or at the beginning of each loading step. This approach can be computationally less expensive, as the stiffness matrix should be factorized only when changed. This solution process has a slower convergence rate than the full Newton–Raphson method, however.

The initial stiffness method uses an initial, elastic stiffness matrix constructed only once and is kept constant throughout the solution process. This method is robust but requires a large number of iterations to converge. The optimal

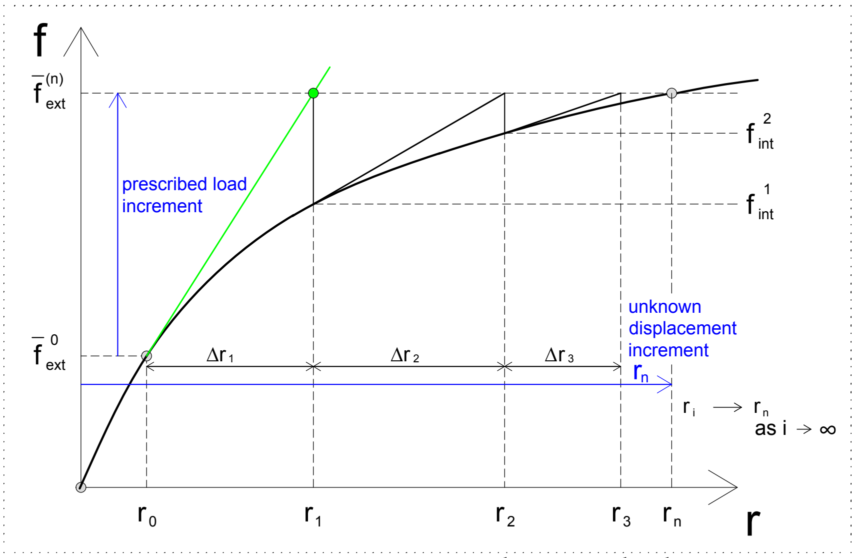


Figure 4.6: Newton–Raphson method.

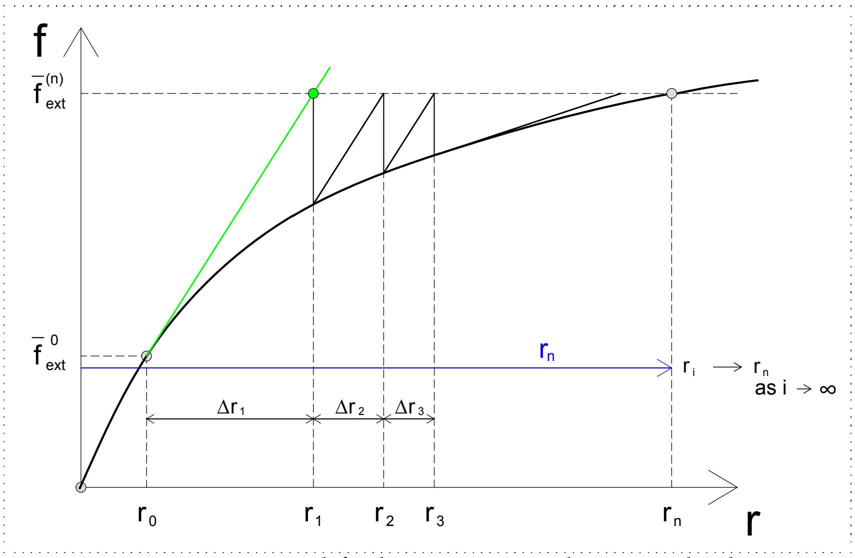


Figure 4.7: Modified Newton–Raphson method.

choice is problem dependent. The iterative solver essentially has the same solution cost for each problem, while a direct solver can profit from existing factorization of the system matrix if there is no change (an iterative solver can also profit, as no pre-conditioner setup is needed), but the effect is not in common with profit from the existing factorization of the system matrix.

The solution process in many cases leads to nonlinear models of complicated geometries and many degrees of freedom. Correctly obtained results directly depend on many attributes, such as discretization, material parameters, equation system solvers, etc. Finite element analysis is based on calculating in the most economical way the solution of the governing partial differential equations with a uniformly distributed error that does not exceed a prescribed threshold. This is achieved by improving discretization in areas where the

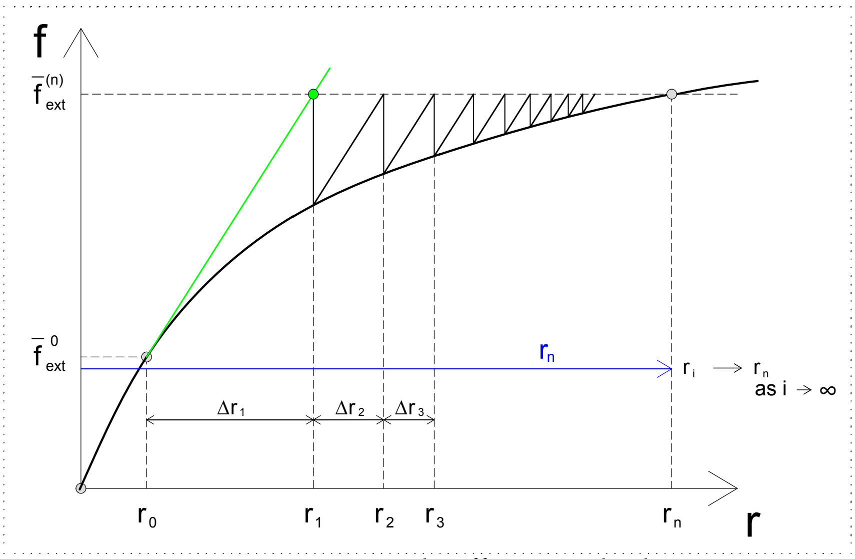


Figure 4.8: Initial stiffness method.

finite element solution is not adequate. It is therefore essential to assess the quality of the approximate solution and be able to enrich discretization.

4.4 INTRODUCTION TO FINITE ELEMENT LOAD BALANCING.

The section examines the solutions of large scale engineering problems by using parallel computation based on a parallel load-balancing framework. This framework can significantly reduce computational time by using available hardware more efficiently. This section presents tune up processes for a parallel load-balancing framework as a part of the solution process in finite element software. Performance of parallel FEM solver depends on domain decomposition. Adapting the decomposition can help to improve the performance significantly, particularly for problems where load is changing. The algorithmic and implementation aspects are discussed. The upgrade approach consists in proposing new technique to evaluate processor weights, which are the input parameters for domain decomposition.

The load-balancing process is driven not only by emerging applications but also by emerging parallel architectures. These parallel architectures span many scales. For example, clusters have become viable alternatives to tightly coupled parallel computers in small scale systems. On the medium scale, supercomputers are constructed as networks of shared memory multiprocessors (SMPs) and have complex and non-homogeneous interconnection topologies. Finally, on the largest scale, grid technologies have enabled computations on widely distributed systems, combining distributed clusters and supercomputers into a single computational resource. These grids are a source of extreme computational power and high network heterogeneity. In order to effectively distribute data from any program on such systems, partitioning must account for heterogeneity in the solution environment. Load balancing requires grouping of computing environment information (e.g. computing speed, memory and network availability) and determining how to apply this information in the load-balancing process (e.g. adjusting computational domain sizes, selecting partitioning algorithms).

The capabilities and performance of the load-balancing framework's depend on its parameters, hardware performance and are illustrated on the solution of selected engineering problems. The advantages of implementing this approach are discussed in this section.

4.4.1 *Introduction and overall design of FEM code*

The main goal of parallel algorithms is partitioning the problem into a set of smaller tasks that can be solved simultaneously. The problem can be partitioned before the solution process, which represents static load balancing. When the problem is partitioned during the solution, it represents dynamic load balancing [26]. Scalability of the solution process is the most important aim in parallel computation.

Ideal scalability is difficult to obtain because of the overhead cost of the parallel algorithm (synchronization and communication) and because some parts of the problem are essentially sequential. A more effective load balancing process that redistributes the work to individual computing units and contributes to optimal scalability is needed nonetheless.

The models are often solved on complex domains leading to many degrees of freedom and often nonlinear effects have to be taken into account. Parallel algorithms should account for load imbalance between particular sub-domains. The imbalance can occur due to several reasons: (i) first load imbalance factor is part of the character of the solution process, as if switching from a linear to nonlinear response in some regions, (ii) second load imbalance factor is an external factor concerning resource relocation, typical in cases when a solution process is run in cluster environments where the available individual computing units are shared by other processes owned by the system or users and lead to changes in allocating processing power. The imbalance is detected in the solution process by monitoring processes during the run time of the process. The decision depends on the amount of load imbalance and the cost of load redistribution. Redistributing the work leads to serializing the problem data (elements, nodes, boundary conditions) into messages sent over the network and a receiving process based on unpacking, followed by a topology update reflecting the new work distribution.

The idea of parallelization strategy is based on the domain decomposition paradigm. In general, two dual partitioning techniques for the parallel distribution of finite element codes exist, node-cut and element-cut strategy [28]. In this thesis, the node-cut strategy, dividing cut dividing the problem mesh into partitions is given in the node-cut technique. The node-cut strategy is presented in chapter 6.1.

Partitioning based on the load balancing process can be affected by many factors. Ideally, no processor during the parallel solving process should be waiting for another processor to complete the solution. The elements as a

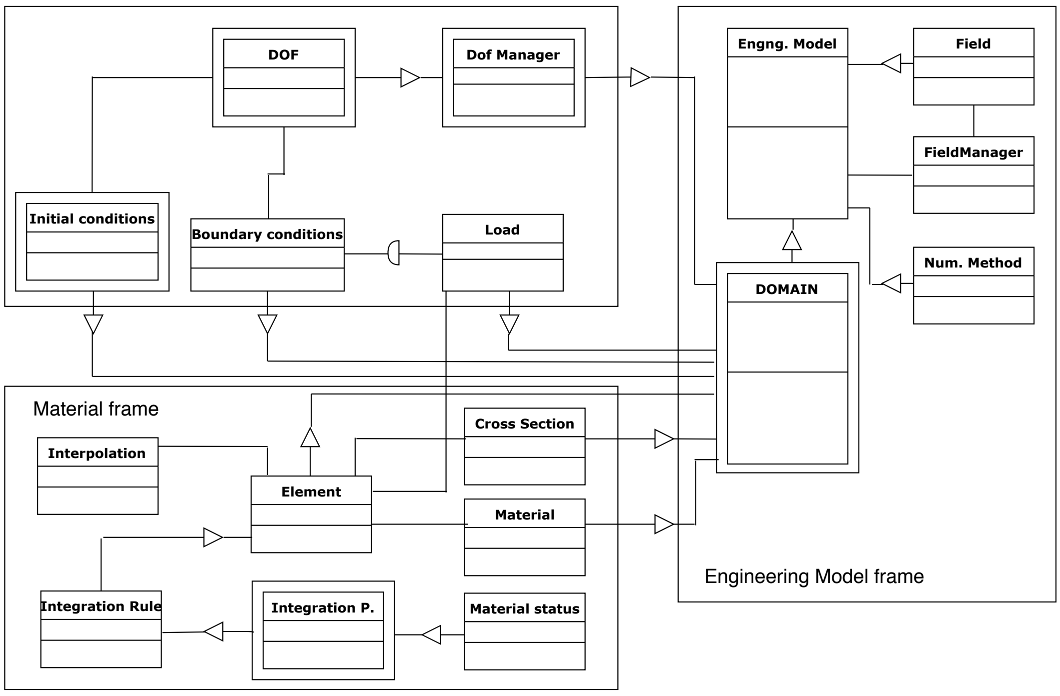


Figure 4.9: General structure of OOFEM.

member of the computational mesh are directly connected to the computational work. The total computational work is determined as the sum of the contribution of individual elements. The computational work of individual element is expressed relatively to computational work of reference element. The load balancing process is based on distributing the overall computational work to an individual processor group with respect to its relative performance. Another significant factor concerns minimizing and reducing communication between computational partitions. The assembly of equilibrium equations for shared nodes requires to sum up contributions from local as well as remote partitions. The cost of accessing remote memory is much higher than the cost of accessing the local memory. It is therefore important to reduce communication between computational partitions.

The present contribution examines the design of an object-oriented framework for static and dynamic load balancing implemented in OOFEM code. The general structure of the OOFEM using the Unified Modeling Language (UML) is presented in Figure 4.9. In the UML diagram, classes are represented by rectangles. Lines with triangles represent the generalization relationship (inheritance) and point to the parent class. Lines with diamonds represent a whole or partial relationship and point to the "whole" class processing the "partial" class. Association is represented by a solid line drawn between classes in the UML.

The ongoing problem is represented by a class illative from the *EngngmModel* class. Assembling the governing equation and using the numerical method (represented by a class derived from the "NumericalMethod" class) to solve the system of equations is the main task of the *EngngmModel* class. The *Domain* class represents the discretization of the problem domain, which maintains the set of objects representing nodes (*DofManager* class and *Node* class), elements (*Element* class), material models (*Material* class), boundary conditions, etc. The *Domain* class is an attribute of the *EngngModel* class. In general, the *Domain* class provides services for accessing particular components. In nonlinear structural analysis, the *EngngModel* class assembles the governing equations for each step of the solution by summing up the contributions from the domain components. For convenience, the implementation of vectors and sparse matrices is provided by corresponding classes. The modular design allows the problem formulation and the numerical solution and sparse storage to be disconnected; they are thus independent of each other.

The *DOF* class represents a single degree of freedom (DOF). It maintains its physical meaning, the associated equation number, and keeps a reference to the applied boundary and initial conditions. The *DofManager* class represents an origin class for entries processing some DOFs. The list of applied loading, and optionally its local coordinate system, is stored in a DOF collection. The *DofManager* class services include methods for gathering localization numbers from the main DOFs, the solution of the applied load vector, and the solution of the transformation into a local coordinate system. The representation of a finite element node or an element side owning some DOFs is typical examples of derived classes. Initial and boundary conditions are represented by the corresponding classes. The particular boundary conditions that can be applied to DOFs (primary boundary conditions), DOF managers (typically nodal load) or elements (Newton or Neumann boundary conditions, surface loads, etc.) are the derived classes from the *BoundaryCondition* class.

Declaring a general interface defined by a set of services is the primary role of these core classes. Some services from the interface can already be implemented by core classes, or implementation can be left to derived classes (so-called virtual methods). Any derived class therefore has to implement interfaces defined by its parent classes (polymorphism). An important consequence of the abstract interface concept is that all derived classes can be handled in the same manner without distinguishing a particular type of object. Some general services can also already be implemented at the parent level by using virtual interface services implemented by derived classes. A detailed description of OOFEM code design can be found in [30].

5 PARALLELIZATION OF LEFT AND RIGHT HAND SIDE ASSEMBLY OPERATIONS

This chapter deals with evaluation of different parallelization strategies of assembly operations for right-hand side vectors and left-hand system matrices, which are one of the critical operations in any finite element software. The global stiffness matrix K and global load vector F are assembled from the individual element and nodal contributions. This process relies on the global numbering of equations. In the case of global vector/matrix assembly, the contributions of individual elements are assembled (added) to the positions in the global load vector according to the global equation numbers of element nodal unknowns. Different assembly strategies for systems with shared memory model are proposed and evaluated, using *Open Multi-Processing* (OpenMP), *Portable Operating System Interface* (POSIX), and *C++11 Threads*. The different frameworks and synchronization techniques are evaluated, involving simple synchronization directives, various block locking algorithms, and finally, intelligent lock-free processing based on colouring algorithms. The different strategies were implemented in a free finite element code with object-oriented architecture (OOFEM).

The existing approaches of assembling have been presented, for example in papers [25], [4], [21]. In [25] the approach is based on OpenMP critical sections and OpenMP atomic directives. The methods reduction on shared memory model represented as OpenMP private clause, OpenMP reduction clause and OpenMP atomic directives are used in paper [4]. In [21], authors illustrated the case where the operation assembles an element contribution into a local matrix which eater assembles into the global matrix using graph colouring as a synchronization multithreaded data operations which is OpenMP non-blocking parallel algorithm (without OpenMP explicit synchronization).

5.1 OPENMP

OpenMP is a shared memory programming model that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran programming languages. It is available for a wide variety of processor architectures and operating systems. It consists of a set of compiler directives, library routines and environment variables that influence run-time behaviour (see [3] for details).

The programming in OpenMP consists in using so called parallel constructs (compiler directives), which are inserted in the source code, instructing the compiler to generate specific code. The OpenMP defines various constructs allowing to parallelize serial code and synchronize the individual threads [29].

The OpenMP *parallel* construct allows to create a team of threads each executing the same code. This construct is often combined with *for* construct to split underling loop into non-overlapping portions executed by individual members of thread team. The programmer can adjust certain parameters that affect the work assignment (static or dynamic), and the scope of variables inside the loop (local or shared). That thread synchronization is still necessary to prevent race condition on global vector/matrix value update.

To reduce granularity of the problem and reduce overhead connected to thread creation and termination, it is usual to parallelize the outermost loops in the algorithm, which in our particular case corresponds to parallelization of the loop over individual elements in the assembly operation.

5.1.1 *Synchronization using Critical Section, Atomic Update, Simple Lock, Nested Lock*

In this subsection, we start with simple solution preventing occurrence of race conditions during data update based on the *critical section*, *atomic update*, *simple lock*, and *nested lock* constructs. They are quire similar, the *critical* construct restricts the execution of the associated statement/block to a single thread at time. Lock can be acquired by a thread. While thread owns the lock, no other thread can get it. This essentially allows to execute specific part of the code by the thread possessing the lock. Once this part of the code is finished, the thread has to release the lock to allow an other thread to acquire the lock. Nested locks are similar to locks, but they can be locked multiple times by the same thread before being unlocked.

The synchronization with using *critical* section is implemented in two variants. In the first variant (marked as A1) the inner loop over element code numbers is enclosed using critical section, the second variant considers only enclosing the actual update operation (A2). The next variant considers only enclosing the actual update operation using atomic update (A3), see Algorithm 3).

The similar approach is followed to synchronize threads using locks. The lock is used to ensure that either single thread can process loop over element code numbers (A1.1 or A1.2 using nested lock) or process update operation (A2.1), see Algorithm 4.

Algorithmus 3 : Prototype code - matrix assembly with explicit synchronization

```
# pragma omp parallel for private($K^e, Loc^e$)
for $elem = 1, nelem$
    $K^e = computeElementMatrix(elem)$
    $Loc^e = giveElementCodeNumber(elem)$
     # pragma omp critical $(A1)$
        for $i = 1, Size(Loc^e)$
            for $j = 1, Size(Loc^e)$
             # pragma omp critical $(A2)$
             # pragma omp atomic $(A3)$
                $K(Loc^e(i), Loc^e(j)) += K^e(i,j)$
```

Algorithm 4 : Prototype code – matrix assembly with explicit locks.

```
omp_init_lock(&my_lock) (A1.1) (A1.2)
omp_init_nest_lock(&my_lock) (A2.1) (A2.2)
# pragma omp parallel for private($K^e$, $Loc^e$)
for $elem = 1, nelem$
    $K^e = computeElementMatrix(elem)$
    $Loc^e = giveElementCodeNumber(elem)$
    omp_set_lock(&my_lock) (A1.1)
    omp_set_nest_lock(&my_lock) (A2.1)
        for $i = 1, Size(Loc^e)$
            for $j = 1, Size(Loc^e)$
                omp_set_lock(&my_lock) (A1.2)
                    $K(Loc^e(i), Loc^e(j)) += K^e(i,j)$
                omp_unset_lock(&my_lock) (A1.2)
    omp_unset_lock(&my_lock) (A1.1)
    omp_unset_nest_lock(&my_lock) (A2.1)
omp_destroy_lock(&my_lock) (A1.1) (A1.2)
omp_destroy_nest_lock(&my_lock) (A2.1)
```

The approach followed in this section is rather conservative, ensuring that only a single thread can perform any update operation. In reality, the race condition on data update can happen only if two or more threads are attempting to update the same entry in global vector/matrix. This essentially means that these threads are assembling contributions of elements sharing the same node(s). The probability of this can be relatively low, so in next sections we try to propose improved algorithms, that allow to perform update operation in parallel provided that different entries of global vector/matrix are updated.

5.1.2 *Synchronization using block locks*

The algorithm presented in this section is based on idea of having an array of locks, each corresponding to consecutive block of values of in global vector/-matrix. Once a specific global value is to be updated by specific thread, the corresponding lock is acquired, preventing other threads to update values in the same block, but allowing other threads to update values in other blocks. It may seem, that the ideal situation is to have unique lock for every global value, but as the global vector/matrices are the dominant data structures (in terms of memory requirements) in a typical FE code, this approach is not feasible. In the presented approach, the individual groups correspond to blocks of rows of global vector/matrix values and the prototype implementation is presented in Algorithm 5.

5.2 POSIX THREADS

The POSIX Threads (*Pthreads*) libraries are standardized thread programming interfaces for C/C++ language. *Pthreads* allows one to spawn a new concurrent processes. *Pthreads* consists of a set of C/C++ language types and procedure calls, see [2] for further details).

The original POSIX Threads application programming interface (API) subroutines can be informally divided into four main groups: Thread manage-

Algorithm 5 : Prototype code – matrix assembly with explicit block locks.

```
# define $NBLOCKS$
omp_init_t_&my_lock $[NBLOCKS]$
for $n = 1, NBLOCKS$
    omp_init_lock(&my_lock $[n]$)
blocksize $= Size(K.rows)/NBLOCKS$
 # pragma omp parallel for private($K^e, Loc^e$)
 for $elem = 1, nelem$
    $K^e = computeElementMatrix(elem)$
    $Loc^e = giveElementCodeNumber(elem)$
        for $i = 1, Size(Loc^e)$
         bI $= Loc^e(i)/blocsize$ $//integer\ division$
            for $j = 1, Size(Loc^e)$
             omp_set_lock(&my_lock $[bI]$)
                $K(Loc^e(i), Loc^e(j)) += K^e(i,j)$
             omp_unset_lock(&my_lock $[bI]$)
omp_destroy_lock(&my_lock $[bI]$)
```

ment, Mutexes, Condition variables, and Synchronization. Routines being part of Thread management allow to create, detach, and joining threads. Constructs allowing to protect code from race conditions are called *mutexes* ("mutual exclusion"). The Mutex group consists of functions for creating, destroying, locking, and unlocking mutexes. Routines allowing communications between threads that share a mutex are part of Condition variables group, which includes functions to create, destroy, wait, and signal based upon specified values. Synchronization group provides routines that can manage read/write lock and barriers.

5.2.1 *Synchronization using Simple Mutex and Recursive Mutex*

The POSIX Threads synchronization routines allow to protect shared data when multiple threads update the data. The concept is very similar to locks in OpenMP library. In POSIX Threads, only single thread can lock mutex variable at any given time. In the case where several threads try to lock mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. The simple mutex can be used only once by a single thread. Attempting to relock the mutex (trying to lock mutex after previous lock) causes deadlock. If a thread attempts to unlock a simple mutex that it has not locked leads to undefined behaviour. The recursive mutex shall maintain the concept of a lock count. When a computing thread successfully acquires recursive mutex for the first time, the lock count shall be set to one. Each time the thread unlocks the recursive mutex, the lock count shall be decremented by one. When the lock count reaches to zero, the recursive mutex shall become available for other threads to acquire. In this section, the prototype algorithms are presented using simple and recursive mutexes to prevent race condition on data update. Three variants are considered, the first marked as (B1.1) using simple mutex to protect localization loop over code numbers, the second, marked as (B2.1) using recursive mutex again protecting the localization loop, and finally the one protecting just update operation using simple mutex, marked as (B1.2). All variants are illustrated in Algorithm 6.

Algorithm 6 : Prototype code – matrix assembly with explicit synchronization using simple and recursive POSIX mutexes.

```
pthread_mutex_tmutex_SIM = PTHREAD_MUTEX_INITIALIZER (B1.1) (B1.2)
pthread_mutex_t mutex_REC (B2.1) (B2.2)
pthread_mutexattr_t REC (B2.1) (B2.2)
int pthread_mutex_tmutexattr_setrecursive (pthread_mutexattr_t REC, int recursive) (B2.1) (B2.2)
 threads = new pthread _t[MAX_THREAD]
 for $i = 0, MAX\_THREAD$
     pthread_create( &threads[i], NULL, Assembly Element Matrix, i)
       for $i = 0, MAX\_THREAD$
     pthread_join( &threads[i], NULL)
 end

 void Assembly Element Matrix ( ... )
   for $elem = 1, nelem$
   $K^e = compute\ Element\ Matrix\ (elem)$
   $Loc^e = give\ Element\ Code\ Numbers\ (elem)$
   pthread_mutex_lock(&mutex_SIM) (B1.1)
   pthread_mutex_lock(&mutex_REC) (B2.1)
      pthread_mutex_lock(&mutex_SIM) (B1.2)
         for $i = 1, Size(Loc^e)$
         for $j = 1, Size(Loc^e)$
            $K(Loc^e(i), Loc^e(j)) += K^e(i, j)$
      pthread_mutex_unlock(&mutex_SIM) (B1.2)
   pthread_mutex_unlock(&mutex_SIM) (B1.1)
   pthread_mutex_unlock(&mutex_REC) (B2.1)
 return
```

5.3 SYNCHRONIZATION USING A COLOURING ALGORITHM

As already discussed in previous sections, the conservative strategy on always protecting the update operation may not lead to optimal results. It enforces the serial execution of the update operation for selected values, regardless if there is a real conflict or not. The fact that it prevents parallel execution can have significant impact on scalability. Partially, this problem has been addressed in the algorithm using array of locks preventing the update to block of row values. In this section, we present an alternative approach which is based on idea of assigning the individual elements into groups, where the elements in a group should not share any node. This essentially means that the elements in a group can be processed concurrently as during the assembly operation only distinct values can be updated. The algorithm loops over the groups and each group is processed in parallel. The objective is to keep the number of groups minimal. This is known as so-called colouring algorithm in graph theory [37].

In this paragraph the introduction to the vertex colouring algorithm will be given. Consider a graph of n mutually connected vertices (representing FE elements). The edges (connections) represent the element connectivity, i.e, the edge between two vertices represents a case, when two elements share the same node. The task is to assign a "color" to each vertex under the condition that no neighbor has the same color and keep the number of "colors" minimal. The algorithm for greedy colouring of a graph is following:

1. - Loop over the vertices $i = 1, N$
2. - Find the colors assigned to neighbors for i
   $C = \cup_{neighboursofi} f(1)$
3. - Find the lowest (available) color and assign it to node i:
   - loop over available colors
   - if color not in C than $f(1) = color$

The computational cost of the solution of Greedy colouring algorithm depends heavily on the vertex ordering. In the worst case the behaviour is poor and solution of algorithm can take a lot of computation time. On the other hand, the graph construction and graph colouring has to be performed only once during the FE code initialization and after that it should be reused in any assembly operation. As already noted, the colouring algorithm splits the elements into groups marked with different colours. The algorithm ensures that a minimum number of colours is used. Once the colouring is available, the assembly algorithm consists of the outer loop over individual colour groups and inner loop over individual elements in a group. Inside inner loop, the element contributions are evaluated and assembled into global vector/matrix. Now the key is that the inner loop can be parallelized (individual members of a group can be processed by different threads) without need of synchronization, as the colouring ensures that the race condition on update could not occur.

Even though this is appealing, the algorithm has its overhead. This includes already discussed need to establish the colouring, but as also pointed out only the inner loop can be parallelized. All threads should finish processing elements with specific colour before processing elements from the next one, This requires synchronization. Finally, there is also an overhead connected to creation and termination of threads for each colour.

The prototype code for colouring assembly is presented in Algorithm 7 using OpenMP directives. The additional arguments of parallel loop directive are used to declare some variables as shared (i.e. each thread accesses the same variable) or private (each thread has its own copy of that variable). The for-loop clause allows accumulating a shared variable without explicit synchronization.

The POSIX Threads variant of the colouring based assembly algorithm is presented in Algorithm 8.

Algorithm 7 : Prototype code – matrix assembly without explicit synchronization using a Colouring Algorithm *(OpenMP).*

```
$ConnectivityTable * ct = domain \rightarrow giveConnTab()$
$std :: vector < std :: vector < int >> colourGroup$
$colourGroup = ct \rightarrow AssembleColAlg()$
for $colPointer = 0, Size(colourGroup.size())$
# pragma omp parallel for private($K^e, Loc^e$)
  for $colIdElem = 1, Size(colourGroup[colPointer].size())$
   $elem = colourGroup[colPointer][colIdElem]$
   $K^e = computeElementMatrix(elem)$
   $Loc^e = giveElementCodeNumber(elem)$
     for $i = 1, Size(Loc^e)$
       for $j = 1, Size(Loc^e)$
         $K(Loc^e(i), Loc^e(j)) += K^e(i,j)$
```

Algorithm 8 : Prototype code – matrix assembly without explicit synchronization using a Colouring Algorithm *(POSIX Threads).*

```
$ConnectivityTable * ct = domain \rightarrow giveConnTab()$
$std :: vector < std :: vector < int >> colourGroup$
$colourGroup = ct \rightarrow AssembleColAlg()$
for $colPointer = 0, Size(colourGroup.size())$
$threads = new\ pthread\_t[MAX\_THREAD]$
for $i = 0, MAX\_THREAD$
 pthread_create( &threads[i], NULL, Assembly Element Matrix, i)
   for $i = 0, MAX\_THREAD$
 pthread_join( &threads[i], NULL)
end

void Assembly Element Matrix ( ... )
 for $elem = 1, nelem$
  $K^e = compute\ Element\ Matrix\ (elem)$
  $Loc^e = give\ Element\ Code\ Numbers\ (elem)$
    for $i = 1, Size(Loc^e)$
    for $j = 1, Size(Loc^e)$
      $K(Loc^e(i), Loc^e(j)) += K^e(i,j)$
for $colPointer = 0, Size(colourGroup.size())$
return
```

5.4 C++11 THREADS

Alongside well established OpenMP and Posix Threads, the C++11 standart has introduced native C++ thread support [38]. The C++11 Thread libraries include utilities for creating, managing threads which are standardized for C/C++ language. The C++11 standard library contains classes for thread manipulation and synchronization, common protected data, and low-level atomic operations.

The parallel program based on C++11 standard library is constructed by defining a new procedure function which is executed by the thread and start the new thread. The synchronization in the C++11 standard is achieved by classical synchronization mechanisms like mutex object, condition variables, and other mechanisms like locks or controlling features used when threads are transferring computational data.

5.4.1 *C++11 Threads - Simple Lock*

In this paragraph, the multitasking synchronization using mutex class is presented. The mutex can be used to protect shared data from being simultaneously accessed by multiple threads. The solving thread owns a mutex from the moment that thread successfully calls either *"lock"* or *"try lock"* until thread calls *"unlock"*. When a thread owns a mutex, no other threads in the parallel block will be refused to gain the lock or receive a false return value for *"try lock"* call if they attempt to claim ownership of the mutex. The behaviour of a program is not defined when a mutex is destroyed while still owned by any thread, or a thread terminates while owning a mutex.

The synchronization is enforced in the localization loop, as illustrated in Algorithm 9.

Algorithm 9 : Prototype code – matrix assembly with explicit synchronization using simple locks *(C++11 Threads)*.

```
std :: mutexMTX
threads = new thread _t[Number_Of_Threads]
for i = 0, Number_Of_Threads
    std::thread[Assembly Element Matrix, &arg[i] ]
      for i = 0, Number_Of_Threads
    std.[i].join()
end

void Assembly Element Matrix ( ... )
  for elem = 1, nelem
   $K^e$ = compute Element Matrix (elem)
   $Loc^e$ = give Element Code Numbers (elem)
  MTX.lock()
      for i = 1, Size($Loc^e$)
      for j = 1, Size($Loc^e$)
         $K(Loc^e(i), Loc^e(j)) += K^e(i, j)$
  MTX.unlock()
return
```

5.5 PERFORMANCE EVALUATION

In this section, the performances and efficiencies of the presented approaches are compared both for matrix and vector assembly operations using two benchmark problems.

The first benchmark problem is 3D model of nuclear containment, marked as *Jete* and shown in Figure 5.1. The second benchmark problem is model of 3D porous micro-structure, marked *Micro* and shown in Figure 5.2.

For both benchmark problems the different discretizations are generated with increasing number of elements, see Table 5.1. The tetrahedral elements with linear approximation have been used in a case of nuclear containment benchmark, while structured grid of brick elements with linear interpolation were used for micro-structure benchmark. The porous micro-structure consists of two phases with different set of elastic constants.

In both cases, the linear structural analysis has been performed and structures were loaded by self-weight, which implied nonzero contribution of

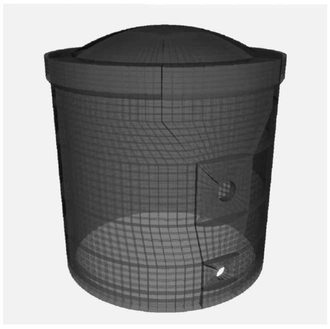


Figure 5.1: Benchmark problem of a 3D finite element model of a nuclear containment dome (*Jete*).

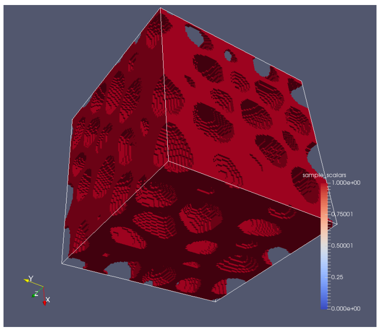


Figure 5.2: Benchmark problem of a 3D finite element model of a cube of porous micro-structure (*Micro*).

every element to the external load vector. The benchmark problems are characterized by different sparsity of the system matrix. The model of porous micro-structure has significantly more nonzeros members than the model of nuclear containment. For example, a number of nonzeros members in the stiffness matrix of the problem is $433M$ in case of *Jete3M*, while the model of the porous micro-structure *Micro3M* has $1528M$ nonzero entries, while number of unknowns is very similar.

All the presented parallelization approaches have been implemented in OOFEM finite element code. The object oriented C++ OOFEM code has been compiled using *g++ 4.5.3.1* compiler version with optimalization flags *-O2*. The tests have been executed on Linux workstation (running Ubuntu 14.04 OS) with two CPUs *Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz* and *132GB RAM*. Each CPU consists of eight physical and sixteen logical cores, allowing up to thirty-two threads to run simultaneously on the workstation. All the tests fit into system memory.

For each benchmark problem, the individual strategies have been executed on increasing number of processing cores and speedups (with respect to serial single CPU execution) have been evaluated as an average from three consecutive runs. The performance of individual strategies of vector assembly (in terms of achieved speedup versus increasing number of processors) for *Jete250k* test are presented in Figure 5.3 and for matrix assembly in Figure 5.4. Similarly, for the *Jete3M* test the results are presented in Figures 5.5, 5.6, for *Micro250k* test in Figures 5.7, 5.8 and finally, for the *Micro3M* test in Figures 5.9, 5.10. The following notation is used on above mentioned figures

name	nnodes	nelems	neqs
Jete250k	$87k$	$959k$	$260k$
Jete3M	$899k$	$1M$	$3M$
Micro250k	$85k$	$80k$	$256k$
Micro3M	$1M$	$970k$	$3M$

Table 5.1: Discretizations of the benchmark problems considered.

name	Colouring [s]	matrix [s]	vector [s]
Jete250k	2.6	2.8	1.7
Jete3M	436	36.8	20.6
Micro250k	3.7	3.2	2.0
Micro3M	509	39.3	22.1

Table 5.2: Coloring algorithm solution times compared with matrix and vector sequential assembly times of the benchmark problems considered.

to distinguish individual parallelization strategies implemented: OpenMP with critical sections (*OMP - CS*), OpenMP with simple locks (*OMP - L*), OpenMP with nested locks (*OMP - NL*), OpenMP with critical sections only in update operation of global matrix/vector (*OMP - LCS*), OpenMP with atomic update directive only in update operation of global matrix/vector (*OMP - LATO*), OpenMP with simple locks only in update operation of global matrix/vector (*OMP - LL*), OpenMP using blocks of simple locks (*OMP - B50, OMP - B500, OMP - B10$*), OpenMP based on colouring (*OMP - CP*), POSIX Threads based on colouring (*PTH - CP*), POSIX Threads with simple mutexes (*PTH - M*), POSIX Threads with recursive mutexes (*PTH - RM*) and finally C++11 Threads with simple mutexes (*THR - M*). The execution times for colouring algorithm as well as times for matrix/vector assembly operations are presented in Table 5.2. The individual parallel strategies are compared to serial simple execution (matrix and vector assembly) which represents the speedup 1 in all presented graphs.

The results for the vector assembly show very similar trend. All the strategies yield approximately same results in terms of scalability and speedups. From the results it is evident, that the speedups are far from ideal, for example in the case of *Jete3M* test, the speedups for 16 CPUs are in range 2.5-4. There are several possible reasons why only sub-optimal scalability has been obtained. The first reason is that the individual tasks are not independent, in many strategies the update of the global entry is a part that can be processed only by one thread at a time. Second, there is an additional overhead connected with the parallel algorithm (thread creation and management, synchronization) that is not present in the serial version. Third, the individual threads share the common resources (memory bus), which do not scale in performance appropriately. Moreover, from the results, one can observe the performance drop after reaching the limit on 16 threads. This can be attributed to the hyper-

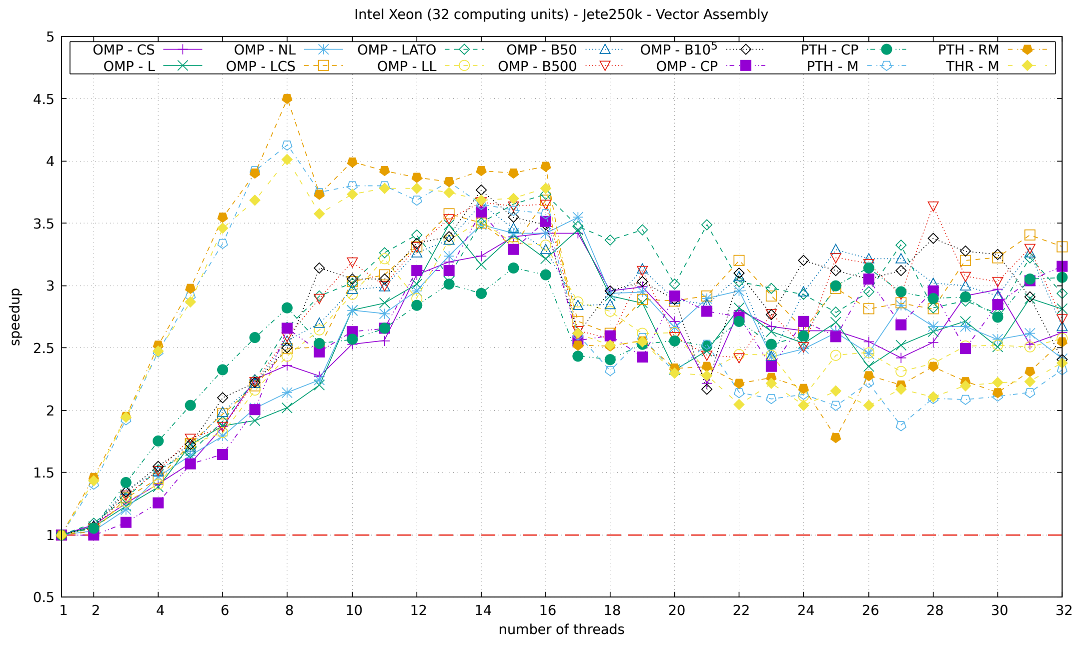


Figure 5.3: Speedups of the external force vector assembly for benchmark *Jete250k*.

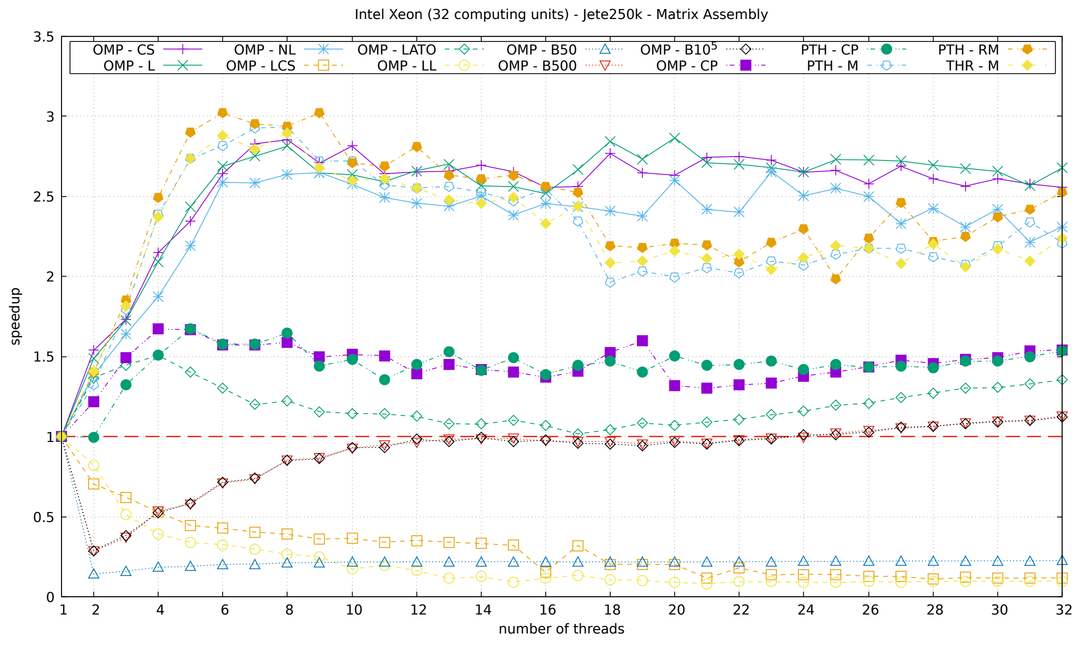


Figure 5.4: Speedups of the sparse matrix assembly for benchmark *Jete250k*.

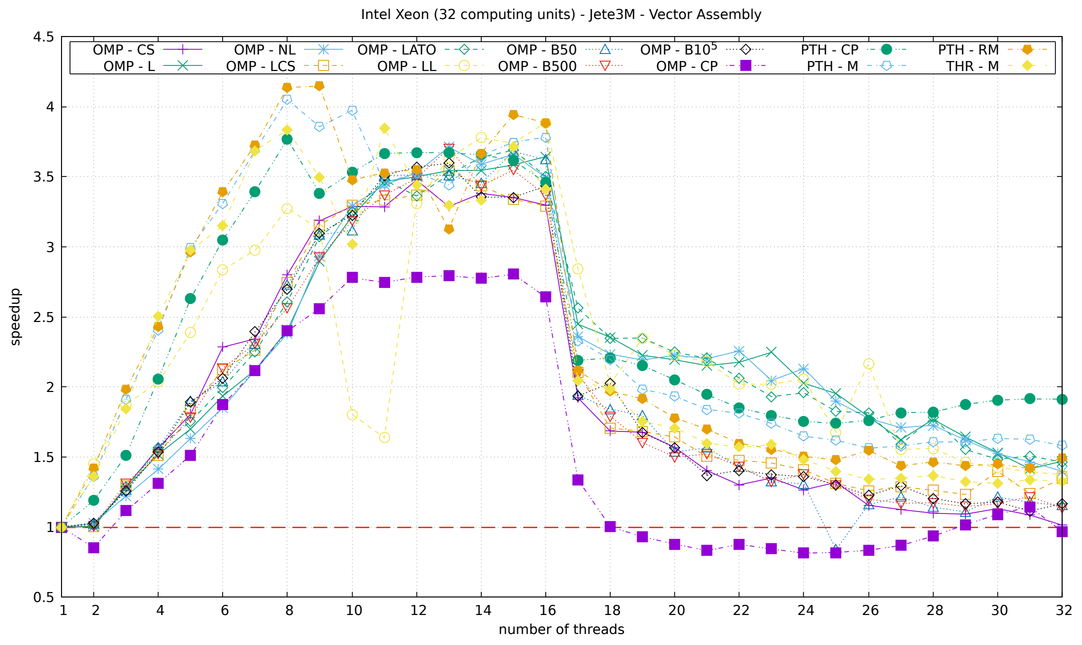


Figure 5.5: Speedups of the external force vector assembly for benchmark *Jete3M*.

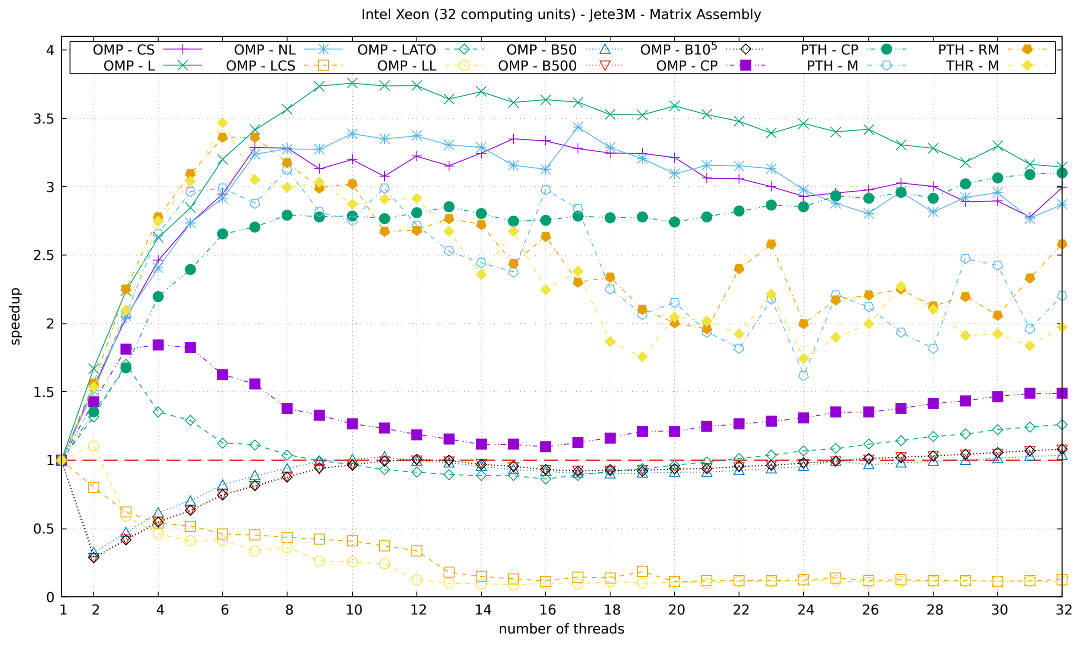


Figure 5.6: Speedups of the sparse matrix assembly for benchmark *Jete3M*.

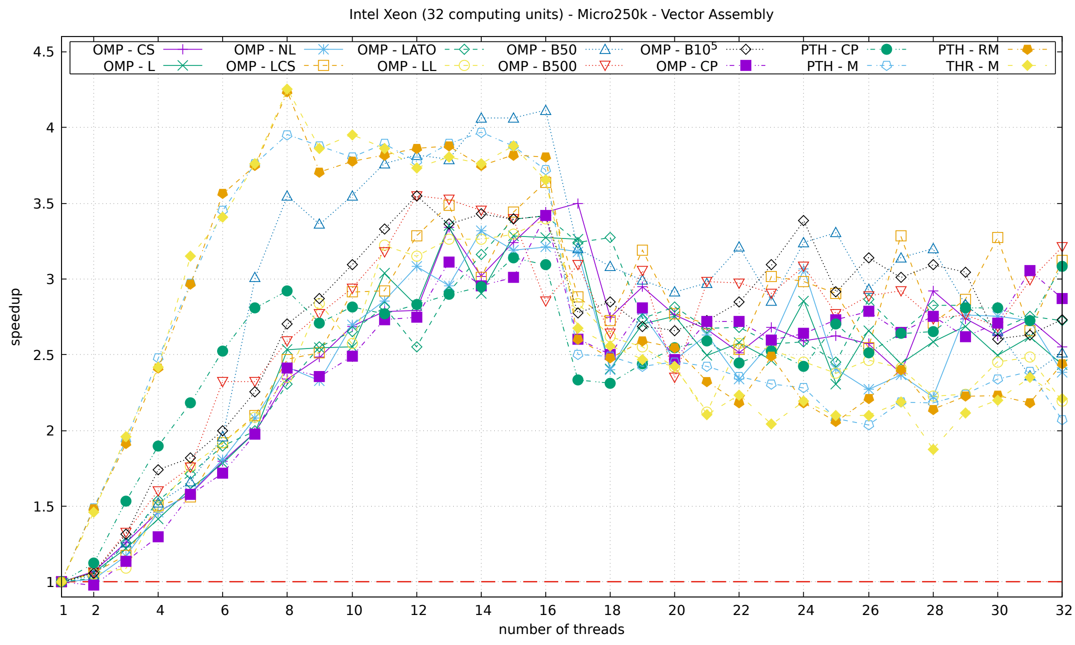


Figure 5.7: Speedups of the external force vector assembly for benchmark *Micro250k*.

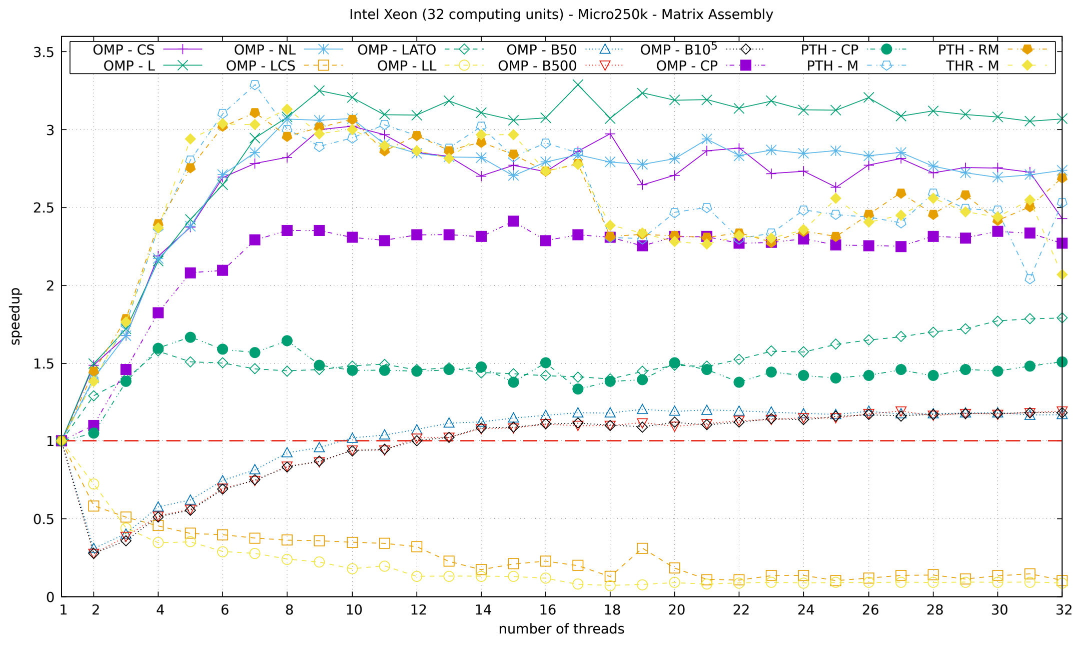


Figure 5.8: Speedups of the sparse matrix assembly for benchmark *Micro250k*.

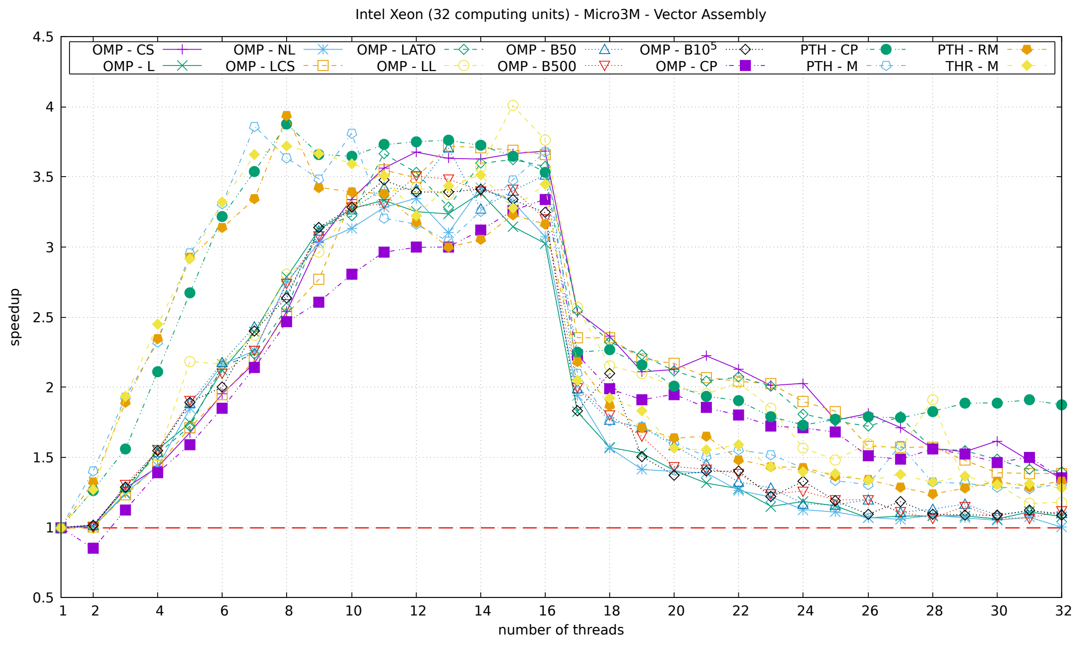


Figure 5.9: Speedups of the external force vector assembly for benchmark *Micro3M*.

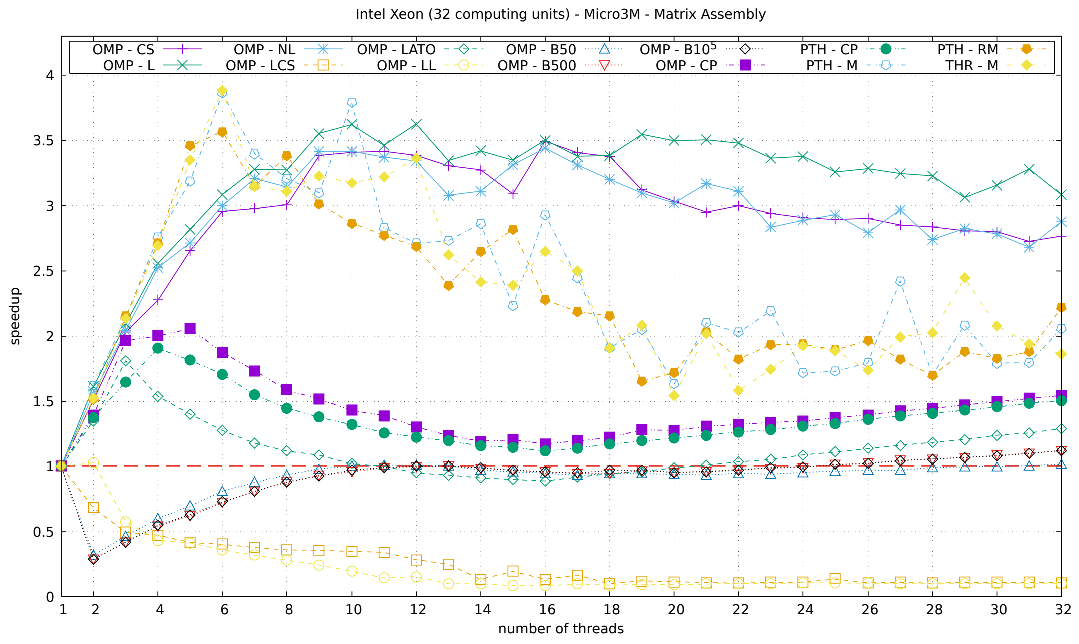


Figure 5.10: Speedups of the sparse matrix assembly using shared libraries on benchmark *Micro3M*.

	OMP-CS	OMP-CP
I refs	1,249,522,199,636	1,277,519,611,096
I1 misses	71,451,623	110,421,787
LLi misses	11,619	14,950

Table 5.3: Number of reCache misses of the benchmark problem *Jete250k*.

threading technology specific to Intel processors [16], which shares some of CPU resources (execution engine, caches, and bus interface) between the hyper-threaded cores. This trend is more pronounced on larger tests (*Jete3M* and *Micro3M*). The POSIX threads and C++11 Threads implementations show better performance than OpenMP versions, particularly for smaller number of threads.

The results for the sparse matrix assembly show different trends. Some strategies (*OMP-LCS, OMP-LL, OMP-B50, OMP-B500, OMP-B10*^{5}*)* were not even able to reach the performance of the serial algorithm and speedups are negative or less than 1. The Colouring based strategies (*OMP-CP, PTH-CP*) have similar speedup trend on *Jete250k* and *Micro3M* benchmark problem. On the other hand, the Colouring based strategy *PTH-CP* shows clearly better scalability than Colouring based strategy *OMP-CP* on benchmark *Jete3M*. The opposite trend can be observed in Colouring strategies on benchmark *Micro250k* (*OMP-CP* show clearly better scalability than *PTH-CP*). The implementations based on colouring algorithm do not perform well, which is somehow surprising observation. This could be (partially) explained by so-called false sharing. Typical SMP system has local data cache organized hierarchically in several levels [22]. When the processor needs to read or write a location in memory, it checks if a corresponding entry is in the cache. If the memory location is in the cache, so called cache hit has occurred. When the memory location is not in the cache, a cache miss has occurred and in this case the data are read or written to the data in the cache. Data is transferred between memory and cache in blocks of fixed size, called cache lines. The system guarantees cache coherence. In case, when one core modifies the memory location that also resides on the different core cache line then the false sharing occurs. This is going to invalidate the cache lines on other cores and forcing to re-read them every time when any other thread has updated the memory in cache lines. The false sharing seems to have much bigger impact on sparse matrix assembly performance than on the vector assembly performance. The false sharing effect in our case is confirmed using valgrind tool with memory management and threading bugs monitoring. The benchmark problem *Jete250k* with using *OMP-CS* or *OMP-CP* based on 32 computational threads is used as a representative example for illustrating false sharing effect and the outputs from valgrind are presented in Table 5.3. From the reported results, one can clearly see the significantly increased number of cache misses in case of colouring algorithm compared to approach using synchronization based on OpenMP critical sections..

In [21] the matrix assembly kernels, based on the OpenMP parallel algorithm with graph colouring was used and achieved speedup 8 for 32 computational threads is reported. However, the obtained results correspond to parallel code based on mixed memory model (combination of distributed and shared memory) and it is difficult to compare with results achieved in this chapter with using only shared memory model.

POSIX thread and C++11 Threads implementations performed best for lower number of threads, but overall the OpenMP implementation (*OMP-L, OMP-NL*), and *OMP-CS* performed the best.

The better results have been reported in literature. In [25] authors reported speedup 11 for 12 threads. However, these results are achieved on a slightly different architecture, where the computational node consist from two processors Intel Xeon X5650 2,66GHz with 6 cores and results for 12 threads. Therefore the results are not affected hyperthreading. The paper [4] present the achieved speedup 9 (atomic directives) for 32 threads on SGI Origin 2000 computer. This better result is obtained on different architecture and it is difficult to compared with the results achieved in our work..

5.6 CONCLUSIONS

The chapter evaluates different parallelization strategies of right-hand side vector and stiffness matrix assembly operations which are one of the critical operations in any finite element software. The performance strategies use different techniques to ensure consistency and are implemented using OpenMP, POSIX Threads and C++11 Threads libraries. The performance of individual strategies and libraries is evaluated using two benchmark problems (3D structural analysis of nuclear containment and of 3D micro-structure) each with two different mesh sizes. For the particular benchmark cases considered, the performance of nearly all strategies has been much better than the performance of serial algorithm with relatively good scalability, however, in case of matrix assembly, considerable differences exist and the presented work provides an insight on how to select optimal strategy.

The achieved results clearly show performance drop on systems with hyperthreading technology when number of processes exceeds number of physical cores. Somehow disappointing are the results of assembly based on Colouring approach, that did not performed as excepted, with performance affected most likely with false-sharing phenomena.

The main conclusion of this study is that the performances of individual libraries are comparable, but performances of individual strategies differ, often significantly.

In general, the presented chapter illustrates the potential of parallel assembly operations and importance of benchmarking, allowing to identify an optimal strategy.

6 DIRECT SOLUTION OF LARGE SPARSE SYSTEMS OF LINEAR EQUATIONS

The solution of large, sparse systems of linear equations as a critical operation in finite element software is the subject of this chapter. A system of linear equations is called sparse only if a relatively small number of its matrix elements are nonzero. Numerical modelling using high performance computers provides new opportunities. Many engineering problems lead to the solution of sparse linear systems of equations. This is also true of the FEM, which is used here as a representative example. The global stiffness sparse matrix stored as a dense matrix of size K would be prohibitively expensive. One of the key features of the FEM is that the stiffness matrix is typically a positive, definite, symmetric matrix with a sparse structure, which is a consequence of using interpolation and test functions with limited nonzero support. An efficient algorithm for solving a linear system must exploit the symmetry and sparse structure by saving considerable memory and CPU resources. Several different storage schemes exist for sparse matrices. The most widely used are the *skyline*, *compressed rows* and *compressed columns* formats.

Several different libraries are available that solve sparse linear systems of equations. The linear equation solvers can essentially be divided into methods based on either direct or iterative algorithms. The aim of this chapter is to evaluate the efficiency of different methods that can be found in any finite element software solving large, sparse, non-symmetric systems of linear equations on high performance machines. The contribution focuses on comparing the efficiency of different existing libraries in solving large, sparse, non-symmetric systems of linear equations based on either direct or iterative algorithms.

6.1 THE DIRECT SUPERLU SOLVER

This section presents an introduction to direct SuperLU solver a general description of the direct SuperLU solver. The SuperLU solver is a general purpose library for directly solving large, sparse, non-symmetric systems of linear equations. The SuperLU library is an open-source library with object-oriented architecture, written in C++ programming language. In our case, the matrix K is a square, non-singular $n\ x\ n$ sparse matrix, and r and F are dense matrices (vectors). The kernel algorithm in SuperLU uses sparse Gaussian elimination. The first step of a high-level algorithm is triangular factorization $P_r D_r K D_c P_c = LU$, where D_r and D_C are diagonal matrices to equilibrate the system, and P_r and P_c are permutation matrices. Premultiplying K by P_r reorders the rows of K, and post-multiplying K by P_c reorders the columns of K. Permutation matrices are selected in order to improve sparsity, numerical stability and parallelism. L is a lower triangular matrix, and U is

$$K = \begin{array}{ccccc|c} 1 & 2 & 3 & 4 & 5 & \\ \hline \mathbf{8} & 6 & 0 & 2 & 11 & 1 \\ 0 & \mathbf{0} & 0 & 3 & 5 & 2 \\ 9 & 0 & \mathbf{2} & 6 & 0 & 3 \\ 4 & 0 & 0 & \mathbf{0} & 13 & 4 \\ 7 & 1 & 0 & 0 & \mathbf{3} & 5 \end{array}$$

K_nzval	8	6	2	11	3	5	9	2	6	4	13	7	1	3

K_colind	1	2	4	5	4	5	1	3	4	1	5	1	2	3

K_row_ptr	1	5	7	10	12

Figure 6.1: Compressed row format of matrix schema.

an upper triangular matrix. To solve $K * r = F$, we evaluate $r = K^{-1} * F = (D_r^{-1} P_r^{-1} LU P_c^{-1} D_c^{-1}) F = D_c(P_c(U^{-1}(L^{-1}(P_r(D_r F)))))$. The solution to this equation is based on multiplying from right to left in the last expression, which means scale rows of F by D_r. Multiplying $P_r F$ means permuting the rows of $D_r F$. Multiplying $L^{-1}(P_r D_r F)$ means solving triangular systems of equations with matrix L by substitution. Similarly multiplying $(U^{-1}(L^{-1}(P_r D_r F))$ means solving triangular systems with U. In addition, to complete factorization, the SuperLU platform also has limited support for the incomplete factorization (ILU) preconditioner, which approximately solves $K * r = F$. The SuperLU routines appear in three different variants: sequential, multi-threaded (shared memory systems) and parallel (distributed memory systems). The libraries use variations of Gaussian elimination (LU factorization) that are optimized to take advantage of both sparsity and computer architecture, in particular, memory hierarchy and parallelism.

The interface to the SuperLU library was implemented in OOFEM by the author by developing new classes representing to include new classes representing the SuperLU solver and sparse matrix storage. In this contribution, the compressed row format is used in a sequential and multi-threaded SuperLU library. This choice comes from the SuperLU library, which requires a sparse matrix in this format on input. In the compressed row format, only nonzero entries of the sparse matrix are stored in a one-dimensional array. An additional integer array is needed to store the column indices of the stored values. In practical implementation, the one-dimensional arrays storing nonzero values are merged into a single array containing all nonzero entries, row by row. The same applies to arrays of column indices. An additional array with a size equal to the number of rows is needed to point to the beginning of each row record (Fig. 6.1).

The SuperLU library has a number of optional arguments that control the behaviour of the SuperLU library. The user can instruct the solver how the linear systems should be solved according to some known characteristics of the system. The element in the diagonal of a matrix by which other elements are divided into an algorithm, for example by Gaussian elimination, is called the pivot element. Partial pivoting is the interchange of rows in order to place a particularly good element in the diagonal position before executing a particular operation. Two permutation matrices are involved in the solution process. The actual factorization is $P_r A P_c^T = LU$, where P_r is determined from partial pivoting (threshold pivoting), and P_c is a column permutation, usually

to make the L and U factors as sparse as possible.
The SuperLU library provides two types of driver routines to solve a system of linear equations. The driver routines can handle a compressed row format of matrix schema. A simple driver solves the system $A * X = B$ by factoring A and overwriting B with solution X. This expert driver uses the symmetry of the matrix in a linear system to its advantage. SuperLU drivers order the columns of a matrix based on multiple minimum degrees applied to the structure of $A^T + A$.

As noted in the previous section, a distributed memory programming model does not provide global address space or global memory accessible to all processing nodes but distributes memory between the processing units. The distributed memory model can enable large scale problems to be solved by using distributed memory resources. For large problems, it is therefore essential to establish a distributed representation of the sparse matrix. This feature is also supported by the SuperLU library, allowing the user to split the global sparse matrix stored in compressed row format into blocks, representing compressed row storage for consequent, non-overlapping groups of rows, which are distributed across computing nodes.

We assume that decomposition of the discretized problem domain has been established and individual, non-overlapping sub domains (partitions) have been assigned to and stored locally on individual computing nodes. In general, two dual partitioning techniques for the parallel distribution of finite element code exist [28]. In our case, the cut dividing of the problem mesh into partitions can be done using the node-cut technique. The node-cut strategy is assumed when cut dividing the problem domain runs through the nodes. Nodes on mutual partition boundaries are called shared nodes, and nodes inside individual partitions are called local nodes. The node-cut partitioning scheme can be interpreted as mesh decomposition using cuts passing through shared nodes of the mesh without crossing any element (Fig. 6.2). The cut strategy ensures duplication of the shared nodes. Each processing node is responsible for assembling its contribution to the global system matrix by summing the contributions from individual elements. In order to minimize communication, it is natural that each processing node will assemble and maintain in its local memory the block of global stiffness matrix rows corresponding to unknowns on its partition. This is uniquely defined for local nodes, which are exclusively shared by local elements on individual partitions. For shared nodes, which are shared by elements from multiple partitions, ownership must be defined. In this work, the partition with the lowest rank sharing the shared node is the owner of the shared node and thus responsible for maintaining the corresponding row entry of the global stiffness matrix. It is clear that for shared nodes, the contributions to the corresponding row have to be received from partitions sharing a particular node. The assembly process requires global, unique numbering of equations to be established across the computing nodes in addition to local numbering on individual partitions.

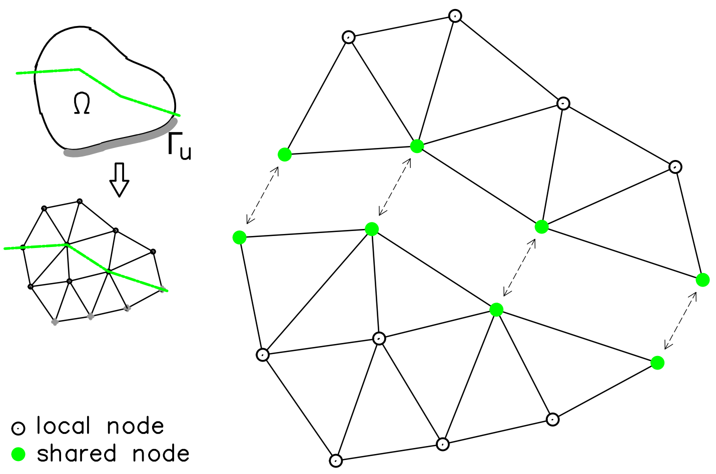


Figure 6.2: Node cut partitioning.

The assembly process of the global, distributed system matrix requires two steps: setup of the data structures required to store the block of compressed row records on individual partitions, and the assembly process itself based on localizing individual element contributions into global equilibrium equations according to their connectivity.

The first stage should determine the required size of arrays for storing all nonzero entries of the global matrix. The nonzero contributions of each element can be identified from element code numbers. The code numbers simultaneously represent the numbering of equilibrium equations and the numbering of corresponding unknowns. The individual values in the stiffness matrix represent nodal forces caused by corresponding unit displacement. The entry in the element stiffness matrix representing the nodal force contributing to the k-th equilibrium equation and multiplied by the displacement located in the global displacement vector at the l-th position should be added to the position (k, l) in the global matrix. By assuming that element contributions are full matrices, we can initialize the nonzero structure of the global stiffness matrix using element code numbers. In a distributed case, each partition is responsible for initializing and preallocating its block structure of compressed row records. In the case of compressed row storage, the number of nonzero entries in each row has to be determined together with the corresponding nonzero column positions of the individual entries.

Each processing node is responsible for initialization and assembly of the assigned block of rows. This block is composed of local contributions as well as shared node contributions from neighbouring partitions. Therefore, the overall initialization can be divided into

- a local stage, where local contributions to the nonzero pattern are identified (from the partition elements).
- a communication stage, where contributions from neighbouring partitions are received for locally maintained rows, corresponding to equilibrium equations for shared nodes.

Any partition, therefore, has to keep its locally maintained block of compressed rows and block compressed rows that correspond to shared nodes as not locally maintained. These two blocks can be stored separately. In the initial stage, the block structure is initialized on every partition using only the contributions from local elements. The row entries that correspond to the shared nodes and are not locally maintained are then communicated (sent) to the corresponding partition maintaining the corresponding row. After the data is sent, the contributions from neighbouring partitions containing the contributions to the locally maintained shared equations are received. Non-blocking communication is used to send and receive contributions. Non-blocking communication allows the potential overlap of communication and computation to be benefited from and is in fact used in actual implementation. After finishing the communication stage, each partition has a fully initialized data structure for a locally maintained block of rows.

After finishing the initialization step in which memory was allocated for every nonzero entry of the globally distributed stiffness matrix, we can proceed with the assembly of the matrix from the element contributions. This is done in a similar fashion to the previous step. The contributions from local elements are assembled on each partition. The rows corresponding to shared nodes not being locally maintained are then sent to the partition maintaining the corresponding row. Finally, each partition receives a remote contribution to the locally maintained rows of shared nodes. This process is illustrated in Figure 6.1. In this example, the problem has three elements (also corresponding to sub-domains) and eight nodes, from which five are local on individual partitions and three nodes are shared.

The process of assembling the global right hand side vector is very similar to the process of assembling the stiffness matrix, but in many aspects is simpler because of the assembly of local vector contributions. The global vector is again distributed between partitions, the distribution corresponding to the distribution of matrix rows. The contributions from local elements are assembled first, then those corresponding to rows maintained on remote partitions are communicated, and the contributions from neighbours are received and entries are updated.

By completing the steps described above, the distributed sparse matrix, represented by locally maintained blocks of row entries on individual partitions, can be directly passed into SuperLU routines.

6.2 SEQUENTIAL SOLVERS

The different methods based on direct or iterative algorithms for solving linear equations are compared in this section. A linear elasticity problem is considered. In particular, a direct solver using Skyline sparse storage, a direct solver from the sequential SuperLU library, and an iterative solver from the Iterative Method Library (IML) are compared in this section.
The Skyline sparse storage format is useful for direct sparse solvers and an ideal format for Cholesky or LU decomposition when no pivoting is required.

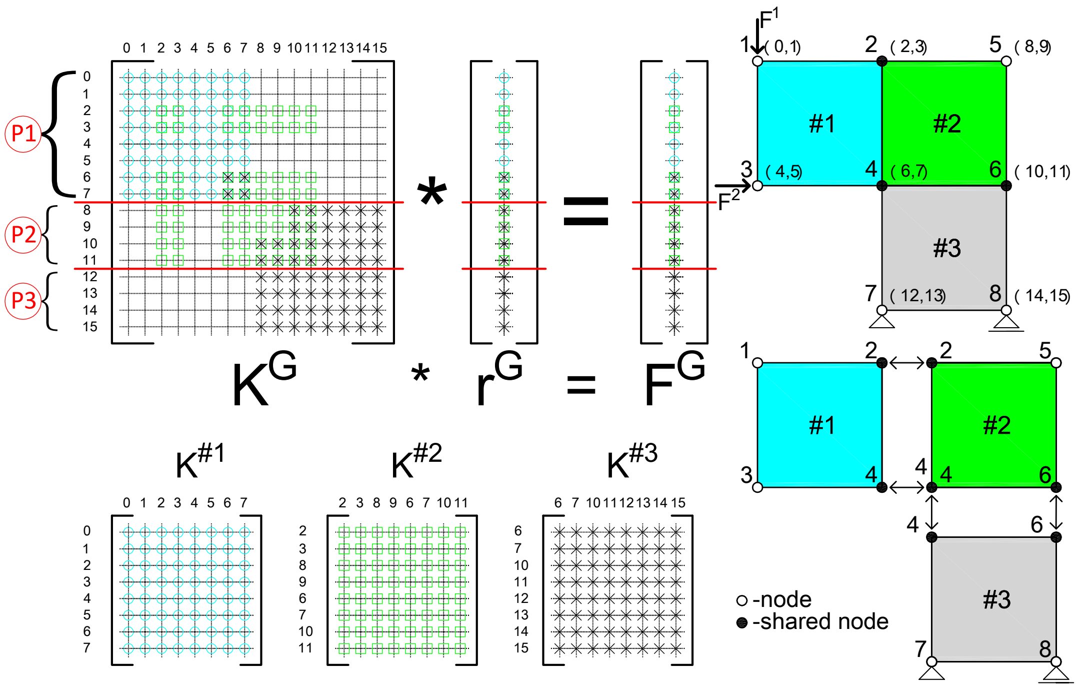


Figure 6.3: Example of a simple problem consisting of three elements distributed into different computing nodes, illustrating the global sparse matrix structure and its distribution.

This sparse matrix storage format can only store the blocks of members of a matrix, and storage format is specified by two arrays represented by the values and positions of these values (Fig. 6.4).

The values of the matrix are stored as a scalar array. or the lower triangular matrix, this array contains the sets of elements from each row the elements from the first nonzero column till the diagonal element. For the upper triangular matrix, it contains the sets of elements from each column starting with the diagonal element ending up with the first nonzero element in a row. Symmetric variant required to store only lower triangular or upper triangular part. An additional one-dimensional array is needed that represents the positions (row or column indices) of the values on the diagonal and has the dimension(e+1), where e is the number of rows.

The Iterative Methods Library (IML) is a collection of algorithms implemented in object-oriented C++ language for solving symmetric and non-symmetric linear systems of equations using iterative techniques. The IML library provides a set of iterative methods: Richardson Iteration, Chebyshev Iteration, Conjugate Gradient, Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilized, Generalized Minimum Residual, and Quasi-Minimal Residual Without Lookahead. All IML iterative solver methods are designed to be used in conjunction with a preconditioner. The preconditioners, namely Diagonal, Incomplete LU and Incomplete Cholesky, are implemented in the IML solver library. In our case, the Conjugate Gradient iterative method with Diagonal preconditioner has been used.

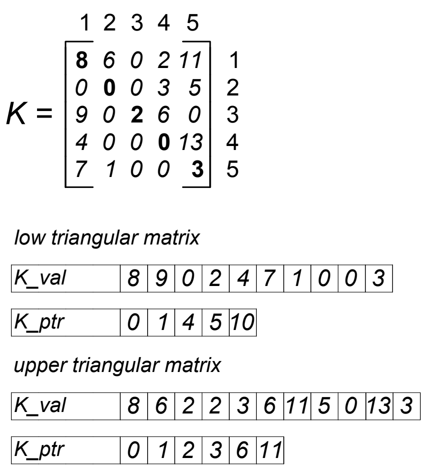


Figure 6.4: The skyline format of matrix schema.

Table 6.1: Sequential linear system solution times for direct (Skyline and SuperLU solvers) and iterative (IML solver) solvers.

Solver type	Library	execution time [s]
Direct Solver	Skyline	2118.09
	SuperLU	372.61
Iterative Solver	IML	180.47

Table 6.2: Sequential linear system solution times for direct (Skyline and SuperLU solvers) and iterative (IML solver) solvers.

In this section, the performance and efficiency of the presented sequential linear equation solver are compared using the benchmark problem of a 3D nuclear containment dome model, already presented as *Jete* in Chapter 5 in section 5.5. The benchmark problem indicated as *Jete250k* consisted of *87 k* nodes and *959 k* tetrahedral elements with linear interpolation, and the total number of equations was *260 k*. The structure was loaded with self-weight. Sequential methods for solving a system of linear equations with a direct solver using Skyline matrix storage, direct SuperLU solver and IML iterative solver are compared in Table 6.1.

The iterative IML solver was two times faster than the direct SuperLU solver in this problem, profiting from the high sparsity of the 3D problem. Nevertheless, applying the direct SuperLU solver to solve a system of linear equations significantly reduced the computational time. The presented conclusions relate to the selected benchmark problem, however. It is therefore important to have different options available in general purpose finite element code.

6.3 SUPERLU SOLVER BASED ON A SHARED MEMORY MODEL

In this section, the multi-threaded SuperLU solver is presented and a linear elasticity problem is considered. The SuperLU solver is used with parallel libraries based on shared memory, represented by OpenMP and POSIX Threads. SuperLU contains a set of sparse direct solvers for solving large sets of linear equations. The kernel algorithm in SuperLU uses sparse Gaussian elimination. The first step of the solution algorithm is triangular factorization. In addition to complete factorization, the SuperLU library also has limited support for an incomplete factorization (ILU) preconditioner, which approximately solves the equation. Permutation matrices can be set up to improve sparsity, numerical stability and parallelism.

Comparing the different method of the direct SuperLU solver used to solve a sparse linear system is the subject of this section. The SuperLU library interface was developed, which consists of an interface class to SuperLU solver and the class representing compress row storage format of SuperLU. The first comparison was was based on using simple and expert SuperLU drivers. The SuperLU expert driver advantageously uses the symmetry of the matrix in a linear system of equations. SuperLU simple is the default option that does not take into account the effect of matrix symmetry. The ordering the equations were determined multiple minimum degrees for both cases.
The SuperLU implementation is based either on OpenMP or POSIX Threads. The performance of shared memorySuperLU solver has been also compared to the performance of iterative PETSc solver using distributed memory paradigm. A parallel PETSc solver based on distributed memory was configured to use a conjugated gradient solver with incomplete Cholesky preconditioner.

Figure 6.5 shows the execution times and speedups for solving a linear system of equations using a direct solver based on a shared memory model (SuperLU simple or expert driver using OpenMP and POSIX threads) and an iterative solver based on distributed memory (PETSc with using MPI) on the benchmark problem of the 3D nuclear containment dome model Jete 250k. The results representing the solution of the system of linear equations are presented as solution times and corresponding speedups (relative to two CPUs). These results were obtained as an average of five consecutive runs.

The results show that the time needed for execution is reduced as the number of threads increased. In general, the direct SuperLU solver variants needed considerably more time to solve the benchmark problem than the PETSc iterative solver. The results from the direct SuperLU solver using the expert driver were better than the results of the direct SuperLU solver using the simple driver. The expert driver can take advantage of the symmetry of the matrix in a linear system, which was reflected in the results. Ordering the columns of the matrix was optimized in the simple and expert drivers using the multiple minimum degrees ordering $A^T + A$. The results of the SuperLU methods using the direct SuperLU solver based on OpenMP or POSIX Threads and shared memory frameworks had a similar speedup effect. Although far from ideal scalability, the results show reasonable linear scalability up to 16 cores. The significant decrease in performance for 16 and more threads, observed in all cases, can be attributed to the hyper-threading technology of Intel processors, which assembles and shares some CPU resources between hyper-threaded cores during the solving process and only takes place when a

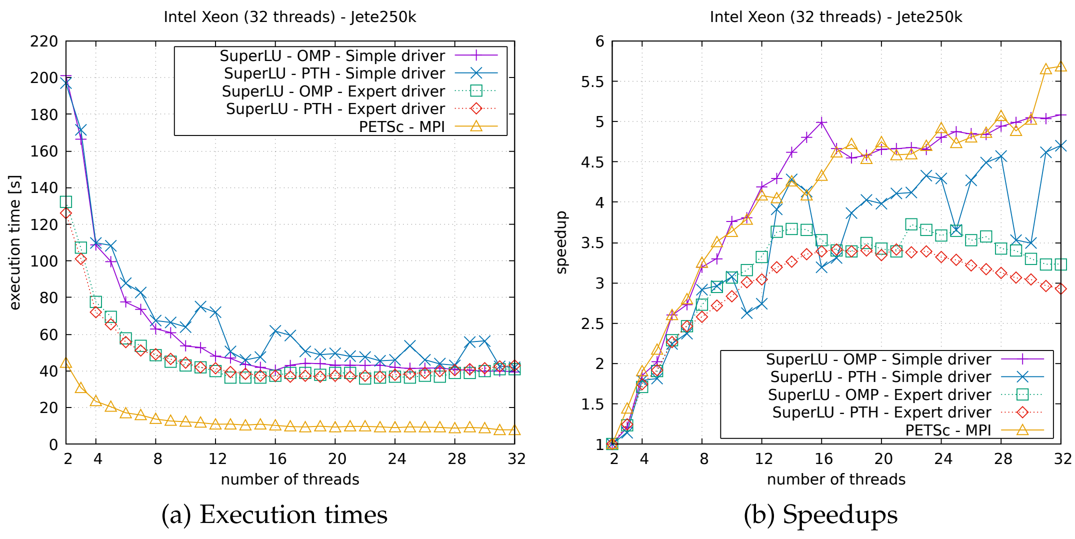


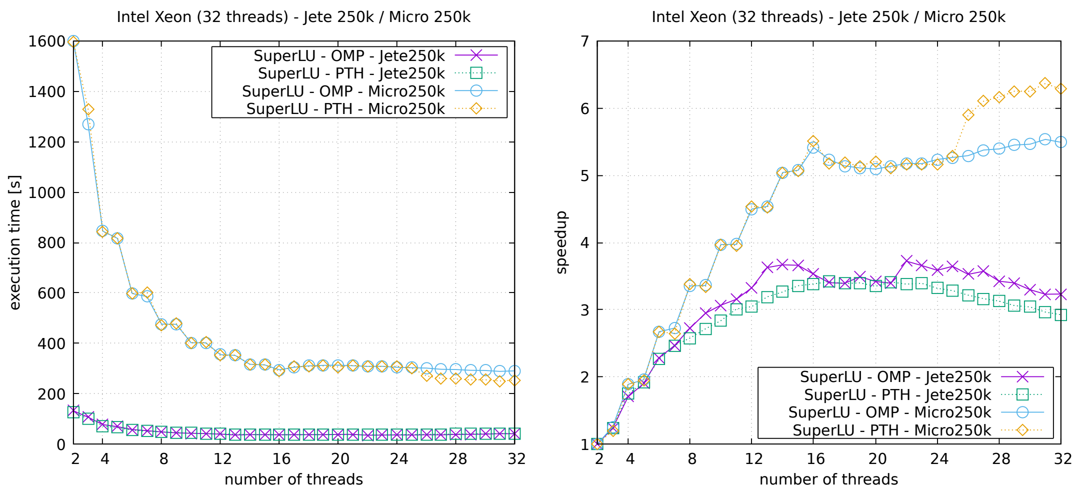
Figure 6.5: Execution times and speedups of a system of linear equations using a direct SuperLU simple driver and direct SuperLU expert driver based on OpenMP or POSIX Threads (shared memory model) and an iterative solver PETSc based on MPI (distributed memory model) on the Jete 250k benchmark problem.

number of threads is higher than the number of physical cores. In general, the results illustrate that regardless of the solver used, the better performance can be achieved that with serial code.

The execution times and speedups to solve a linear system of equations using a direct solver based on shared memory model (SuperLU expert driver using OpenMP and POSIX Threads) on the 3D model benchmark problems of a nuclear containment dome Jete 250k and porous micro structure Micro 250k are presented in Figure 6.6. The reported execution times and speedups were obtained as an average of five consecutive runs.

In general, the direct SuperLU methods on the Micro 250k benchmark problem needed considerably more time for a solution than the Jete 250k benchmark problem. The benchmark problems were characterized by different system matrix sparsity. The porous microstructure model had significantly more nonzero members than the nuclear containment dome model, which is a very sparse problem, and it was clear that the Micro 250k benchmark needed significantly more time to solve the problem. The results of SuperLU methods using the direct SuperLU solver based on OpenMP or POSIX Threads in a shared memory framework had a similar speedup effect. Although far from ideal scalability, the results show reasonable linear scalability up to 16 cores. The significant decrease of performance for 32 threads, observed in all cases, can be attributed to the hyper-threading technology of Intel processors, which is mentioned in the paragraph above. However, by using direct or iterative parallel solvers, we can achieve better performance than serial code.

The similar comparison has been done for other benchmark problems as well. The results show significant differences between the performance of direct and iterative solvers in each of the benchmarks. The PETSc iterative



(a) Execution times (b) Speedups

Figure 6.6: Execution times and speedups of the system of linear equations using direct SuperLU expert driver based on OpenMP or POSIX Threads (shared memory model) on the Jete 250k and Micro 250k benchmark problems.

solver based on distributed memory performed better than the SuperLU direct solver based on shared memory. The direct SuperLU solver's main disadvantage was high demands on system memory. The memory requirements are higher during the solution of the Jete 3M, Micro 500k, Micro 1M and Micro 3M benchmarks. For example, the direct SuperLU solver based on shared memory with expert driver settings (symmetric mode) on the Jete 250k benchmark needed 8 GB of system memory, while the Micro 250k benchmark needed 18 GB. By contrast, the PETSc iterative solver based on distributed memory needed only 1.2 GB of the system memory for Jete 250k (3800 iterations) and 1.3 GB for Micro 250k (14700 iterations). The presented conclusions are related to the selected benchmark problems, however. In general, the option to have both iterative and direct parallel solvers can be an advantage. It is therefore important to have different options available in general purpose finite element code.

6.4 SUPERLU SOLVER BASED ON A DISTRIBUTED MEMORY MODEL

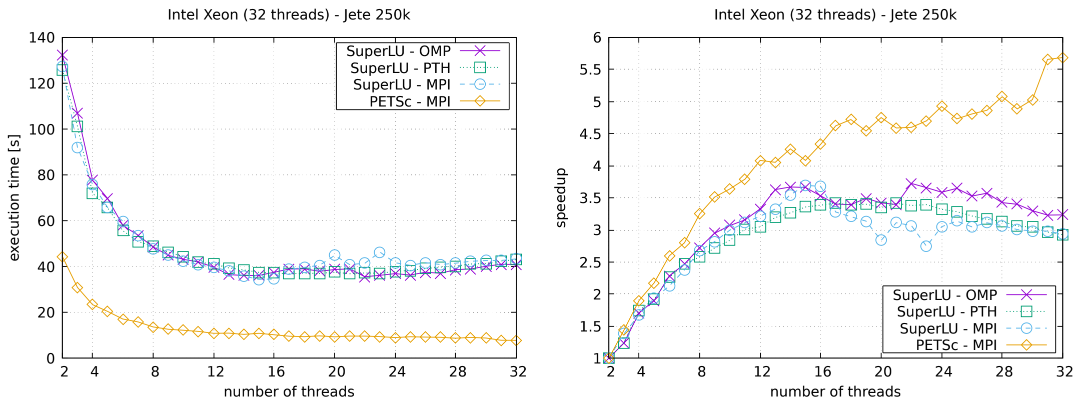
The performance of the distributed memory variant of SuperLU solver is illustrated in this section, for which the interface has been also designed and implemented. The performance evaluation is done on linear and nonlinear structural analysis benchmarks.

6.4.1 *Linear analysis using the SuperLU solver*

The SuperLU library provides a parallel implementation of the direct solver based on a shared memory model (multithreaded SuperLU) or distributed memory. The main differences between these two versions of SuperLU solver are about storing the computational data. In distributed SuperLU interface distributed data structure is required to store the sparse matrix. The decomposition of the sparse matrix into blocks of rows assigned to individual processing nodes is performed. The individual nodes are responsible for assembling the assigned block of rows of the sparse matrix. While in a shared memory model the whole sparse matrix data structures are stored in a global memory accessible to all computing units.

A comparison of direct SuperLU solver based on distributed memory with a distributed sparse matrix stored in compressed row format and the previously mentioned multi-threaded SuperLU solver based on shared memory with a compressed row format is presented in this section. For reference, the performance of PETSc solver is also included.

The efficiency of these solvers is demonstrated on benchmark problems *Jete* and *Micro*. The results are presented as solution times and corresponding speedups (relative to two CPUs). The presented results were obtained as an average of five consecutive runs. The results for the *Jete 250k* benchmark problem are presented in Figure 6.7.



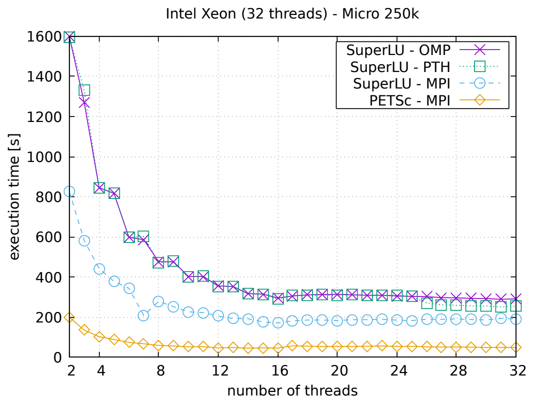
(a) Execution times (b) Speedups

Figure 6.7: Execution times and speedups of the system of linear equations using the direct SuperLU expert driver based on OpenMP or POSIX Threads (shared memory model), direct SuperLU based on MPI (distributed memory model) and iterative PETSc solver based on MPI (distributed memory model) on the Jete 250k benchmark problem.

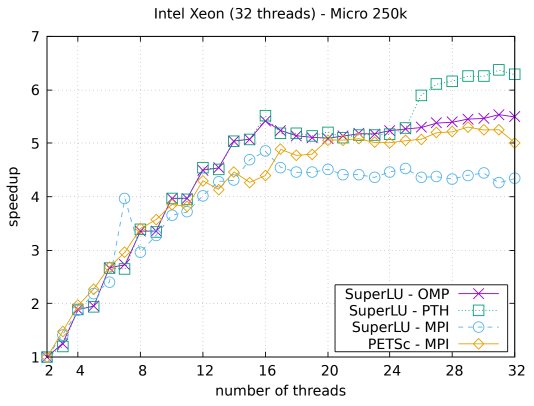
These results show that the SuperLU method based on a shared memory model (OpenMP and POSIX Threads) and SuperLU method based on a distributed memory model (MPI) are slightly similar. The direct SuperLU strategies need considerably more time to solve the *Jete 250k* problem than the PETSc iterative solver based on distributed memory (MPI). Speedups show the same trend of scalability as the presented execution times.

The results of the Micro 250k benchmark problem are shown in Figure 6.6.

The results of the solution process using the SuperLU method based on a shared memory model (OpenMP and POSIX Threads) and SuperLU method based on a distributed memory model (MPI) demonstrate that the distributed memory SuperLU is clearly better than the shared memory SuperLU on the



(a) Execution times



(b) Speedups

Figure 6.8: Execution times and speedups of a system of linear equations using the direct SuperLU expert driver based on OpenMP or POSIX Threads (shared memory model), direct SuperLU based on MPI (distributed memory model) and iterative PETSc solver based on MPI (distributed memory model) on the Micro 250k benchmark problem.

Micro 250k benchmark problem. This is because the Micro benchmark has less sparsity than the Jete benchmark. The results show that the distributed SuperLU solver is more effective in solving a linear system of equations with less sparsity than the multi-threaded SuperLU solver. However, the PETSc iterative solver based on distributed memory is clearly more effective in the Micro 250k benchmark problem than direct solver SuperLU based on distributed memory.

6.4.2 *Nonlinear analysis using the SuperLU solver*

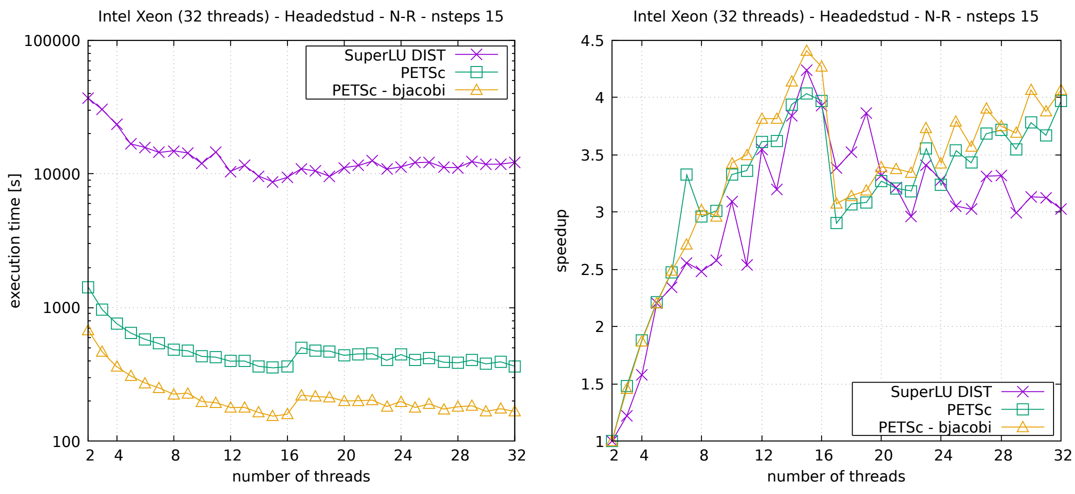
The mentioned nonlinear solution strategies in subsection 4.3.2 were evaluated on the nonlinear 3D crack propagation finite element model of an anchor pullout test using two different sparse linear solvers: the direct SuperLU solver and the iterative PETSc solver, with a distributed memory programming model. The model used non-local anisotropic damage constitutive model and consists of 22441 nodes and 125400 linear tetrahedra elements. The evaluated nonlinear solution strategies included the Newton–Raphson method with a stiffness matrix update after each iteration, Modified Newton–Raphson method with a stiffness matrix updated after every second and tenth iteration, and the initial stiffness method. The total number of iterations in the testing benchmark problem for all of these strategies is presented in Table 6.3. It is clear that the number of iterations needed to successfully solve the benchmark problem is higher for the initial stiffness matrix method than the Newton–Raphson method. The number of iterations in order to successfully solve one loading step ranges from 2 (Newton–Raphson method) to 30 (initial stiffness method) iterations as a part of the solution process.

The execution times and speedups illustrate the performance of the direct SuperLU solver (based on shared or distributed memory version) and PETSc

Solution method	**Num. of iterations**	**Matrix update**
N-R	204	241
mod. N-R s. 2	251	131
mod. N-R s. 10	297	37
initial stiffness	6049	1

Table 6.3: Number of iterations for different methods on the 3D finite element model of an anchor pullout test (number of steps 15).

iterative solver. In the case of the PETSc iterative conjugated gradient solver, convergence criteria based on a relative solution error of 10^{-6} was used. Performance was evaluated for cases without preconditioning and for cases with block Jacobi parallel preconditioning. The solution times (averaged over five consecutive runs) and corresponding speedups (relative to two CPUs) are shown in Figures 6.9, 6.10 and 6.11.

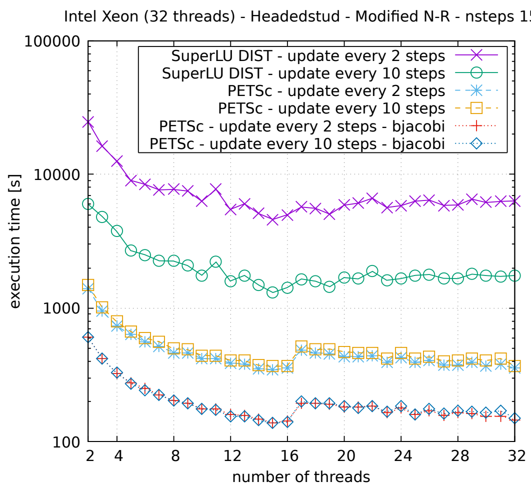


(a) Execution times (b) Speedups

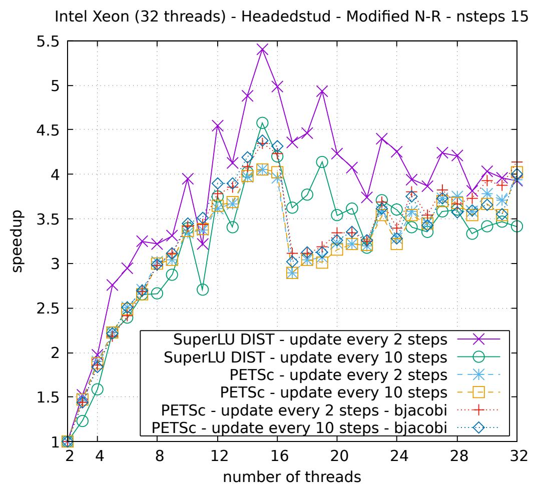
Figure 6.9: Execution times and speedups using the Newton–Raphson method with secant stiffness updated after every iteration.

The achieved results show that the effect of parallelization using both solvers is substantial. In the full Newton–Raphson method, where stiffness is updated after each iteration, performance of the direct solver was less than the performance of the iterative solvers. This is because of the dense character of the stiffness matrix the iterative solver can profit from. The direct solver could not profit from existing factorization, as the system matrix was updated after each iteration. The performance of the preconditioned iterative solver was better than the performance of the non-preconditioned solver (Fig. 6.9).

A similar trend can be seen using the Modified Newton–Raphson method. Again, the performance of the iterative solver was superior to the performance



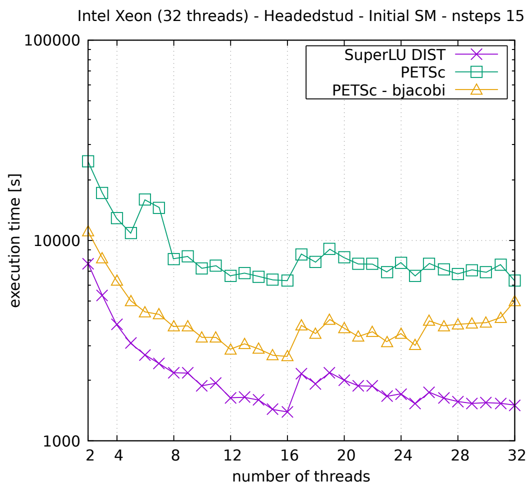
(a) Execution times



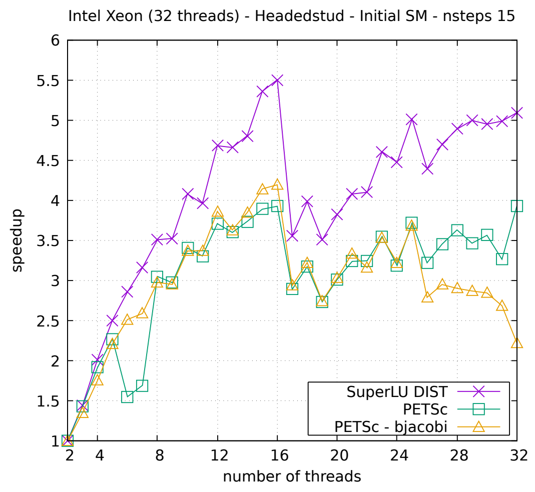
(b) Speedups

Figure 6.10: Execution times and speedups using the Modified Newton–Raphson method with secant stiffness updated after every second and tenth iteration.

of the direct solver (see Fig. 6.10, for a comparison of solution times for both solvers with stiffness updated every second and tenth iteration).



(a) Execution times



(b) Speedups

Figure 6.11: Execution times and speedups using the Initial Stiffness Matrix method.

Finally, the results from the Initial Stiffness Matrix method using direct and iterative solvers are presented. In this case, the best performance was obtained using a direct solver, which profited highly from the existing factorization.

It is difficult to generalize the results, as the relative performance of direct and iterative solvers depends highly on the particular problem. However, the scalability of both solvers can be evaluated. The results for different settings are presented in Figs. 9, 10 and 11. Although far from ideal scalability, the results show reasonable linear scalability up to 16 cores. The significant decrease of performance with 32 threads, which was observed in all cases, can be attributed to the hyper-threading technology of Intel processors.

6.5 CONCLUSIONS

The performance of sequential SuperLU solver is compared to Skyline solver and iterative IML solver in an object-oriented FEM framework was in section 6.2. Further, the performance of a direct SuperLUand PETSc iterative solver based on a shared and distributed system memory model applied to linear static analysis was evaluated in section 6.3, 6.4 and performance of SuperLU solver based on distributed memory model for nonlinear static analysis was evaluated in section6. 6.4.2. The parallelization strategies were based on shared memory libraries (OpenMP and POSIX Threads) and on message passing for distributed memory architecture. The performance of direct and iterative solvers was compared on linear benchmark problems with different sparsity (Jete-3D model of nuclear containment and micro-3D model of porous microstructure) and on complex nonlinear benchmark problem of 3D simulation of an anchor pullout test. In the sequential solution case, the performance of iterative IML solver is the best. On the other hand, the direct SuperLU solver is significantly better than direct skyline solver. The performance of parallel solvers based on shared memory model showed that the iterative solver PETSc has better performance than direct SuperLU solver. SuperLU solver based on OpenMP or POSIX Threads achieved very similar scalability, In the nonlinear static for solvers based on distributing memory model the performance of iterative solver was superior, except when the Initial Stiffness method was used, where the direct solver showed better performance. In general, both parallel solvers based on different memory models showed relatively good scalability. In terms of the relative performance of individual solvers for different use cases and problems, it is definitely an advantage to implement both solver types in any Finite Element code.

7 TUNING FINITE ELEMENT LOAD BALANCING FRAMEWORKS

The chapter deals with tuning the parallel load balancing framework of the finite element software, which is based on a domain decomposition paradigm for a distributed memory model. This chapter introduces the domain decomposition paradigm and describes the technique for determining actual weights by comparing the computational performance of individual processing units. These weights are fundamental inputs for mesh (re)partitioning that has to be performed at the beginning of or during a simulation and whenever the load imbalance is significant. The capabilities and performance of the proposed technique are evaluated on a benchmark problem and discussed.

7.1 LOAD BALANCING ALGORITHM

Load balancing is a process of (re) redistributing work between the involved computational units. In this section, the distributed memory programming model is assumed. It requires the decomposition of the FE problem into subdomains assigned to individual processing units. The domain decomposition is based on node-cut technique leading to unique assignment of elements and duplication of shared nodes. The decomposition is in general complex optimization problem aiming to balance the work while satisfying a number of conditions, including the requirement for minimum communication between partitions, for example. In this work, the ParMETIS library is used for decomposition [36]. The library allows to input performance measure of individual nodes to reflect different CPU speeds, for example. On the other hand, other requirements, such as mentioned minimum communication cost, could not be controlled and are implicitly encoded in the library. The load balancing can be either static during the solution process or dynamically adapted.

Determining the processor's weight parameters is, therefore, an interesting subject. The contribution of this work consists in proposing a better algorithm to evaluate individual processor weights that the existing one in OOFEM solver.

Partitioning models must more accurately represent a broader range of operations. Partitioning algorithms also need to be sensitive to heterogeneous computer architectures and adjust work assignments relative to processing, memory and communication resources. However, the multi-criteria partitioning is the subject of ongoing research and the existing partitioning libraries have only limited capabilities. Therefore, in this thesis, the optimization of dynamic load partitioning algorithm is based on variables that are related only to the performance of the individual nodes. The parallel computer may be composed of different nodes that possess processors with different performance characteristics and be connected to a network for communication between these processors. The performance of the individual processor, therefore, has to be detected and this can be done using several simple tests. The individual tests are executed each time the load balancing occurs to determine actual performance. The individual tests should represent typical FEM operations and at the same time, they should have a small impact, consuming only a small fraction of the total solution time of the problem. The performance is measured as a wall clock time needed to complete individual tests. Overall performance is evaluated as a weighted average from individual tests. Processor performance depends on many factors, for example, cache memory. In symmetric multiprocessor (SMP) systems, each processor has a local cache. The processing unit has several layers of cache memory between the processors and main memory (RAM). The cache layers are obviously designed in two or three layers but are smaller and faster than RAM. Therefore, the weight of the processor parameter not only includes processor performance with regard to the inseparable part of the processor (cache memory) but also the influence of the other system component (RAM). However, in our case, the performance of the processor based on measuring wall clock time is sufficient.

7.2 DESIGN AND ASPECTS OF MICRO BENCHMARKS TESTS

Overall processor performance depends on many factors, notably on its frequency, the performance of memory subsystem, and code executed. The performance of two CPUs can be different for integer and floating point operations, on some SMPs, some resources are shared between processing cores, etc. The adopted approach to evaluate the processor performance is based on a set of so-called micro benchmark tests, that evaluate processor performance for different typical tasks and computing overall performance as the weighted average of performances of individual micro-benchmarks.

The first micro benchmark test proposed is a function to compute a numerical integral. In the test, the integral of *cos* function is evaluated using numerical integration using Newton-Cortes formula. The parameter n_i defines the number of integration points used to divide the area. Therefore, with increasing parameter n_i (increasing number of integration point), we have a solution process closely approximating the integral value. This tests measures performance in floating point operations.
This micro benchmark test was conducted using different sets of the parameter n (number of integration points used to divide the area). The performance of individual processor is defined as relative execution time of the processor to complete the test $\frac{p_i=t_i}{\sum t_i}$, where t_i is execution test time on i-th processor. The subject of the investigation was the dependence of the number of integration points on the estimated performance. The results of processor weights with different sets of parameter n_i using different numbers of computational threads are presented in Table 7.1. The individual variants were tested on a workstation (running Ubuntu 16.04 OS) with an Intel® Core™ i7-4790 CPU @ 3.60 GHz (*PC*#2) with four cores consisting of two logical processors per

n_i	2 thr.	4 threads		6 threads			8 threads				
10^4	0.2175	0.1324	0.1316	0.0797	0.0950	0.0812	0.0653	0.0573	0.0704	0.0644	*PC*#1
	0.7825	0.3679	0.3681	0.2507	0.2459	0.2471	0.1839	0.1843	0.1876	0.1871	*PC*#2
10^6	0.2330	0.1725	0.1354	0.0934	0.0903	0.0757	0.0672	0.0660	0.0658	0.0628	*PC*#1
	0.7670	0.3642	0.3646	0.2394	0.2393	0.2392	0.1863	0.1863	0.1829	0.1828	*PC*#2
10^8	0.3653	0.1831	0.1835	0.1088	0.1081	0.1085	0.0751	0.0762	0.0758	0.0752	*PC*#1
	0.6347	0.3189	0.3139	0.2249	0.2249	0.2249	0.1745	0.1745	0.1744	0.1744	*PC*#2

Table 7.1: Processors weights for the micro benchmark problem represented as a computed numerical integral using the $cos(x)$ function, different numbers of rectangles to divide the area (parameter n_i) and different numbers of computational threads.

core connected to a workstation with an Intel® Core™ i3-2370M CPU @ 2.40 GHz (PC#1) with two cores consisting of two logical cores. A maximum of eight threads on machine PC#2 and four threads on machine PC#1 could run simultaneously. The testing machines had 15 GB and 8 GB of system memory, respectively. The reported results were obtained as an average of five consecutive runs. The run time to compute the numerical integral was based on two computational threads. The run times on threads on PC#1, which were the longest (greater influence on computational time by the benchmark problem), for n_i set to 10^4 was $0.44s$, for n_i set to 10^6 was $0.51s$ and for n_i set to 10^8 was $3.85s$. These run times were substantially quicker than the computation time of our benchmark problem using structural analysis based on nonlinear static analysis. In the case when the solution of the micro benchmark problem is not expensive, it is ideal, because the micro benchmark can be run more times during nonlinear analysis of the benchmark problem, and the micro benchmark computational process does not rapidly increase the solution time of the analysis.

The next type of micro benchmark problem was based on solving a linear system of equations. A FEM model of the cantilever beam, divided into n_{eq} elements is used to define a linear system. The right hand side of the force vector in our case is a simple vector with its first member set to 1, while all the other elements of the vector are set to zero. The number of equations of the linear system directly depends on parameter n_{eq}, which also represents the dimension of the stiffness matrix and right hand side of the force vector. The results of processor weights (solving a linear system of equation micro benchmark) in individual cases for parameter n_{eq} using different numbers of computational threads are presented in Table 7.2. The individual methods were tested on the same workstations mentioned above. The reported results were obtained as an average of five consecutive runs. The run time to compute the solution of the linear system of equations using a direct solver based on two computational threads for n_{eq} set to 10^2 was $4.92 * 10^{-4}s$ on PC#1 and $2.56 * 10^{-4}s$ on PC#2. For parameter n_{eq} set to 10^3, the run times were $5.58s$ on PC#1 and $2.33s$ on PC#2. For parameter n_{eq} set to $2.5 * 10^3$, the run times were $126.3s$ on PC#1 and $78.85s$ on PC#2. For parameter n_{eq} set to $5.0 * 10^3$, the run times were $1261.75s$ on PC#1 and $748.11s$ on PC#2. In the case when the

n_{eq}	2 thr.	4 threads		6 threads			8 threads				
10^2	0.3422	0.1610	0.1375	0.0774	0.0813	0.0893	0.0575	0.0533	0.0580	0.0576	*PC*#1
	0.6578	0.3437	0.3577	0.2509	0.2505	0.2506	0.1983	0.1894	0.1965	0.1895	*PC*#2
10^3	0.2955	0.1501	0.1489	0.0683	0.0695	0.1098	0.0473	0.0456	0.0459	0.0462	*PC*#1
	0.7045	0.3506	0.3506	0.2514	0.2491	0.2519	0.2037	0.2039	0.2038	0.2038	*PC*#2
2500	0.3752	0.1783	0.1787	0.0862	0.0912	0.0850	0.0541	0.0535	0.0537	0.0541	*PC*#1
	0.6248	0.3211	0.3220	0.2457	0.2456	0.2463	0.1959	0.1964	0.1962	0.1961	*PC*#2
5000	0.3723	0.1735	0.1751	0.0762	0.0771	0.0763	0.0335	0.0336	0.0343	0.0349	*PC*#1
	0.6277	0.3249	0.3266	0.2558	0.2573	0.2574	0.2164	0.2152	0.2168	0.2154	*PC*#2

Table 7.2: Processors weights for the micro benchmark problem represented as a solution of a linear system of equations using different numbers of equations (parameter n_{eq}) and computational threads.

number of equations (parameter n_{eq}) increased, the solution time of the linear system of equations using a direct solver with dense matrix was characterized by an n^3 increase in time requirements. The rules for computational costs caused the solution of this micro benchmark problem to have the same trend as the function to compute the numerical integral mentioned above.

Another micro benchmark problem called Whetstones is a test that attempts to measure the speed and efficiency of a computer performing floating-point operations. The Whetstone benchmark was the first designed for benchmarking [15]. This benchmark is very simple, comprising several sub-tests with active loops executed via procedure calls. This module represents a mix of operations typically performed in scientific applications. The test involves integer arithmetic, floating point arithmetic, "if" statements, calls and so on. At the end of this benchmark, a statement with the results is printed. Weights were attached to the different modules (realized as loop bounds for loops around the individual modules statements). The weight distribution of Whetstone instructions for the benchmark matched the distribution seen in the program sample. The weights were selected so that the program executed a multiple of one million Whetstone instructions. The tests are not expensive to hardware and only reference a small amount of data that fits into the $L1$ cache of any processor. Hence, $L2$ cache and memory speed should have no effect on performance ratings. Speed is invariably proportional to the processor unit's MHz on any given type of processor.

In general, the Whetstone benchmarks are represented by elementary mathematical operations. The modules provide a set of transformation statements, trigonometric function evaluation, evaluations single transformations of the standard square root function, conditional jumps with sets of conditional statements, integer arithmetic and array addressing, array addressing and references. The benchmark has high floating point data and floating point operations performance. A high percentage of execution time is spent in mathematical library functions. Instead of local variables, Whetstone uses a handful of global data (several scalar variables and four element arrays of constant size) repeatedly. A compiler in which the most heavily used global variables are allocated to registers (an optimization usually considered of secondary

n_l	2 thr.	4 threads		6 threads			8 threads				
6*10^4	0.3197	0.1572	0.1548	0.0834	0.0953	0.0970	0.0539	0.0573	0.0626	0.0645	*PC*#1
	0.6803	0.3440	0.3440	0.2414	0.2414	0.2414	0.1905	0.1904	0.1904	0.1904	*PC*#2
10^5	0.3196	0.1586	0.1588	0.1081	0.0827	0.0840	0.0614	0.0568	0.0583	0.0671	*PC*#1
	0.6804	0.3470	0.3357	0.2418	0.2417	0.2417	0.1893	0.1893	0.1886	0.1893	*PC*#2
2*10^5	0.3135	0.1606	0.1593	0.0895	0.0934	0.0999	0.0641	0.0629	0.0638	0.0664	*PC*#1
	0.6865	0.3401	0.3400	0.2391	0.2391	0.2390	0.1857	0.1856	0.1857	0.1857	*PC*#2
10^6	0.3354	0.1649	0.1637	0.0970	0.0969	0.1094	0.0631	0.0654	0.0653	0.0644	*PC*#1
	0.6646	0.3358	0.3357	0.2332	0.2332	0.2303	0.1858	0.1859	0.1846	0.1855	*PC*#2

Table 7.3: Processors weights for the micro benchmark problem represented as a solution of mathematical floating-point operations in a Whetstone benchmark test using different numbers of instructions (parameter n_l) and computational threads.

importance) would, therefore, boost benchmark performance. Each module was represented as a Whetstone instruction, including the coding containing the for a loop. The execution frequency of each module was proportional to the input value of such that the scaling factors for $n_l = 10$ gave the modules a total weight corresponding to one million Whetstone instructions for more details see [23].

The final processor weights for individual cases dependent on the n_l parameter and using different numbers of computational threads are presented in Table 7.3. The individual methods were tested on the same workstations mentioned above. The reported results were obtained as an average of five consecutive runs. The run time to compute the solution of a linear system of equations using a direct solver based on two computational threads for n_l set to $6 * 10^4$ was $1.07s$ on PC#1 and $0.50s$ on PC#2. For n_l set to 10^5, run times were $1.78s$ on PC#1 and $0.84s$ on PC#2. For n_l set to $2 * 10^5$, run times were $3.57s$ on PC#1 and $1.62s$ on PC#2. Finally, for n_l set to 10^6, run times were $39.10s$ on PC#1 and $19.67s$ on PC#2.

This section evaluated the performance of three micro benchmark problems to determine processor weights with an illustration of their impact on the performance of the load balancing. The final load balancing processor weighs are obtained as a weighted average of individual weights from micro benchmark tests. The final weights are assembled in the ratio of 15% micro benchmark problem based on a function to compute numerical integral, 80% benchmark problem based on the solution of a linear system of equations and 5% benchmark problem based on measuring the speed and efficiency of a computer performing floating point operations. The time requirements of individual micro benchmark problems were controlled using parameter n as follows: benchmark to compute numerical integral with parameter $n_i = 10^6$, solution of a linear system of equations with parameter $n_{eq} = 10^3$ and computer performing floating-point operations with parameter $n_l = 6 * 10^4$.

7.3 STATIC LOAD BALANCING FRAMEWORK

This section presents the design process using processor weights parameters and results of a case with a static load balancing framework based on partitioning the benchmark problem before it is solved. Static load balancing can be effectively used in a linear static case, which is solved in one step. However, in the nonlinear static case, the static load balancing framework is not effective.

7.3.1 *Example using a Static load balancing framework*

The individual methods were evaluated using the benchmark problem of a 3D finite element of a nuclear containment dome model. The Jete250k benchmark problem consisted of 87 k nodes and 959 k tetrahedral elements with linear interpolation, and the total number of equations was 260 k. The structure was loaded with self-weight.

The individual methods were tested on a workstation (running Ubuntu 16.04 OS) with an Intel® Core™ i7-4790 CPU @ 3.60 GHz with four cores consisting of two logical processors per core connected to a workstation with an Intel® Core™ i3-2370M CPU @ 2.40 GHz with two cores consisting of two logical cores. Each workstation could simultaneously run a maximum of eight threads and four threads on their CPUs and had 15 GB and 8 GB of system memory, respectively.

The iterative PETSc solver library for solving a system of equations (linearized system solution) and ParMETIS as an MPI parallel library for partitioning problems was used for the benchmark problem. The results of a linear static problem based on static load balancing partitioning using processor weights input parameters are presented in Figure 7.1 and were compared to the results obtained with static load balancing using equal processing weights. The performance of static load balancing with equal weights, as a default setting of the static load balancing, is compared to the static load balancing with estimated weights using micro-benchmark problems. The number of threads represents a combination of two workstations. For example, two computational threads are represented as one computational thread running on each workstation. The results show that the effect of parallelization using a static load balancing process with improved processor weight parameters makes the solution process more efficient and takes less computational time.

7.4 DYNAMIC LOAD BALANCING FRAMEWORK

The software design process is an important part of dynamic load-balancing research. The design process and results of a case with a dynamic load balancing framework based on a problem partitioned during the solution process are discussed in this section. Dynamic load balancing is typically used for problems involving multiple solution or time steps, where work redistribution is performed to reflect potential work imbalance. Partitioning the problem during solution is based on monitoring computation during the solution step.

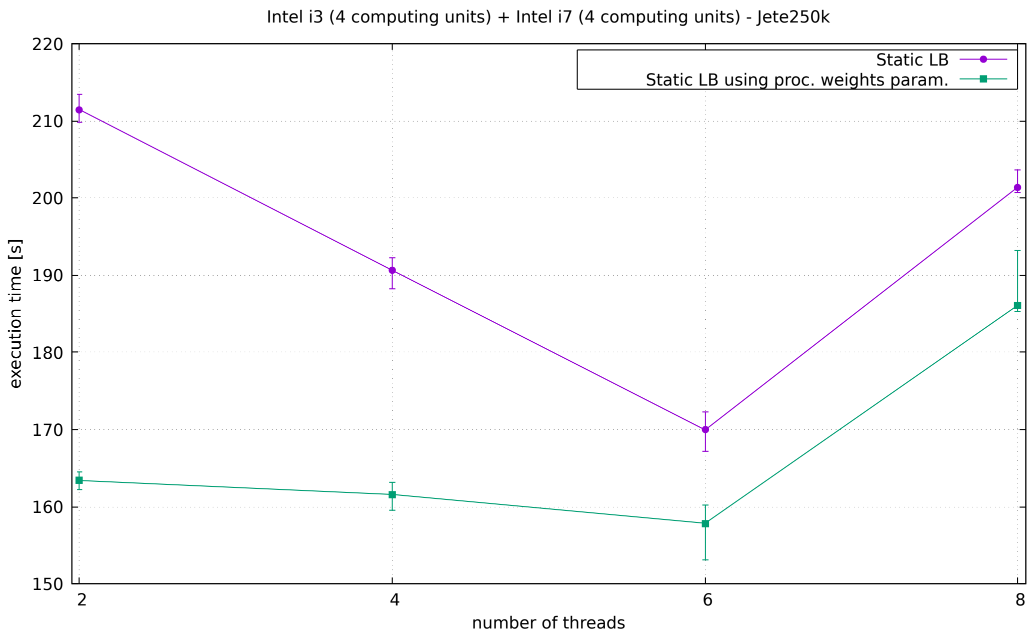


Figure 7.1: Execution times for static load balancing using estimated processing weights compared to uniform processing weights.

Once the solution step is finished, the load balance process is activated by evaluating the imbalance. If the imbalance is larger than the user defined tolerance, the re-partitioning is performed, taking into account the actual performance of processing nodes and processing cost of individual elements. The partitioning is performed in respect of the load balance monitoring results. When solving a real problem on a parallel computer, the load balance can change during the solution. The first source of imbalance originates from the character of the problem. For example, in nonlinear problems, the transition from an initial elastic material response to a nonlinear regime is often associated with increased computational cost, and this transition is often associated only to certain regions of the overall domain. The second source of load imbalance includes external factors that can change the performance of individual processing nodes or communication networks. This typically happens in non-dedicated cluster environments, where processing nodes and communication infrastructure is shared between users. In both cases, the gradual growth of imbalance can have a significant effect on performance and scalability.

The only way of reflecting the growing imbalance is to adaptively redistribute work between processing units in order to restore load balance and thereby ensure optimal use of resources. The monitoring and evaluation of imbalance is the task of load balance monitor. This monitor can detect load imbalance by monitoring the time required to perform allocated work on individual processing units. The differences in processing time indicate an imbalance. After imbalance is detected, the decision of whether to restore the load balance or continue is made. This can be a complex task, as load redistribution may in fact be a very complex problem with non-negligible time requirements. The cost of load re-balancing may be higher than the cost of con-

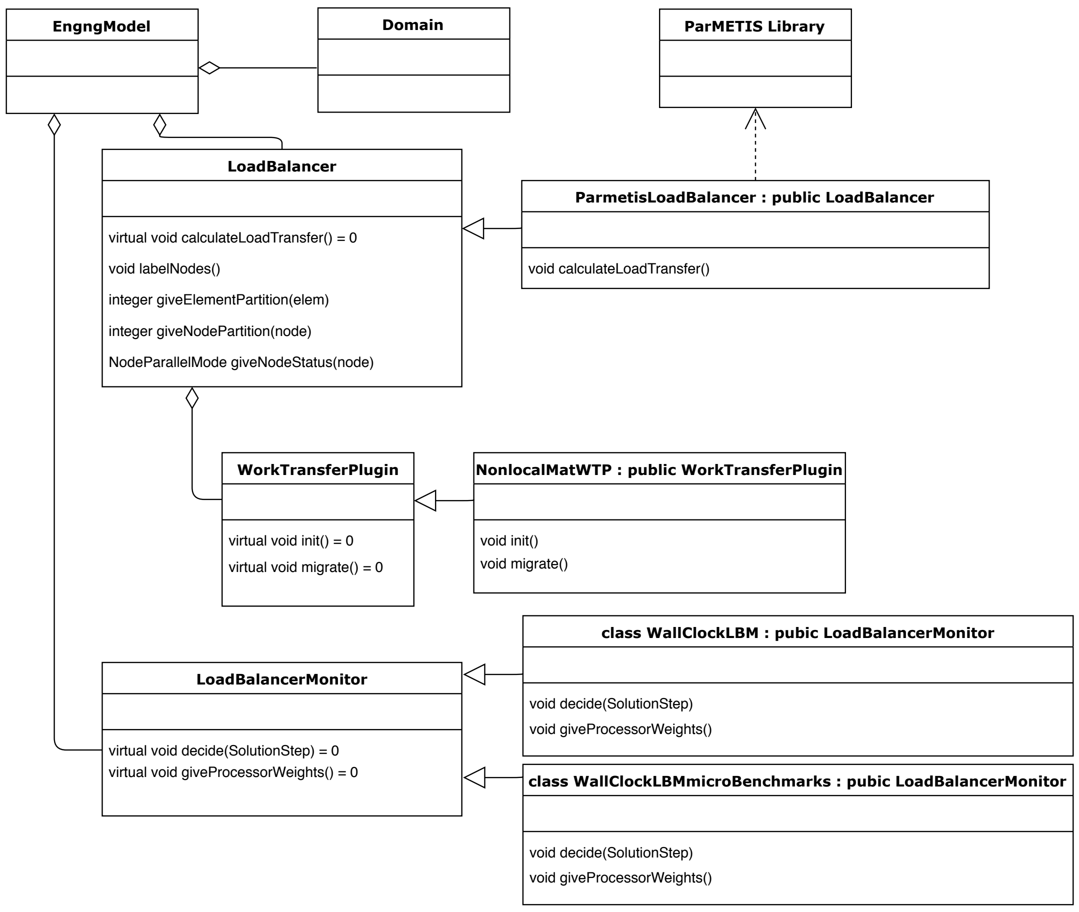

EngngModel
Domain
ParMETIS Library
LoadBalancer
virtual void calculateLoadTransfer() = 0
void labelNodes()
integer giveElementPartition(elem)
integer giveNodePartition(node)
NodeParallelMode giveNodeStatus(node)
ParmetisLoadBalancer : public LoadBalancer
void calculateLoadTransfer()
WorkTransferPlugin
virtual void init() = 0
virtual void migrate() = 0
NonlocalMatWTP : public WorkTransferPlugin
void init()
void migrate()
class WallClockLBM : pubic LoadBalancerMonitor
void decide(SolutionStep)
void giveProcessorWeights()
LoadBalancerMonitor
virtual void decide(SolutionStep) = 0
virtual void giveProcessorWeights() = 0
class WallClockLBMmicroBenchmarks : pubic LoadBalancerMonitor
void decide(SolutionStep)
void giveProcessorWeights()

Figure 7.2: Class hierarchy related to the load balancing framework.

tinuing with a slight load-imbalance. All these aspects have to be considered and are, unfortunately, specific to the problem and implementation.

The design and structure of load balancing framework in OOFEM is illustrated in Figure 7.2. The Load balance monitor class is responsible for detecting potential load imbalance between participating processors. The main role of the load balancing monitor is to determine the amount of work that should be ideally transferred from one processing unit to another in order to recover the detected imbalance. Load imbalance is monitored by measuring the computational time required to process assigned work (process given partition). In an ideal case, the individual times should be equal, indicating that the work load is ideally balanced between processes that are solved on different computational units with different performance parameters. When the imbalance is detected, the work (expressed in terms of individual elements) has to be redistributed to reflect the performance of individual processing nodes. Individual elements have assigned weights that reflect the relative computational costs. These depend on element type, material state (elastic, plastic, etc.). These weights are typically obtained in advance from estimated measurements. The processing power of individual processing units also has to be determined. The load re-balancing process in this chapter is based on re-distributing the computational work proportionally to the performance of individual processing units. In principle, additional factors, including available communication bandwidth between individual processing units or available memory, can be taken into account. Any load balancing should not only distribute the work according to processing powers but also attempt to minimize the communication cost between individual processing units. In the FEM, this means the cuts between partitions (number of shared nodes) should be minimized. Failing to meet the secondary criteria can significantly impact overall performance, as the cost of communication (in terms of the time required) is much higher than the cost of computation. Load re-balancing should also attempt to minimize the reallocation of elements as much as possible in order to minimize communication costs.

The ParMETIS library was used in this work, however, the ParMETIS implementation is not restricted to one specific partitioning library. A parallel partitioner can take advantage of the increased memory capacity of parallel machines (distributed memory model) and improve overall performance. A general load balancing algorithm is responsible for dynamic repartitioning using the processor weights provided by the load balance monitor during the micro benchmark problem and should provide the new partitions with numbers for all local elements on each partition. After the updated element partition assignment is determined, the distribution and classification of nodes also have to be determined. Each node is classified as either a local node that remains local on an existing partition or a shared node that is assigned to a remote partition. Node classification can be determined from element partitioning. The dynamic load balancing framework implemented in OOFEM is using the weights based on measuring of individual time involved in the computation. These times represent how long each CPU work on the predefined number of solution steps. The repartitioning is based on default weights set as the ratio of total solution time to individual computational thread time.

7.4.1 *Example using a dynamic load balancing framework*

The above-mentioned nonlinear solution strategies for reducing imbalance using dynamic load balancing methods were evaluated on a 3D finite element model of an anchor pullout test. The FE model involved nonlinear nonlocal anisotropic damage model to describe fracture process and consists of 1456 nodes and 16772 tetrahedral elements, which was subsequently refined in 20 steps into a final mesh with 125400 elements and 22441 nodes. The number of solution steps was set to 20 and 80 steps. Due to the character of the solution, the number of required equilibrium iterations was increasing with progressing solution steps. The number of iterations in one loading step ranged from 9 (first solution step) to 263 (solution step number 80) iterations in order to successfully solve the loading steps in the solution process.

The problem was solved in parallel on three workstations with different performances. This setup was deliberately selected in order to demonstrate the role of using appropriate processing weights. Two of the workstations were identical to those mentioned in section 7.3.1, and the third had an Intel® Core™ i9-9000K CPU @ 3.60 GHz with eight cores consisting of two logical processors per core. On these workstations, a maximum of sixteen threads (Intel i9), eight threads (Intel i7) and four threads (Intel i3) could be run simultaneously with hyper-threading support. To disable the Intel hyper-threading technology, our computations had to be set to a maximum of 14 computational threads, which represents the number of cores available on these computers.

The iterative linear equation solver from the PETSc library was used with a block Jacobi preconditioner. The dynamic load balancing framework parameters were set up. Load re-balancing was activated after each fifth solution step. The relative wall clock imbalance parameter represents the relative im-balance between wall clock solution time of individual computational threads. When greater than the provided threshold the rebalancing procedure is activated. The additional parameter called absolute wall clock imbalance parameter allows to triggered rebalancing procedure when an achieved absolute imbalance between solution times of individual processing threads is greater than the threshold. The absolute wall clock imbalance parameter was set to 10.0 $[s]$. The minimum absolute wall clock imbalance parameter allows setting minimum absolute wall clock imbalance threshold for performing rebalancing and was set to 0.5 $[s]$.

The first set of results is presented in Figures 7.3, 7.4 and 7.5, from a workstation with an Intel® Core™ i7-4790 CPU @ 3.60 GHz connected to a workstation with an Intel® Core™ i3-2370M CPU @ 2.40 GHz. The results obtained using the existing dynamic load balancing framework without the micro benchmark tests to set up processor weights were compared to the results of the computation process without load balancing. These load balancing strategies were finally compared to the method using a load balancing framework in order to better illustrate load balancing frameworks in general. Figures 7.3 and 7.4 show examples of computation done by processors with hyper-threading technology disabled (Intel i3 max. 2 threads, Intel i7 max. 4 threads). The computational process with hyper-threading technology enabled on the i3 processor is shown in Figure 7.5.

The results confirm that better performance is obtained when appropriate weights are used. In the case with hyper-threading technology enabled, the scalability trend is not ideal in the solution based on six and eight computational threads and is worse, for example, than the solution based on two and four computational threads. The significant decrease in performance can be attributed to the hyper-threading technology of Intel processors, which assembles and shares some CPU resources between hyper-threaded cores and only takes place with 4 threads (note that the workstation with Intel® Core™ i3 had two physical hyper-threaded cores). However, using dynamic load balancing and appropriate weights (processor performance parameter), we can achieve better performance than in the solving process without load balancing or with the existing dynamic load balancing framework not using micro benchmark tests.

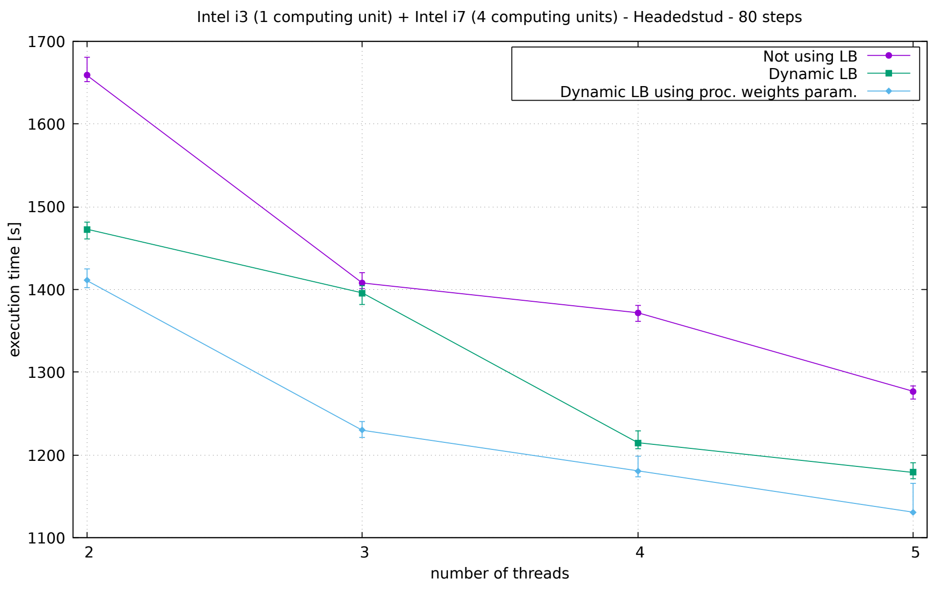


Figure 7.3: Graph of execution times comparing solutions with estimated processing weights, uniform processing weights and without a load balancing framework.

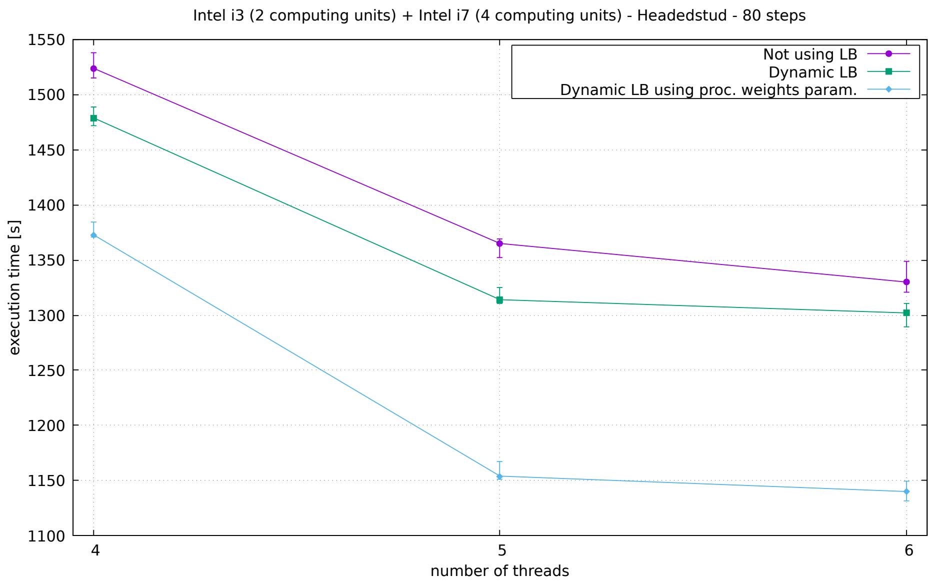


Figure 7.4: Graph of execution times comparing solutions with estimated processing weights, uniform processing weights and without a load balancing framework.

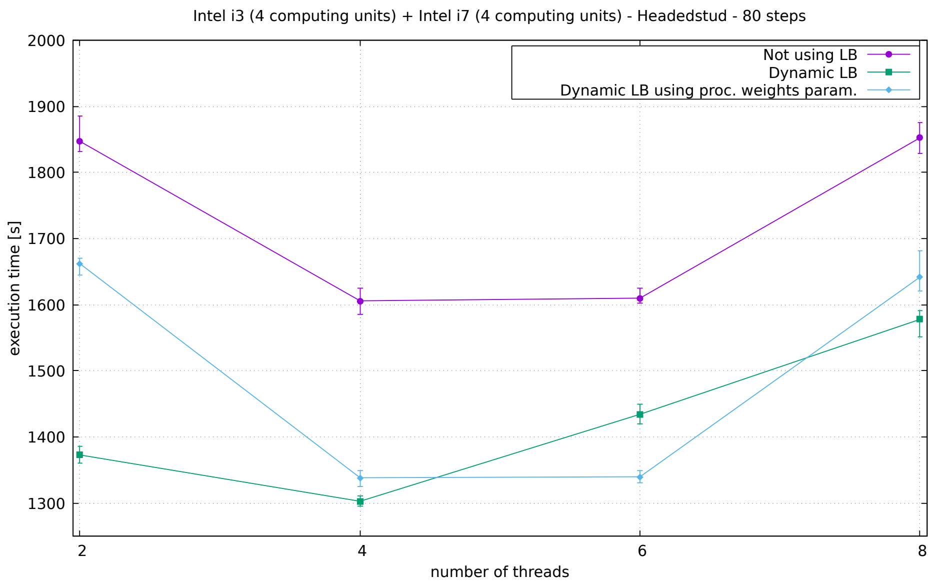


Figure 7.5: Graph of execution times with hyper-threading comparing solutions with estimated processing weights, uniform processing weights and without a load balancing framework.

The next set of results is shown in Figures 7.6 and 7.7, from the workstations with a Core^{TM} i7-4790 CPU @ 3.60 GHz, Core^{TM} i3-2370M CPU @ 2.40 GHz and Core^{TM} i3-2370M CPU @ 2.40 GHz. The maximum number of threads with Intel hyper-threading technology disabled was used for the parallel computation.

First, the results confirm that better performance is obtained when appropriate weights are used. The difference between these two figures is the number of computational threads used in the Intel i3, which in the first graph was set to one thread and in the second graph set to two. Scalability clearly did not have a linear trend and was not ideal in both cases. Scalability in our case was clearly affected by multiprocessors sharing of resources (memory, network and system I/O).

The effect of shared resources in multiprocessors on scalability is shown in Figure 7.8 from an object-oriented, parallel finite element framework with dynamic load balancing [31]. The problem was solved on an SGI Altix 4700 machine installed at the CTU computing centre (dual-core Intel Itanium2 CPUs at 1.6 GHz, NUMA LINK 4 with 6.4 GB/s bidirectional data transfer rate). The SGI Altix 4700 machine with fifty-five dual processors is used as a multicomputer, which means each core uses only one thread in computation. This case is characterized by a zero degree of shared resources in one node (memory, I/O). Our case used multiprocessors with shared resources. The results from the multicomputer SGI Altix showed a clear linear trend and represented good scalability of the parallel algorithm.

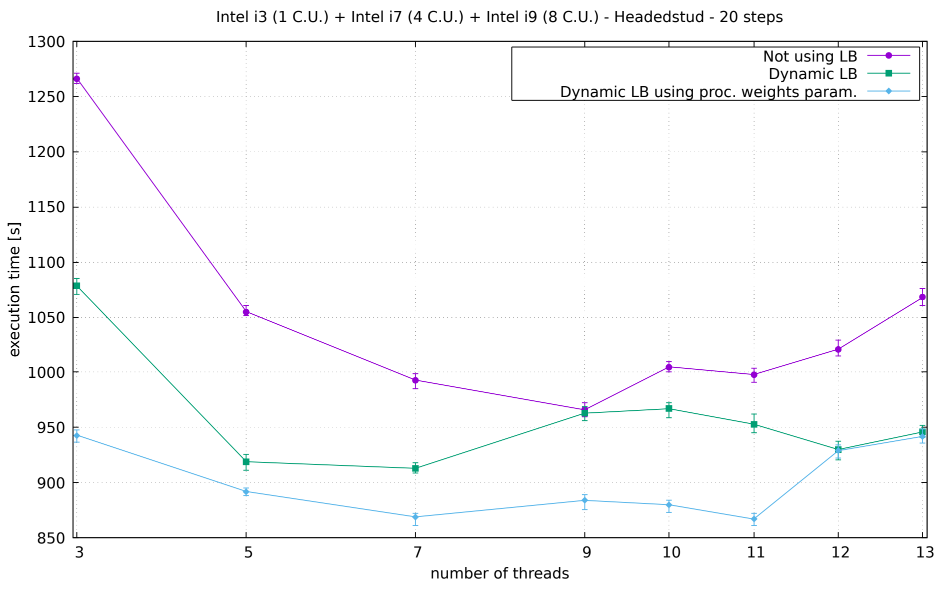


Figure 7.6: Graph of execution times comparing soultions with estimated processing weights, uniform processing weights and without a load balancing framework.

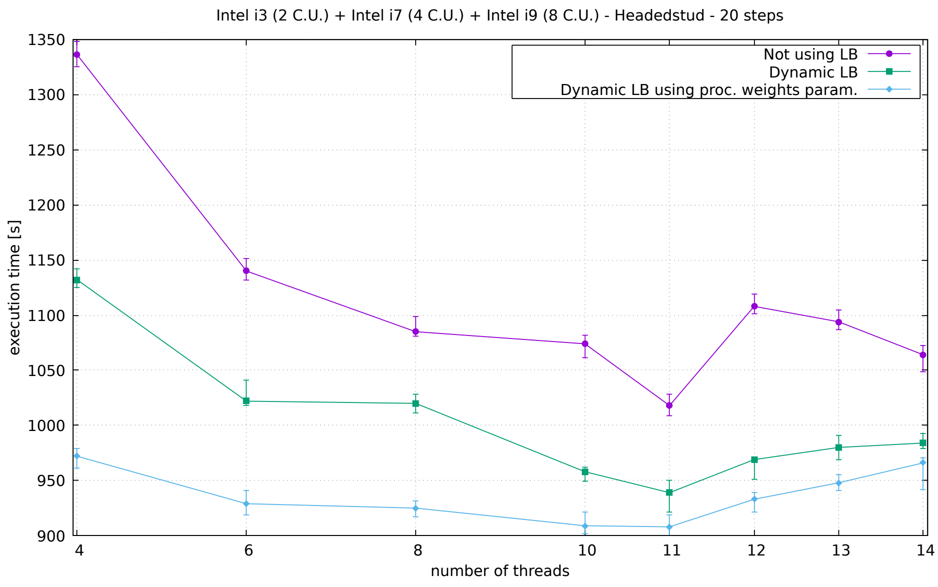


Figure 7.7: Graph of execution times comparing solutions with estimated processing weights, uniform processing weights and without a load balancing framework.

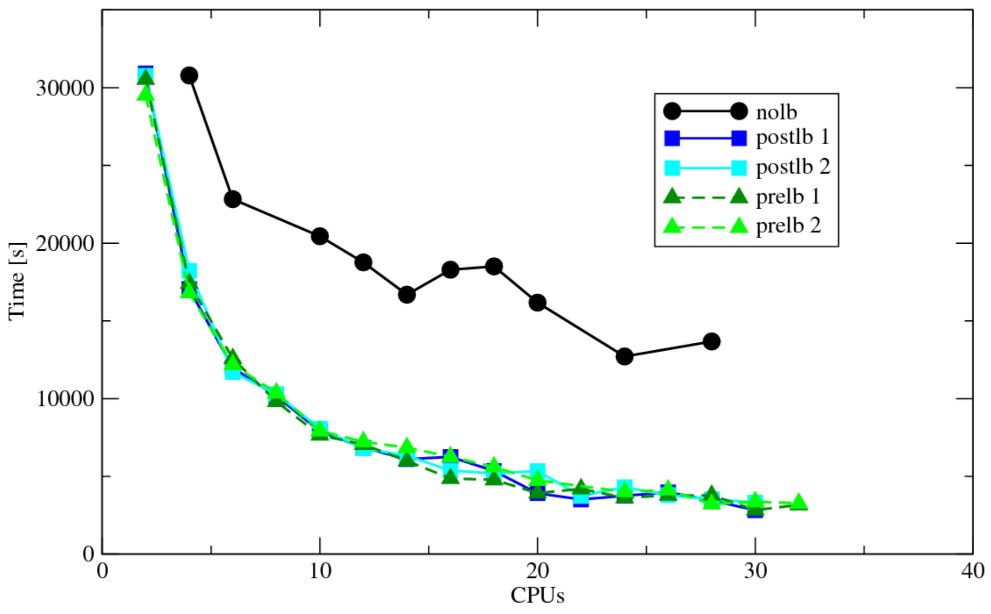


Figure 7.8: Execution times of an anchor pullout test (with load balancing performed before (pre lb) and after (post lb) error assessment).

7.5 conclusion

The contribution based on an improved methodology to determine processing weights parameters as a part of the load balancing framework was presented in this chapter. The parallelization strategy was based on the static and dynamic load balancing process using weights that represented the performance of computational units. The performance of the upgraded static and dynamic load balancing process (processor weights parameters) was compared to the previously implemented static and dynamic load balancing process. In the first case, the linear benchmark problem was solved with a static load balancing framework. The next case examined a nonlinear benchmark problem solved with a dynamic load balancing framework. The results showed differences in the performance of the upgraded and previously implemented static and dynamic load balancing process for the considered benchmark. The upgraded static and dynamic load balancing process had better performance than the previously implemented static and dynamic load balancing process. Future studies will investigate other testing examples (simple linear equation system solutions) to more precisely assess variables as a representation of computational unit performance. It is clearly a benefit to using the appropriate measures of performance of individual computational units performance of computational units in the load balancing process as a necessary part of any Finite Element parallel code.

8 CONCLUSIONS

8.1 SUMMARY

The thesis focused on different aspects of parallelization of finite element software. The first part of the thesis described high-performance computing and available parallel platforms and libraries Chapter 3 generally and then introduced the notion of parallel computing by describing computer architectures and scalability.

The Finite element method and its application to solving structural problems are outlined in Chapter 4. The assembly operations for the right-hand side vector and left-hand side stiffness matrix based on parallel algorithms, which are a critical operation in any finite element software, were discussed in the second part of the thesis (Chapter 5). Performance strategies using different techniques to ensure consistency were developed using with *OpenMP*, *POSIX Threads* and *C++11 Threads libraries*. The performance of individual strategies was evaluated on benchmark problems based on structural mechanics analysis. This study compared different strategies based on individual methods.

The parallel approach to solving large-scale, sparse, non-symmetric systems of linear equations using a direct solver or direct parallel solver based on a shared, distributed memory model was discussed in the next part of the thesis (Chapter 6). The aim of this part was to apply, implement and compare different solver types based on different memory models. The performance of a sequential direct SuperLU solver was compared with an iterative solver and a sequential direct solver with alternative matrix storage (skyline storage). An interface to parallel direct SuperLU solver (based on either a shared memory model or distributed memory model) was implemented and compared with parallel iterative PETSc solver based on a distributed memory model. The performances of individual solvers were evaluated and compared using selected benchmark problems.

Chapter 7 focused on evaluation and tuning of load balancing framework in OOFEM solver. The need for load balancing has been discussed and design of existing load balancing framework has been described. An improved method to determine actual processing weights has been proposed and compared to existing approaches on the selected benchmark test.

8.2 original contributions

- Evolutionary algorithms for right-hand side vector and stiffness matrix assembly operations were presented. The different parallelization strategies based on different shared memory libraries *OpenMP*, *POSIX Threads* and *C++11 Threads* were implemented. This is a major difference from previous sequential vector and matrix assembly strategies. However, their focus is on optimizing the assembly process by reducing computational time and more efficiently using modern parallel hardware. The different assembly parallelization strategies based on a shared memory model were described in papers [5], [6], [7], [8] and [9].

- A new interface to the direct SuperLU solver for solving large sparse systems of linear equations were presented in the thesis. The SuperLU solver is a general purpose library and has three libraries *Sequential SuperLU*, *Multithreaded SuperLU* (shared memory model) and *Distributed SuperLU* (distributed memory model). The implemented SuperLU interfaces were compared with a previously implemented sequential direct solver (Skyline matrix), sequential iterative solver (IML) and parallel iterative solver based on the distributed memory model (PETSc). The presentation of different interfaces of the direct SuperLU solver and a comparison with the above-mentioned solvers were discussed in papers [10], [11], [12] and [13].

- The improvement of load balancing framework, based on a realistic estimation of actual processing performances of individual nodes is presented. The input parameters represent the performance of computational units. The original contribution is based on an upgraded static and dynamic load balancing process (processor weight parameters). The new implemented methods were compared with previously static and dynamic load balancing frameworks and the solution process without a load balancing process. The newly static load balancing framework designing process and opportunities were presented in paper [14]. New opportunities for a dynamic load balancing framework that had not previously been published were presented in the thesis.

8.3 recommendations for future work

The performance of the described load balancing framework could potentially be significantly improved by considering communication speeds between processing nodes by using parameters that represent communication speeds through different, heterogeneous networks. The load balancing framework is based on parallel computation using a distributed memory model. This model uses a message passing interface. For example, many distributed systems are now being constructed using a variety of different communication networks, such as Ethernet and Asynchronous Transfer Mode (ATM). In addition to this hardware heterogeneity, there is a heterogeneity in the types of messages produced by parallel programs. Short synchronization messages require low latency. Conversely, large data messages require high-bandwidth, though they can tolerate high start-up latency. The different types of networks have different performance characteristics, while the different types of communication messages may have different communication requirements. The performance of parallel computations based on a distributed memory model is typically limited by communication overhead. High-performance networks and the use of multiple heterogeneous networks can help reduce this overhead. Further research could focus on designing the load balancing framework's input parameters, which represent different types of networks providing a data path between the same pair of network nodes. These load balancing parameters (network weight parameters) may allow a solution process to maximize efficient use of different types of networks rather than passively accept the given features of a single network.

The importance of parallel computing is increasing rapidly with multi-core CPUs, GPUs (graphics processing unit) and cluster systems. Computationally demanding and data-intensive scientific engineering computations are strongly affected by this trend and require innovative efficient parallel solutions. Further research could focus on improving the performance of parallel computations using GPUs. The GPU accelerates solutions running on the CPU by offloading some of the computationally intensive and time consuming portions of the code. Applications can reduce computational time by using the massively parallel processing power of the GPU to boost performance. This type of parallel computing is known as heterogeneous or hybrid computing. A CPU consists of tens of CPU cores, while a GPU consists of hundreds of smaller cores. Indeed, some types of GPUs are designed as computational accelerators or companion processors optimized for scientific and technical computing applications.

BIBLIOGRAPHY

[1] R. Leland B. Hendrickson. "Multilevel algorithm for partitioning graphs." In: *Proc. Supercomputing 95* (1995).

[2] D. Buttlar B. Nicols and J. Proulx Farrell. *Pthreads Programming*. O'Reilly and Associates, 1996. ISBN: 1-56592-115-1.

[3] B. Barney. *OpenMP - An Application Program Interface*. 2014. URL: https://computing.llnl.gov/tutorials/openMP/.

[4] S. Benkner and T. Brandes. "Efficient Parallelization of Unstructured Reductions on Shared Memory Parallel Architectures." In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 435–442. DOI: 10.1007/3-540-45591-4_58. URL: https://doi.org/10.1007%2F3-540-45591-4_58.

[5] M. Bosansky and B. Patzak. "Using OpenMP in OOFEM." In: *In Proceedings of 4th Conference Nano & Macro Mechanics, 4th Conference Nano & Macro Mechanics 2013, Prague* (2013). ISBN: 978-80-01-5332-4, pp. 23–26.

[6] M. Bosansky and B. Patzak. "On Parallelization Of Stiffness Matrix Assembly." In: *n 20 th International Conference on Engineering Mechanics 2014, Svratka, 2014. Brno: Brno University of Technology* (2014). ISBN: 978-80-214-4871-1, pp. 100–103.

[7] M. Bosansky and B. Patzak. "Different Approaches to Parallelization of Sparse Matrix Assembly Operation." In: *Applied Mechanics and Materials* 825 (Feb. 2016). ISBN:978-3-03835-603-5, pp. 91–98. ISSN: 1660-9336. DOI: 10.4028/www.scientific.net/amm.825.91. URL: https://doi.org/10.4028%2Fwww.scientific.net%2Famm.825.91.

[8] M. Bosansky and B. Patzak. "Different Approaches to Parallelization of Vector Assembly." In: *Applied Mechanics and Materials* 821 (Jan. 2016), pp. 341–348. ISSN: 1662-7482. DOI: 10.4028/www.scientific.net/amm.821.341. URL: https://doi.org/10.4028%2Fwww.scientific.net%2Famm.821.341.

[9] M. Bosansky and B. Patzak. "On Parallelization Of Assembly Operations In Finite Element Method." In: *Engineering Mechanics 2016 - Book of full text. 22nd International Conference on Engineering Mechanics, Svratka* (2016). ISBN: 978-80-97012-59-8, pp. 198–201. ISSN: 1805-8248.

[10] M. Bosansky and B. Patzak. "Evaluation of Different Approaches to Solution of the Direct Solution of Large, Sparse Systems of Linear Equations." In: *Advanced Materials Research* 1144 (Mar. 2017). ISBN: 978-3-0357-1092-2, pp. 97–101. ISSN: 1022-6680. DOI: 10.4028/www.scientific.net/amr.1144.97. URL: https://doi.org/10.4028%2Fwww.scientific.net%2Famr.1144.97.

[11] M. Bosansky and B. Patzak. "On Parallelization Of Linear System Equation Solver In Finite Element Method." In: *Proceeding of Engineering Mechanics 2017. Engineering Mechanics 2017, Svratka, 2017. Brno: Brno University of Technology* (2017). ISBN: 978-80-21454-97-2, pp. 198–201. ISSN: 1805-8248.

[12] M. Bosansky and B. Patzak. "Parallel Approach To Solve Of The Direct Solution Of Large Sparse Systems Of Linear Equations." In: *Acta Polytechnica CTU Proceedings* 13 (Nov. 2017), pp. 50–53. ISSN: 978-80-01-06346-0. DOI: 10.14311/app.2017.13.0016. URL: https://doi.org/10.14311%2Fapp.2017.13.0016.

[13] M. Bosansky and B. Patzak. "Performance Evaluation Of Different Linear Equation Solvers For Solving Nonlinear FE Problems On Multicore Architectures." In: *Acta Polytechnica CTU Proceedings* 15 (Dec. 2018), pp. 6–11. ISSN: 1805-8248. DOI: 10.14311/app.2018.15.0006. URL: https://doi.org/10.14311%2Fapp.2018.15.0006.

[14] M. Bosansky and B. Patzak. "On tuning of finite element load balancing framework." In: *Engineering Mechanics 2019 - Book of full text. 25nd International Conference on Engineering Mechanics, Svratka* (May 2019). ISBN: 978-80-87012-71-0, pp. 61–64. ISSN: 1805-8248. DOI: 10.21495/71-0-61. URL: https://doi.org/10.21495%2F71-0-61.

[15] H. J. Curnow. *A synthetic benchmark*. Vol. 19. 1. Oxford University Press (OUP), Jan. 1976, pp. 43–49. DOI: 10.1093/comjnl/19.1.43. URL: https://doi.org/10.1093%2Fcomjnl%2F19.1.43.

[16] D. Hill et al. D. Marr F. Binns. "Hyper-Threading Technology Architecture and Microarchitecture." In: *Intel Technology Journal* 6.1 (2002). URL: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=6769155&site=ehost-live&scope=site.

[17] H. Simon H. Mauer E. Strohmaier J. Dongarra and M. Mauer. "TOP 500 The list home page." In: *https://www.top500.org/* (1993).

[18] M. Flynn. "Flynn's Taxonomy." In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 689–697. DOI: 10.1007/978-0-387-09766-4_2. URL: https://doi.org/10.1007%2F978-0-387-09766-4_2.

[19] V. Kumar. G. Karypis. *ParMETIS: Parallel graph partitioning and sparse matrix ordering library*. Department of Computer Science, University of Minnesota, 1997.

[20] V. Kumar. G. Karypis. *A fast and high quality multilevel scheme for partitioning irregular graphs*. University of Minnesota, Dept. Computer Science, Minneapolis, 1999.

[21] X. Guo, G. Gorman, M. Lange, L. Mitchell, and M. Weiland. "Exploring the Thread-level Parallelisms for the Next Generation Geophysical Fluid Modelling Framework Fluidity-ICOM." In: *Procedia Engineering* 61 (2013), pp. 251–257. DOI: 10.1016/j.proeng.2013.08.012. URL: https://doi.org/10.1016%2Fj.proeng.2013.08.012.

[22] J. Handy. *The cache memory book*. Academic Press, Inc., 1998. ISBN: 0-12-322980-4.

[23] S. Harbaugh and John A. Forakis. “Timing studies using a synthetic Whetstone benchmark.” In: *ACM SIGAda Ada Letters* IV.2 (Sept. 1984), pp. 23–34. DOI: 10.1145/998395.998396. URL: https://doi.org/10.1145%2F998395.998396.

[24] B. Hendrickson and R. Leland. *The Chaco user's guide. Version 1.0*. Office of Scientific and Technical Information (OSTI), Nov. 1993. DOI: 10.2172/10106339. URL: https://doi.org/10.2172%2F10106339.

[25] P. Jarzebski, K. Wisniewski, and R. L. Taylor. “On parallelization of the loop over elements in FEAP.” In: *Computational Mechanics* 56.1 (2015), pp. 77–86. DOI: 10.1007/s00466-015-1156-z. URL: https://doi.org/10.1007%2Fs00466-015-1156-z.

[26] G. Karypis K. Schloegel and V. Kumar. “Parallel static and dynamic multi-constraint graph partitioning.” In: *Concurrency and Computation: Practice and Experience* 14.3 (2002), pp. 219–240. DOI: 10.1002/cpe.605. URL: https://doi.org/10.1002%2Fcpe.605.

[27] G. Karypis and V. Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.” In: *SIAM Journal on Scientific Computing* 20.1 (Jan. 1998), pp. 359–392. DOI: 10.1137/s1064827595287997. URL: https://doi.org/10.1137%2Fs1064827595287997.

[28] P. Krysl and Z. Bittnar. “Parallel explicit finite element solid dynamics with domain decomposition and message passing; deal programming scalability.” In: *Comput Struct* 79.3 (2001), pp. 45–60.

[29] A. Marowka. *Book Review [review of "Using OpenMP: Portable Shared Memory Parallel Programming" (Chapman, B. et al, 2007)]*. Vol. 9. 1. Institute of Electrical and Electronics Engineers (IEEE), Jan. 2008, pp. 3–3. DOI: 10.1109/mdso.2008.1. URL: https://doi.org/10.1109%2Fmdso.2008.1.

[30] B. Patzak and Z. Bittnar. “Design of object oriented finite element code.” In: *Advances in Engineering Software* 32.10-11 (Oct. 2001), pp. 759–767. DOI: 10.1016/s0965-9978(01)00027-8. URL: https://doi.org/10.1016%2Fs0965-9978%2801%2900027-8.

[31] B. Patzak and D. Rypl. “Object-oriented, parallel finite element framework with dynamic load balancing.” In: *Advances in Engineering Software* 47.1 (May 2012), pp. 35–50. DOI: 10.1016/j.advengsoft.2011.12.008. URL: https://doi.org/10.1016%2Fj.advengsoft.2011.12.008.

[32] F. Pelligrini. *SCOTCH 3.4 user's guide*. LaBRI, 2001.

[33] R. Diekmann. R. Preis. *The PARTY partitioning library, user guide*. Vol. version 1.1. Dept. of Computer Science, University of Paderborn, Paderborn, Germany, 1996.

[34] K. Buschelman V. Eijkhout W. Gropp D. Kaushik M. Knepley L. C. McInnes B. Smith S. Balay J. Brown and H. Zhang. *PETSc Users Manual Revision 3.4*. Tech. rep. June 2014. DOI: 10.2172/1178104. URL: https://doi.org/10.2172%2F1178104.

[35] C. Walshaw. *The Parallel JOSTLE Library, User's Guide*. Vol. version 3.0. University of Greenwich, London, UK, 2002.

[36] C. Walshaw and M. Jostle. "Parallel multilevel graph-partitioning software – an overview." In: *Magoules F, editor. Mesh partitioning techniques and domain decomposition techniques. Civil-Comp Ltd* (2007), pp. 27–58.

[37] D. B. West. *Introduction to graph theory.* Vol. Vol. 2. Upper Saddle River Prentice hall, 2001.

[38] A. Williams. *C++ Concurrency in Action*. Manning Publications Co, United States of America, 2012. ISBN: 978193398877.