UNIVERSITÀ DEGLI STUDI DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Ingegneria Industriale

SCUOLA DI DOTTORATO DI RICERCA IN INGEGNERIA INDUSTRIALE

INDIRIZZO INGEGNERIA DELL'ENERGIA

XXVI CICLO

APPLICATION OF HPC IN EDDY CURRENT ELECTROMAGNETIC PROBLEM SOLUTION

Direttore della Scuola: Ch.mo Prof. Paolo Colombo

Coordinatore d'indirizzo: Ch.ma Prof.ssa Luisa Rossetto

Supervisore: Ch.mo Prof. Michele Forzan

Dottorando: Cristian Pozza

To my family.

Acknowledgements

This research work has been carried out from January 2011 to December 2013 at the Laboratory for Electroheat, Department of Industrial Engineering, University of Padova. Part of the experimental activity has been performed during an internship at Cedrat Group, Grenoble, France. The development of new numerical techniques, as low rank approximation and multithreading approach, has been accomplished by MUMPS team.

With these few lines, I would like to acknowledge all people who helped me in this work, starting from the supervisor Dr. Michele Forzan, and the leader of the research group, Prof. Fabrizio Dughiero. I would like to thank Cedrat Group, in particular its Technical Director Vincent Leconte and all R&D team, for offering me the opportunity to spend three months among such valuable people. In the same way, I would like to thank Prof. Patrick Amestoy and MUMPS development team for the fruitful cooperation, which helped me focusing on the practical aspects of parallel computing. A special thanks goes to the reviewers of this thesis, Prof. Ivo Dolezel and Prof. Egbert Baake. Finally, I would like to thank all colleagues from LEP: Aristide, Alessandro C., Alessandro T., Dario, Elisabetta, Fernando, Francesco, Marco, Mattia, and all those who I might have omitted here.

A special thanks goes to my family for supporting me over the years.

Abstract

As engineering problems are becoming more and more advanced, the size of an average model solved by partial differential equations is rapidly growing and, in order to keep simulation times within reasonable bounds, both faster computers and more efficient software implementations are needed.

In the first part of this thesis, the full potential of simulation software has been exploited through high performance parallel computing techniques. In particular, the simulation of induction heating processes is accomplished within reasonable solution times, by implementing different parallel direct solvers for large sparse linear system, in the solution process of a commercial software. The performance of such library on shared memory systems has been remarkably improved by implementing a multithreaded version of MUMPS (MUltifrontal Massively Parallel Solver) library, which have been tested on benchmark matrices arising from typical induction heating process simulations.

A new multithreading approach and a low rank approximation technique have been implemented and developed by MUMPS team in Lyon and Toulouse. In the context of a collaboration between MUMPS team and DII-University of Padova, a preliminary version of such functionalities could be tested on induction heating benchmark problems, and a substantial reduction of the computational cost and memory requirements could be achieved.

In the second part of this thesis, some examples of design methodology by virtual prototyping have been described. Complex multiphysics simulations involving electromagnetic, circuital, thermal and mechanical problems have been performed by exploiting parallel solvers, as developed in the first part of this thesis. Finally, multiobjective stochastic optimization algorithms have been applied to multiphysics 3D model simulations in search of a set of improved induction heating device configurations.

Sommario

Nell'ultima decade, i problemi ingegneristici sono diventati sempre più complessi e le dimensioni dei relativi modelli numerici sono notevolmente aumentate. Al fine di mantenere i tempi di calcolo entro limiti ragionevoli è necessario utilizzare computer sempre più performanti ed implementare codici di calcolo più efficienti.

Nella prima parte di questo elaborato sono descritte ed ampiamente utilizzate le più recenti tecniche di programmazione per il calcolo parallelo ad alte prestazioni, permettendo di sfruttare pienamente le potenzialità dei moderni software di simulazione. In particolare, il tempo di calcolo necessario per la simulazione numerica dei processi di riscaldamento per induzione magnetica è stato considerevolmente ridotto attraverso l'implementazione di solutori diretti paralleli per matrici sparse nel processo di soluzione di un software commerciale. Successivamente, grazie alla collaborazione con gli sviluppatori del solutore diretto MUMPS (MUltifrontal Massively Parallel Solver), le prestazioni di tale libreria sono state ulteriormente migliorate grazie all'utilizzo di librerie BLAS parallele. Una serie di test sono stati condotti sulla soluzione di matrici ricavate dalle analisi agli elementi finiti di problemi tipici dell'elettromagnetismo e del riscaldamento per induzione.

Grazie all'introduzione di un nuovo approccio "multi-threading" e all'utilizzo di tecniche di compressione delle matrici (low-rank approximation), il team di MUMPS (Lione-Tolosa) ha sviluppato alcune funzionalità sperimentali. Nel contesto di una collaborazione tra il team di MUMPS ed il Dipartimento di Ingegneria Industriale, Università di Padova, l'utilizzo della libreria in versione sperimentale ha permesso una notevole riduzione del costo computazionale e della memoria necessaria per la fattorizzazione e la soluzione dei problemi analizzati.

Nella seconda parte di questo elaborato sono riportati alcuni esempi di prototipazione virtuale attraverso software agli elementi finiti. Lo studio di sistemi multiphysics molto complessi, che comprendono fenomeni elettromagnetici, circuitali, termici e meccanici, è stato effettuato su modelli di dimensioni notevoli ed in tempi ridotti, sfruttando le tecniche di calcolo parallelo sviluppate nella prima parte di questa tesi. Infine, grazie ai miglioramenti introdotti con il calcolo parallelo, l'ottimizzazione di dispositivi elettromagnetici attraverso algoritmi stocastici multiobiettivo è stata applicata ad un problema multiphysics su modelli tridimensionali.

Contents

Abstract I
Sommario II
List of Figures V
List of Tables IX
Glossary XI
Introduction 1
1. What is parallel computing? 3
1.1 Concepts and terminology 10
1.1.1 Classification based on instruction and data streams 11
1.1.2 Classification based on computer structure 13
1.1.3 Classification based on grain size 16
1.2 Parallelism conditions 18
1.3 Amdahl-Gustafson's law 20
2. Parallelization in shared memory computers 23
2.1 OpenMP 23
2.2 Basic Linear Algebra Subprograms 27
3. Parallelization in distributed memory computers 29
3.1 Message passing paradigm 29
3.2 MPI – Message Passing Interface 29
3.2.1 MPICH library 31
4. State of the art of multicore systems 33
4.1 Emerging technologies 35
4.1.1 Nvidia GPU 36
4.1.2 Intel MIC 39
4.2 Scalability Issues for Many-core Processors 40
5. Simulation of electromagnetic field problems 43
5.1 Finite Element Method 43
5.1.1 FEM in induction heating simulation 47

5.1.2 Nodal and edge elements 48
5.1.3 Definition of the electromagnetic problem 49
5.2 Integral equations and hybrid methods in induction heating simulation 53
6. Sparse linear systems: direct and iterative solvers 57
6.1 Direct methods 61
6.2 Frontal solvers 62
6.3 Multifrontal solvers 64
6.5 MUMPS software package 71
6.6 Improvements of shared memory parallelism 73
6.7 Improvements by Low-Rank approximation techniques 79
6.8 PARDISO software package 82
6.9 Iterative methods 83
7. Application of parallel direct solvers to FEM software 89
7.1 Benchmark matrices from induction heating simulations 91
7.2 Experimental results 97
8. Design and optimization of induction heating applications 103
8.1 Design of a Permanent Magnet Heater 103
8.2 Optimization of temperature profiles in heated parts 112
8.3 Multiphysics modeling of quasi-resonant induction cooktops 122
Conclusions 131
Nomenclature 133
Bibliography 135

List of Figures

1	Serial execution of instructions ……	3
2	Parallel execution of instructions ……	3
3	Parallel task ……	4
4	Semiconductor manufacturing technology ……	6
5	Multicore processor ……	7
6	Performance of different processors ……	7
7	Growth of parallel computing ……	8
8	System share ……	9
9	Architecture of a CPU ……	10
10	Instruction and data streams ……	11
11	Flynn's Classical Taxonomy ……	11
12	Execution on a SISD computer ……	12
13	Execution on a SIMD computer ……	12
14	Vectorized sum on a SIMD computer ……	13
15	Execution on a MIMD computer ……	13
16	Shared memory system ……	14
17	Interconnection networks ……	15
18	Distributed memory system ……	16
19	Parallel execution of instructions ……	17
20	Parallelism levels ……	17
21	Dependency conditions ……	19
22	Control dependency ……	20
23	Several gains through parallelism ……	21

24 Amdahl’s law ……………………………………………………………………….. 22

25 OpenMP parallel region ……………………………………………………….. 24

26 Graphical representation of a parallel region ……………………………………. 25

27 Data dependency ………………………………………………………………. 26

28 MPI example …………………………………………………………………… 31

29 Cache coherence mechanism ………………………………………………. 34

30 GPU functional subdivision ………………………………………………… 37

31 GPU streaming multiprocessor …………………………………………….. 37

32 NVIDIA Tesla k40 GPU ……………………………………………………. 39

33 Intel Xeon Phi …………………………………………………………………. 39

34 FEM test function ……………………………………………………………… 46

35 Linear combination of test functions ………………………………………. 46

36 Representation of conductors by PEEC method …………………………. 55

37 Sparse matrix-vector product ……………………………………………….. 59

38 Representation of sparse matrix-vector product …………………………. 59

39 Solution of a lower triangular system …………………………………… 60

40 Solution phases by a sparse direct solver …………………………………. 62

41 Factorization example ………………………………………………………. 64

42 Relation between matrices and graphs ……………………………………. 65

43 Assembly tree associated to a sum ……………………………………… 66

44 Example of assembly tree ………………………………………………… 67

45 Nested dissection ……………………………………………………………. 68

46 One level of nested dissection procedure ………………………………… 69

47 Subgraphs identified by nested dissection ……………………………… 70

48 Structure of a front ……………………………………………………….. 71

49 Construction of L_{th} layer …………………………………………………………………… 77

50 Example of Strong and weak interactions in a domain ………………………………… 80

51 Example of a Block Low-Rank matrix ……………………………………………….. 81

52 Solution loop for the simulation of induction heating processes ………………………. 89

53 Solution scheme of a non-linear problem …………………………………………….. 90

54 Solution loop in Cedrat Flux software ……………………………………………….… 90

55 Pancake coil finite-element model ……………………………………………………. 92

56 Mesh of pancake coil and graphite susceptor ………………………………………….. 93

57 Sparsity pattern of a matrix arising from pancake coil model …………………………….. 93

58 Eddy current distribution in a graphite susceptor …………………………………….. 94

59 Induction hardening finite-element model ……………………………………………. 95

60 Mesh of inductor and gear …………………………………………………………….. 95

61 Sparsity pattern of a matrix arising from induction hardening model (1) …………….. 96

62 Sparsity pattern of a matrix arising from induction hardening model (2) …………….. 96

63 Eddy current distribution in a steel gear ……………………………………………… 97

64 In-Core factorization time …………………………………………………………….. 98

65 Factorization time for large matrices ………………………………………………… 99

66 Finite-element model of a permanent magnet heater ………………………………….. 104

67 Prototype of a permanent magnet heater ……………………………………………….. 104

68 Measuring system ……………………………………………………………………….. 105

69 Power induced in the aluminum billet ………………………………………………….. 106

70 Experimental validation of numerical results ………………………………………… 107

71 Temperature evolution inside the billet ……………………………………………….. 108

72 Global efficiency for different rotational speed ………………………………………. 109

73 Finite-element model of a dual rotor permanent magnet heater ………………………… 110

74 Eddy current distribution inside the billet ………………………………………………. 110

75 Power induced in the billet for different rotational speed ………………………………. 111

76 Example of direct problem in optimization strategies ………………………………….. 113

77 Criterion of proximity ………………………………………………………………………. 114

78 Implementation of criterion of proximity ………………………………………………. 115

79 Pareto front – first strategy ……………………………………………………………… 116

80 Temperature profiles corresponding to the first iteration ………………………………. 116

81 Temperature profiles corresponding to the last iteration ……………………………….. 117

82 Comparison between two individuals on Pareto front …………………………………. 117

83 Pareto front – second strategy …………………………………………………………… 118

84 Temperature profiles corresponding to the first iteration ………………………………. 119

85 Temperature profiles corresponding to the last iteration ………………………………. 119

86 Comparison between two individuals on Pareto front ………………………………….. 120

87 Representation of temperature uniformity for different iterations …………………….. 121

88 Pancake coil for induction cooktops …………………………………………………….. 123

89 Half-bridge resonant converter ………………………………………………………….. 124

90 Quasi Resonant converter ……………………………………………………………….. 124

91 Equivalent 2D finite-element model of a pancake coil …………………………………. 125

92 3D finite-element model of a pancake coil ……………………………………………… 126

93 Coupling between circuital and finite-element model ………………………………….. 127

94 Multiphysics model of a Quasi-Resonant cooktop ……………………………………… 128

95 Evolution of main electrical quantities vs time …………………………………………. 129

96 Distribution of magnetic field and eddy currents in the domain ……………………….. 130

List of Tables

1 Cache coherence mechanism 35

2 GPUs technical specifications 38

3 Matrices from MUMPS development team 74

4 Factorization times with threaded BLAS 76

5 Factorization times with AlgLth algorithm 78

6 Evaluation of costs for dense and low-rank linear algebra operations 80

7 Matrices from induction heating benchmarks 98

8 Memory usage for solving large problems 98

9 Factorization times with threaded BLAS 99

10 Factorization times with AlgL0 algorithm 100

11 Application of low-rank approximation techniques to induction heating benchmarks .. 101

12 Temperature of the billet for different rotational speed (dual rotor configuration) 111

13 Evaluations of objective functions for different individuals 121

Glossary

MUMPS	MUltifrontal Massively Parallel Solver
PDE	Partial Differential Equation
CPU	Central Processing Unit
FLOPS	FLoating point Operations Per Second
SAAS	Sofware As A Service
MIPS	Million Instructions Per Second
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
GPU	Graphic Processor Unit
SMP	Symmetric MultiProcessing
NUMA	Non-Uniform Memory Access
PVM	Parallel Virtual Machine
MPI	Message Passing Interface
BLAS	Basic Linear Algebra Subprograms
MPICH	MPI Chameleon
OS	Operating System
SMPD	Simple MultiPurpose Daemon
CMOS	Complementary Metal-Oxide Semiconductor
MIC	Many Integrated Core Architecture
DRAM	Dynamic Random Access Memory
PCI	Peripheral Component Interconnect
FEM	Finite Element Method

ICC	Inductively Coupled Circuits method
VIM	Volume Integral Method
BEM	Boundary Element Method
PEEC	Partial Element Equivalent Circuit method
IBC	Impedance Boundary Condition
EMC	ElectroMagnetic Compatibility
PCB	Printed Circuit Board
COO	Coordinate format for matrix storage
CSR	Compressed Sparse Row format for matrix storage
CSC	Compressed Sparse Column format for matrix storage
SOR	Successive Over-Relaxation method
SSOR	Symmetric SOR
ICCG	Incomplete Cholesky preconditioned Conjugate Gradient method
GMRES	Generalized Minimal Residual method
MINRES	Minimal Residual method
SYMMLQ	Symmetric LQ method
CGNE,CGNR	Conjugate Gradient methods on the Normal Equations
BiCG	Bi-Conjugate Gradient method
BiCGstab	Stabilized BiCG
QMR	Quasi Minimal Residual method
CGS	Conjugate Gradient Squared method
FS	Fully Summed variables
NFS	Non-Fully Summed variables
BLACS	Basic Linear Algebra Communication Subprograms
ScaLAPACK	Scalable Linear Algebra PACKage
PARDISO	PArallel DIrect SOlver

I-DSS	Induction Directional Solidification System
IC	In-Core mode
OOC	Out-Of-Core mode
MKL	Intel Math Kernel Library
BLR	Block Low Rank matrix format
LEP	Laboratory of Electroheat of Padova
PMH	Permanent Magnet Heater
NSGA	Non-Dominated Sorting Genetic Algorithm
QR	Quasi-Resonant converter
IGBT	Insulated Gate Bipolar Transistor
ZVS	Zero Voltage Switching condition
EMI	Electromagnetic Interference

Introduction

Today's computer simulations are employed to study many of the physical experiments required to gain deeper understanding of our world. Often, these experiments are impractical or impossible to be performed in the real world and scientists or engineers can only refer to mathematical models that describe the behaviors and the phenomena of interest. The models may contain Partial Differential Equations (PDE) that need to be solved by numerical methods on computers. As the models are required to be more and more complex and detailed, the size of an average simulation is rapidly growing. In order to keep simulation times within reasonable bounds, both faster computers and more efficient implementations are needed. A common way to reduce simulation times is to use parallel computers, in which the computational problem is distributed over a number of processors and each processor is accountable for a part of the problem. Simulation times can be reduced further by taking into account both the hardware and application characteristics in the parallel implementation of the simulation software.

In this thesis, a reduction of the computing time and memory requirements for the simulation of induction heating processes is one of the most important goals. In order to study real applications, an analytic approach would not be possible due to the presence of non-linear phenomena, and magnetic saturation in magnetic flux concentrators, iron core losses, radiative thermal exchanges could not be accurately taken into account. The need for more detailed analysis during the design and verification process, known as virtual prototyping, led to the diffusion of powerful numerical methods, which allow for the determination of electric current density distribution inside workpieces and inductors, even in case of complex geometrical and physical properties.

Solving linear systems of equations lies at the heart of many problems in computational science and engineering. In many cases, the discretization of continuous problems by differential numerical methods lead to large sparse linear systems. This holds in induction heating applications, as well as for structural problems and fluid dynamics problems.

In last years, the potential of simulation software could be exploited only by a small fraction, as developed models could not be too detailed in order to obtain useful results in a reasonably short time. The scientific community is pushing towards an ever increasing model complexity, with the aim to perform multiphysics simulations and numerical optimizations of the heat treatment processes. This is a consequence of the study of new induction heating processes, that requires an accurate determination of electric and magnetic fields distribution in a physical domain, and temperature distribution inside heated parts.

Benefits carried by parallel computing are discussed in chapter 1. Then, a detailed description of different parallel computing environments is given in chapters 2,3 and 4.

High performance solution methods and their application to induction heating simulation are reviewed and discussed in chapters 5 and 6. Outstanding improvements have been achieved in the context of a joint project developed by the MUMPS team in Bordeaux, Lyon and Toulouse, and supported by many French institutions, as CERFACS, CNRS, ENS Lyon, INPT(ENSEEIHT)-IRIT, INRIA and University of Bordeaux.

In the work described in this thesis, remarkable enhancements have been achieved by improving direct sparse solvers through parallel computing, and simulation time could be reduced of an order of magnitude on inexpensive multicore computers in respect to reference commercial codes. Thanks to a fruitful collaboration with MUMPS team, very recent functionalities developed in Lyon and Toulouse and described in chapter 6, could be tested on matrix benchmarks from induction heating models, leading to further important savings in the computing time and memory needed with respect to reference commercial codes.

Part of this thesis has been developed at Cedrat Group headquarter, in Meylan, Grenoble-France. Most of the improvements presented in this thesis have been implemented in Cedrat Flux 11.1, a commercial 3D finite-element simulation software.

Finally, a review of some innovative induction heating applications, their numerical design and optimizations is addressed in chapter 8, by focusing mainly on the computational cost that such models usually require and on benefits brought by parallel computing in this scientific area.

1. What is parallel computing?

Traditionally, software has been written for serial computation, to be run on a single computer having a single Central Processing Unit (CPU) [1]. By following this approach, a problem is divided in a discrete series of instructions that are executed one by one, as represented in Figure 1.

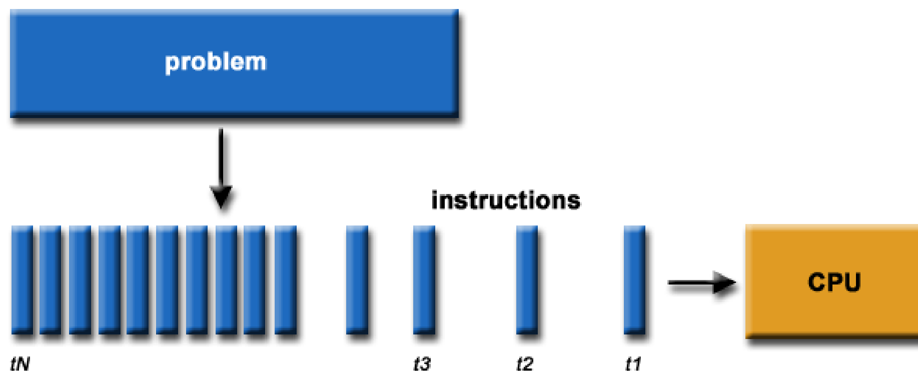


Figure 1. *In a serial computation, a problem is divided in a discrete series of instructions that are executed one by one, in sequence.*

Conversely, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. In this framework, a problem is divided into discrete parts that can be solved concurrently. Each part consists of a sequence of instructions, that can be executed simultaneously by using multiple processors, as it is shown in Figure 2. An overall control mechanism is needed in order to coordinate the computation.

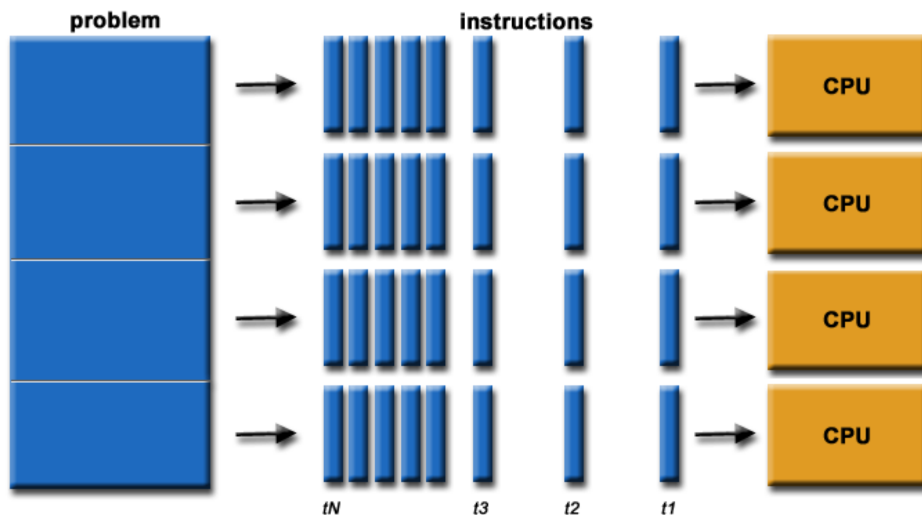


Figure 2. *In a parallel computation, a problem is divided into discrete parts that can be solved concurrently on different processors.*

It is important to point out that, even if every problem could be broken in parts to be executed in parallel, data dependencies and interactions between different parts could prevent from a parallelization of computations and lead to incorrect results. Furthermore, the computational problem should meet some prerequisites, in order to perform the computation efficiently.

The computing resources might be a single computer with multiple processors and a common memory address space. This kind of system is usually referred to as a shared memory system. Otherwise, the computing resources could be a distributed memory system, that is an arbitrary number of computers, each one with a dedicated memory address space, connected by a network. It is also possible to combine benefits of both kind of systems, as it is usual in modern cluster facilities [1].

Historically, parallel computing has been used to model difficult problems in many areas of science and engineering, from the study of applied physics, atmosphere science and geology, to mechanical and electrical engineering. In recent years, industrial and commercial applications provided new driving force in the development of faster computers, able to process large amounts of data in sophisticated ways, as needed for:

- Web search engines, advanced graphics and collaborative work environments
- Storm forecasting, climate prediction
- Quantum chemistry, molecular nanotechnology, pollution modeling, cosmology
- Data mining, oil exploration, financial modeling, vehicle design

By throwing more resources at a task could shorten its process time, or make a bigger task feasible. Furthermore, a well performing parallel computer could be assembled from a few cheap commodity components (Figure 3).

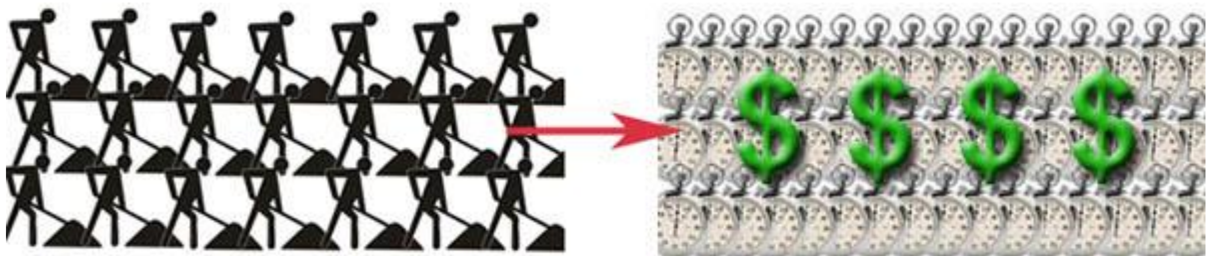


Figure 3. *Throwing more resources at a task could shorten its process time, with potential cost savings, or make a bigger task feasible.*

Furthermore, many problems are so large and complex that it is impractical or impossible to solve them on a single computer, because of limited computing power and available memory. An example is the “Grand Challenge” problems, requiring PetaFLOPS and PetaBytes of computing

resources [2]. Other examples are the web search engines that process millions of transactions per second, searching in huge databases.

Another important aspect of parallel computing is the provision of concurrency. A single compute resource can do only one operation at a time. Vice versa, multiple computing resources are able to perform many operations simultaneously. For example, the Access Grid [3] provides a global collaboration network.

Cloud computing is an emerging technology that allows to perform complex calculations via a SAAS (Software As A Service) approach. In this way, the use of non-local resources on a wide area network, or even on Internet, allows for the solution of very large problems on modern facilities, at a low price [4].

Finally, both physical and practical reasons pose significant constraints to simply building ever faster serial computers. In particular, the speed of a serial computer is directly dependent upon how fast data can be moved through the hardware. Absolute limits are the speed of light (30 cm/ns) and the transmission limit of copper wire (9 cm/ns). To increase speed would mean to approach the processing elements. On the other side, there are still significant limits in miniaturization technologies. Processor technology is allowing for an increasing number of transistors to be placed on a chip (Figure 4). However, even with molecular or atomic-level components, a limit will be reached on how small components could be built [1].

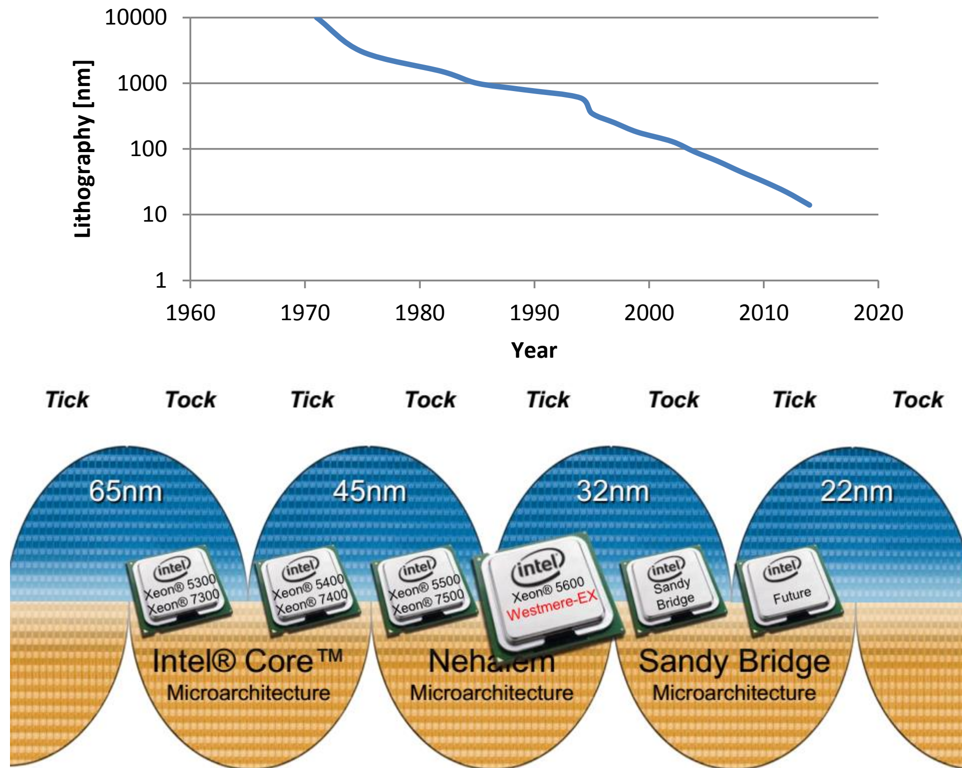


Figure 4. *Increasing speeds would mean to reduce distances between processing elements, by improving lithography technology. There are still significant limits in miniaturization technologies [5].*

Of course, there are economic limitations. It is increasingly expensive to make a single processor faster. Conversely, to use a large number of moderately fast commodity processors is an inexpensive way to achieve similar or better performances. This direction is followed by computer producers since current computer architectures are increasingly relying upon hardware level parallelism to improve performances, as multiple execution units, pipelined instructions and multicore architectures. An example of a multicore processor die is shown in Figure 5.

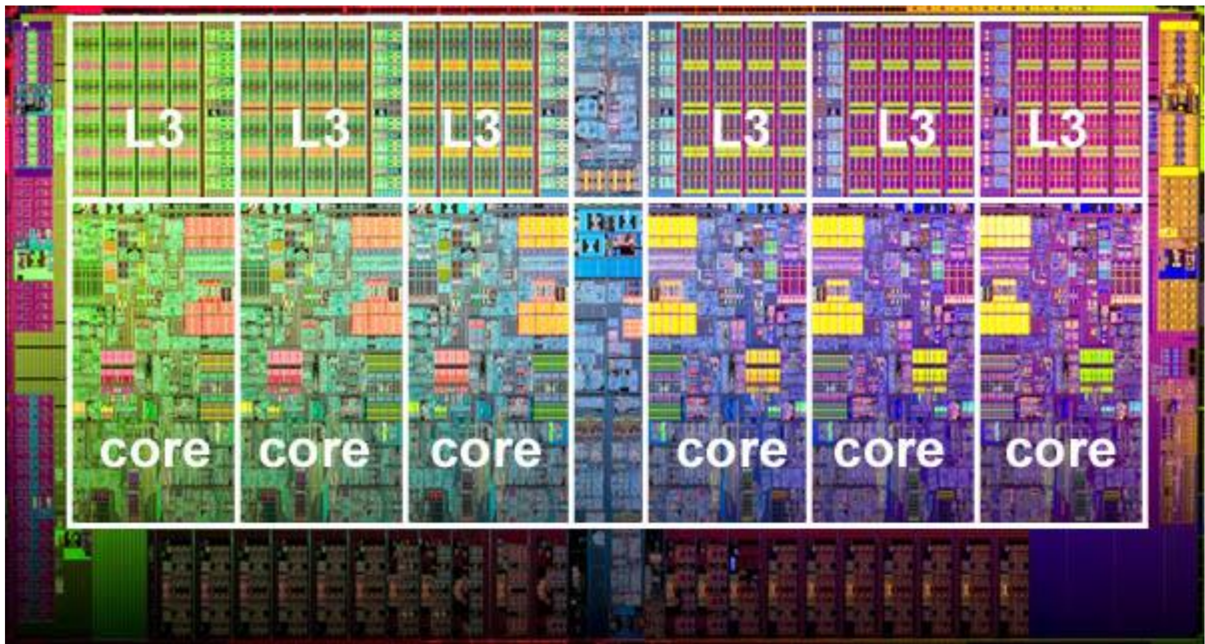


Figure 5. *Example of multicore processor and L3 cache layout.*

CPU	GHz	Core / CPU	MIPS
AMD 6140	2.6	8	62.4k
AMD 6172	2.1	12	75.6k
AMD 6176 SE	2.3	12	82.8k
AMD 6180 SE	2.5	12	90.0k
Intel 5570	2.93	4	46.88k
Intel 6550	2.00	8	64.00k
Intel 5675	3.06	6	73.44k
Intel 7560	2.266	8	75.51k
Intel 5680	3.33	6	79.92k
Intel 5690	3.46	6	83.04k
Intel E7-x870	2.4	10	96.90k

Figure 6. *Million Instructions Per Second (MIPS) is a measure of hardware performance. Results about most recent multicore processors are reported. Note: for Intel processors, the number of instructions retired per clock cycle is 4, for AMD the number is 3 [5].*

During the last decades, the trends indicated by ever faster networks, distributed systems and multiprocessor computer architectures show that parallelism is the future of computing. In the same period, there has been a greater than 1000x increase in supercomputer performance, and today parallel computing facilities are extensively exploited around the world in a wide variety of applications.

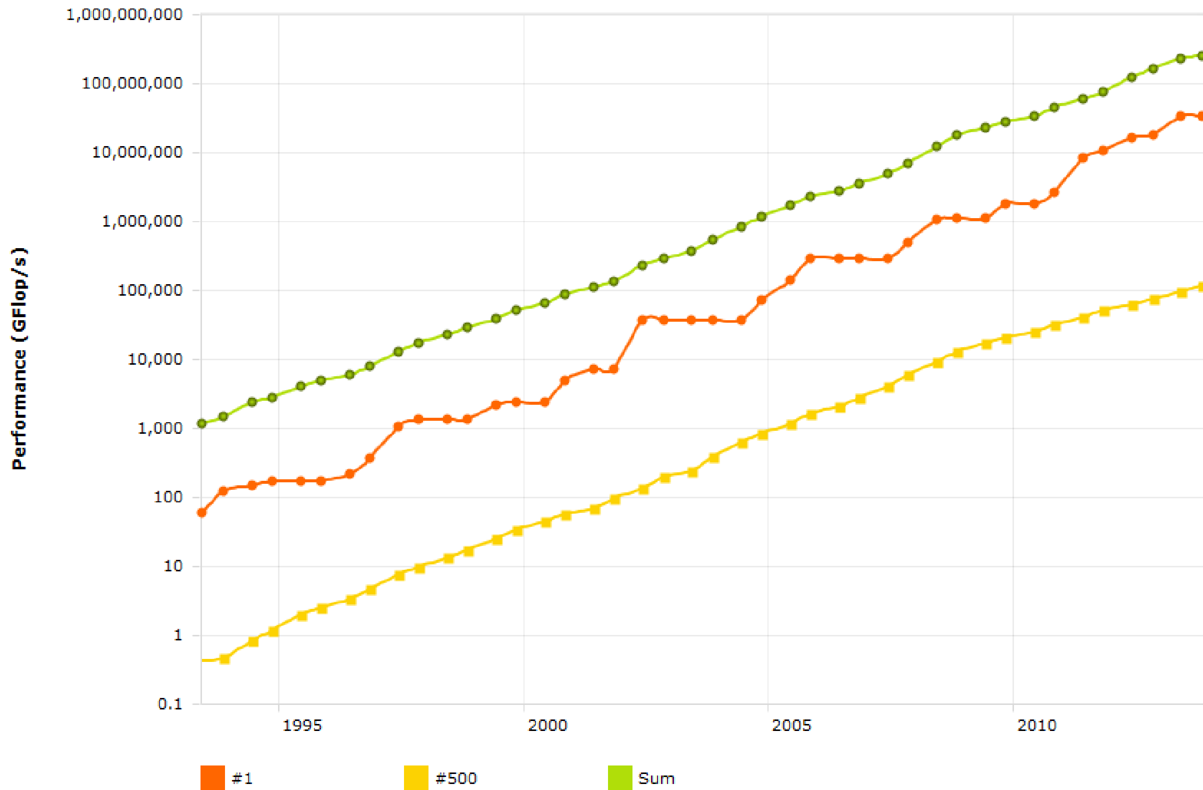


Figure 7. *Exponential growth of supercomputing power as recorded by the TOP500 [6].*

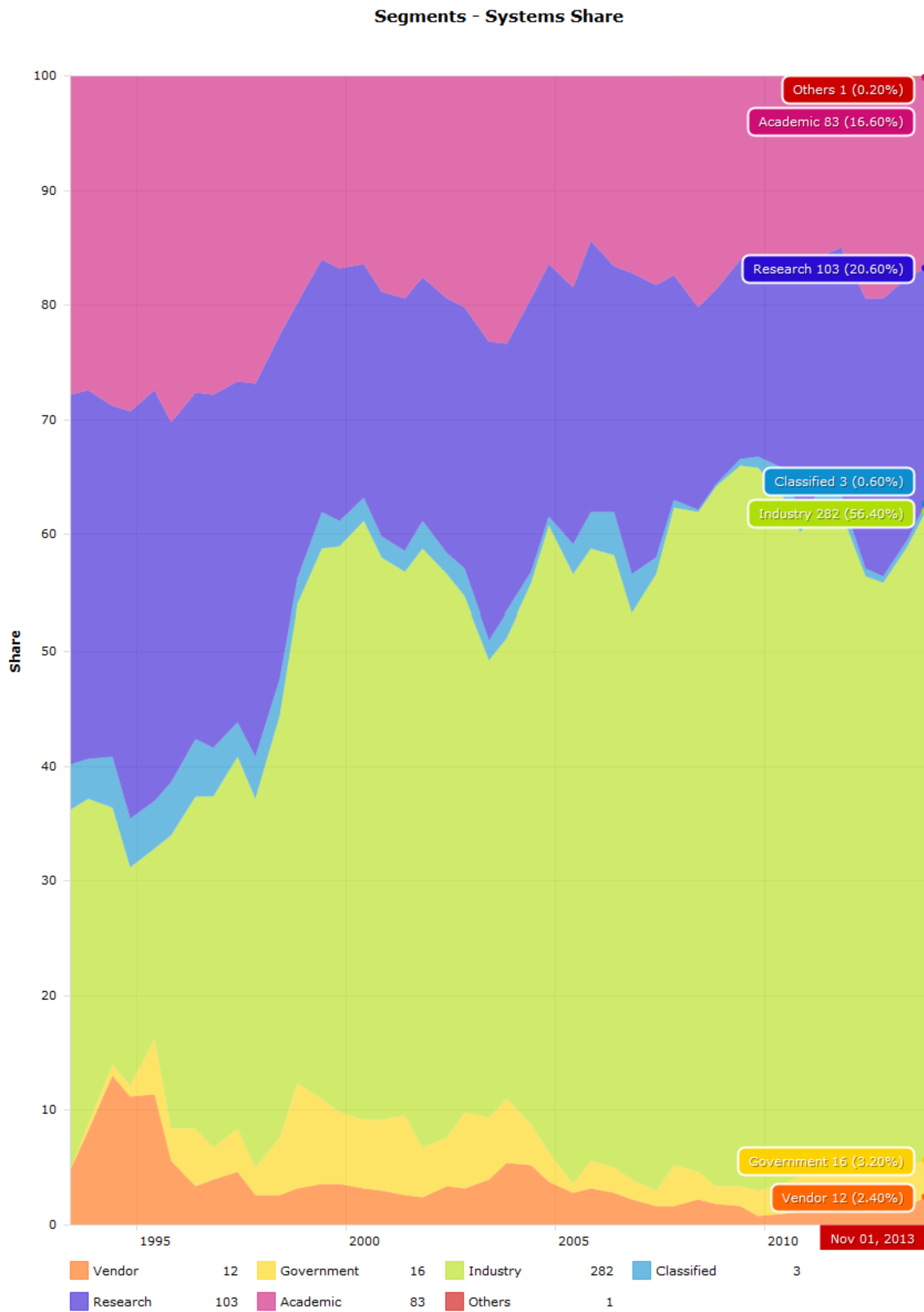


Figure 8. *Systems share through last two decades has moved from academy and research to industry, as recorded by the TOP500 [6].*

1.1 Concepts and terminology

General requirements of a computer were described by the work of John Von Neumann in first half of XX century. Since then, virtually every computer has followed the same basic design, differing from earlier computers which were programmed through “hard wiring”. The main layout of a computer comprises a memory, a control unit, an arithmetic logic unit and an input/output interface. Readable and writeable RAM memory is used to store both program instructions and data. In particular, program instructions are coded data which tell the computer to perform some operations, while data is the information to be processed. The control unit fetches instructions and data from the memory, decodes the instructions and then sequentially coordinates operations, in order to accomplish the programmed task. Each basic operation is performed by the arithmetic logic unit. The interface to human operator or to peripherals allows for an input and an output data exchange. Also in the case of a parallel computer, the basic fundamental architecture is the same for the different units.

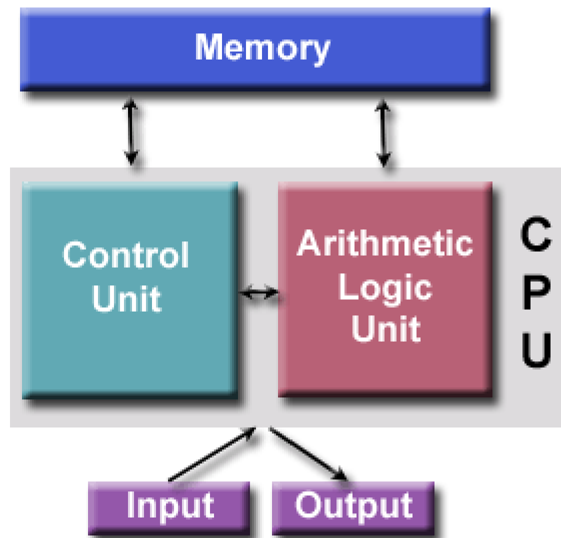


Figure 9. *Fundamental architecture of a CPU.*

There are different ways to classify parallel computers [1]. Parallel computers can be characterized according to the data and instruction streams, forming various types of computer organizations. They can also be classified in relation to the computer structure, i.e. multiple processors having separate memory or one shared global memory. Furthermore, parallel processing levels can also be defined in relation to the granularity in a program, that is the dimension of blocks of instructions, called grain size. Thus, a classification of parallel computers can be identified:

1) Classification based on the instruction and data streams
2) Classification based on the structure of computers
3) Classification based on granularity

1.1.1 Classification based on instruction and data streams

In the first classification, known as Flynn's Classical Taxonomy, a behavioral concept is discussed [1]. The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In particular, in the complete cycle of instructions execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called instruction stream. Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called data stream. These two types of streams are shown in Figure 10.

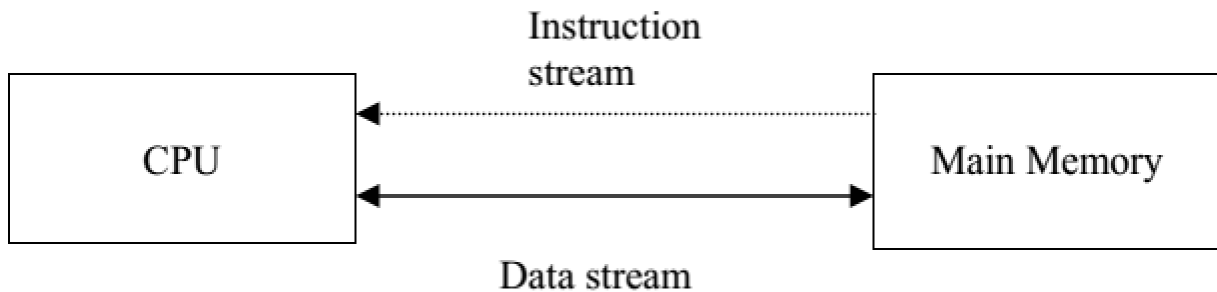


Figure 10. *Instruction stream is established from main memory to the CPU. A bi-directional data stream is established between processor and memory.*

Multi-processor computer architectures are distinguished according to how they can be classified along the two independent dimensions of Instruction Stream and Data Stream. Each of these dimensions can have only two states, Single or Multiple (Figure 11).

S I S D **Single Instruction Stream** **Single Data Stream**	**S I M D** **Single Instruction Stream** **Multiple Data Stream**
M I S D **Multiple Instruction Stream** **Single Data Stream**	**M I M D** **Multiple Instruction Stream** **Multiple Data Stream**

Figure 11. *Flynn's Classical Taxonomy.*

A SISD computer is a serial computer, in which only one instruction stream is acted on by the CPU during any clock cycle, and only one data stream is being used as input during any clock cycle (Figure 12). The execution of the operations is deterministic.

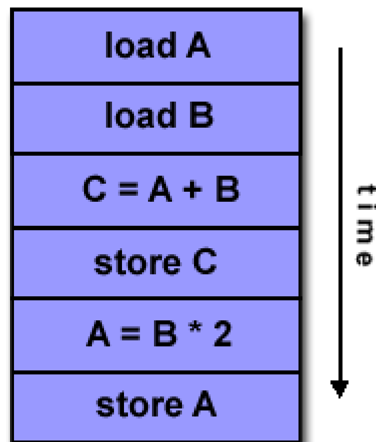


Figure 12. *Execution of instructions in a SISD computer.*

A SIMD computer is a parallel computer, in which all processing units receive the same instruction broadcast from the control unit. As in Figure 13, the same instruction at a given clock cycle is executed simultaneously by each processing unit on a different data element. Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Instruction execution is deterministic because it is synchronized, as all executions are lockstepped together. This kind of computers are best suited for specialized problems characterized by a high degree of regularity, such as computations involving dense matrices (Figure 14). Two kind of computers fall within this category: processor arrays, as GPUs, and vector pipeline systems.

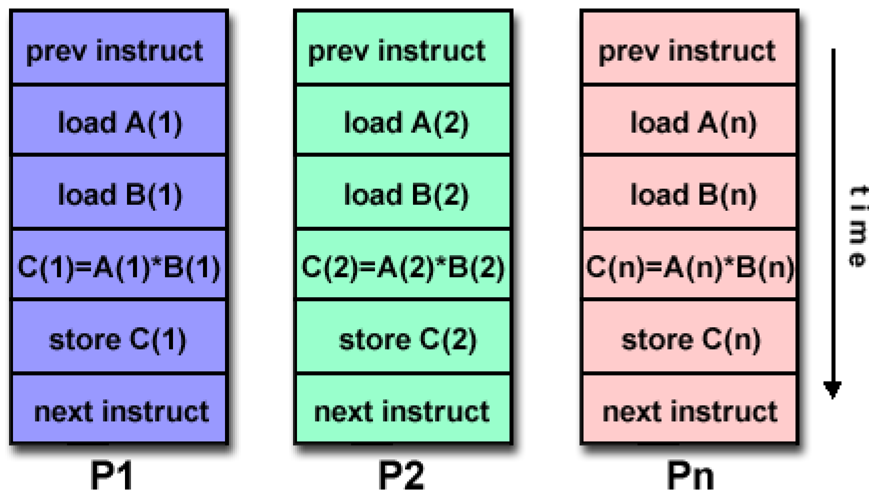


Figure 13. *Execution of instructions in a SIMD computer.*

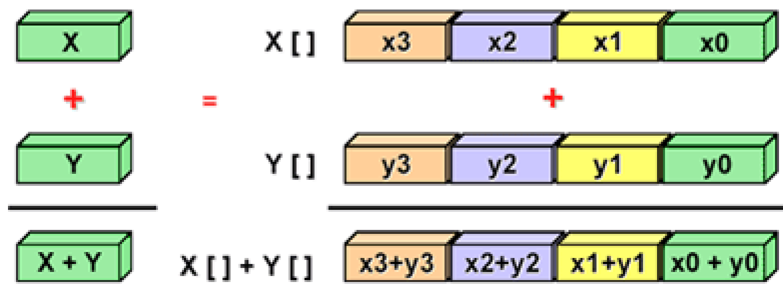


Figure 14. *Example of a vectorized sum on a SIMD computer.*

A MISD computer is a parallel computer, in which each processing unit acts on the same data independently, via separate instruction streams. This kind of computer has a very limited use, as cryptography.

Finally, in a MIMD computer every processor may be executing a different instruction stream and working with a different data stream (Figure 15). Executions could be either synchronous or asynchronous, deterministic or non-deterministic. Currently the most modern type of supercomputers fall in this category, i.e. networked parallel computer clusters and "computing grids", multi-processor computers, multi-core computers, cloud computing network.

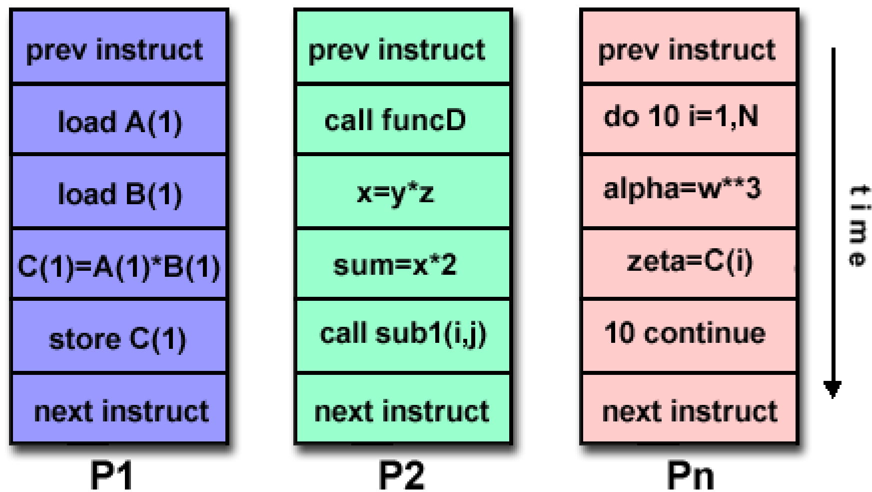


Figure 15. *Execution of instructions in a MIMD computer.*

1.1.2 Classification based on computer structure

In the second classification, the computer structure is considered. As seen above, a parallel computer (MIMD) can be characterized as a set of multiple processors with a shared memory, or memory modules communicating via an interconnection network. When multiprocessors communicate through a global shared memory, the system is a shared memory computer (Figure 16).

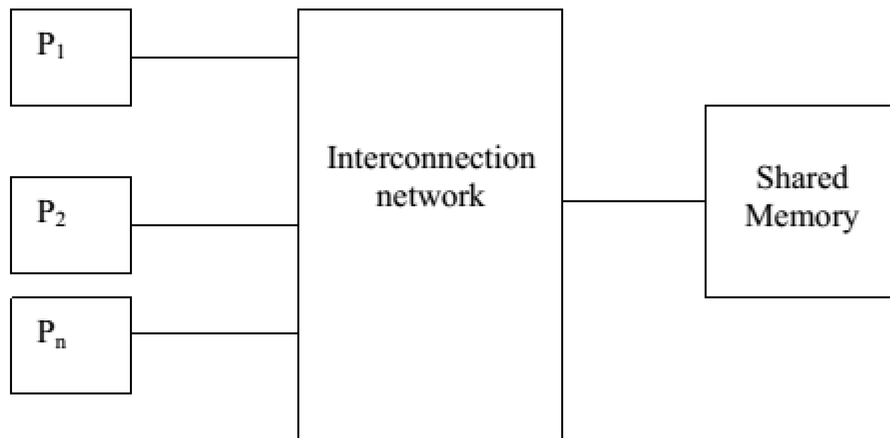


Figure 16. *Layout of a shared memory system.*

Shared memory multiprocessors show the following characteristics:

- Every processor communicates through a shared global memory
- For high speed real time processing, these systems are preferable as their throughput is high as compared to distributed memory computers.
- The processors have also access to I/O devices
- The inter-communication between processors, memory and other devices are implemented through various interconnection networks (Figure 17):
    - Processor-Memory Interconnection Network: this is a switch that connects various processors to different memory modules. Different strategies are adopted in order to connect every processor to every memory module. There can be a conflict among processors when they attempt to access the same memory modules. Even if this conflict is managed by the switch, the access to memory is delayed, i.e. leading to a bottleneck for computations.
    - Input-Output Processor Interconnection Network: this interconnection network is used for communication between each processors and I/O channels, i.e. a hard disk memory.
    - Interrupt Signal Interconnection Network: when a processor needs to send an interruption to another processor, the request is sent to this switch and delivered to the destination processor. In this way, synchronization between processors is ensured. Moreover, in case of failure of one processor, the switch can broadcast the message to other processors.
    - Every reference to the memory in shared memory computers is managed via interconnection network, thus there is a delay in executing the instructions. In order to reduce this delay, every processor may use cache memory for the frequent references made by the processor. Cache memory is a smaller, faster memory which stores copies of the data from frequently used global memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to global memory latency, i.e. leading to faster computations.

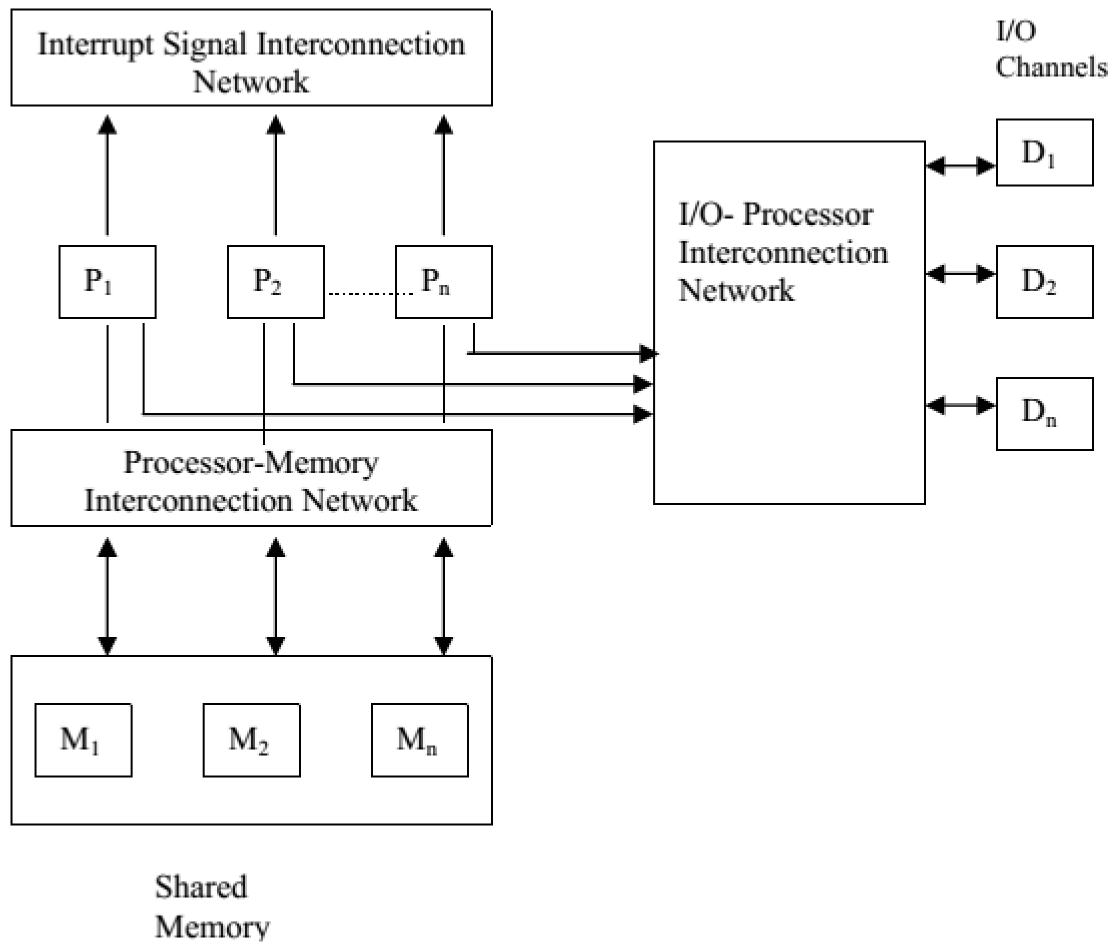


Figure 17. *In shared memory systems, processor-memory communication bandwidth and latency determines the speed of a computation.*

Otherwise, when every processor in a multiprocessor system manages its own local memory and each processor communicate with others via messages transmitted between local memories, the system is called distributed memory computer (Figure 18).

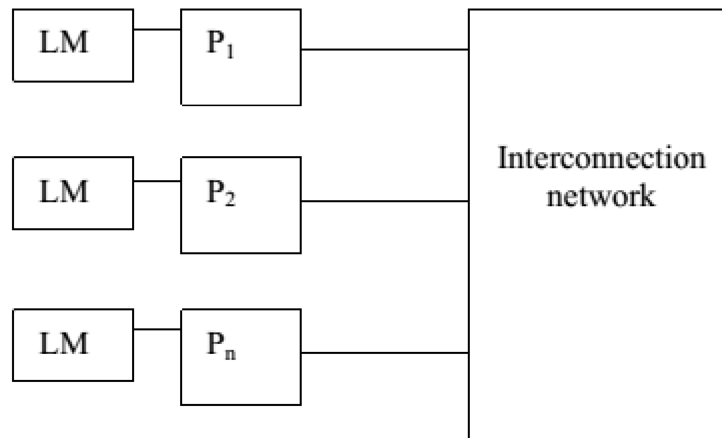


Figure 18. *Layout of a distributed memory system.*

Distributed memory computers do not share the global memory and the problem of memory conflicts, which slows down the execution of instructions in shared memory systems, is avoided. Such systems have multiple processors with their own large local memory and a set of I/O devices. This set of processor, memory and I/O devices is a computer system by itself, called node. Thus, distributed memory computers are also called multicomputer systems. The nodes are connected together via message passing interconnection network, through which processes communicate by passing messages each other. Every node in a multicomputer system owns a separate memory, the system is called distributed memory computer. Since local memories are accessible to the attached processor only, no processor can access other memories remotely. Therefore, message passing interconnection network provides connection to every node and inter-node communication with message depends on the type of interconnection network. For example, interconnection network for a non-hierarchical system could be a shared bus.

1.1.3 Classification based on grain size

The third classification is based on granularity, i.e. the amount of computation in relation to communication. Fine-grained parallelism means that individual tasks are relatively small in terms of code size and execution time. The data is transferred among processors frequently, in amounts of one or a few memory words. Conversely, in a coarse-grained parallelism, data is communicated occasionally, after larger amounts of computation.

To recognize different level of parallelism in a program to be executed on a multiprocessor system can be tricky. The idea is to identify the sub-tasks or instructions in a program that can be executed in parallel. For example, in Figure 19 there are three statements in a (sequentially executed) program and statements S1 and S2 can be exchanged. This means that an execution of S1 and S2 in parallel would not change program output.

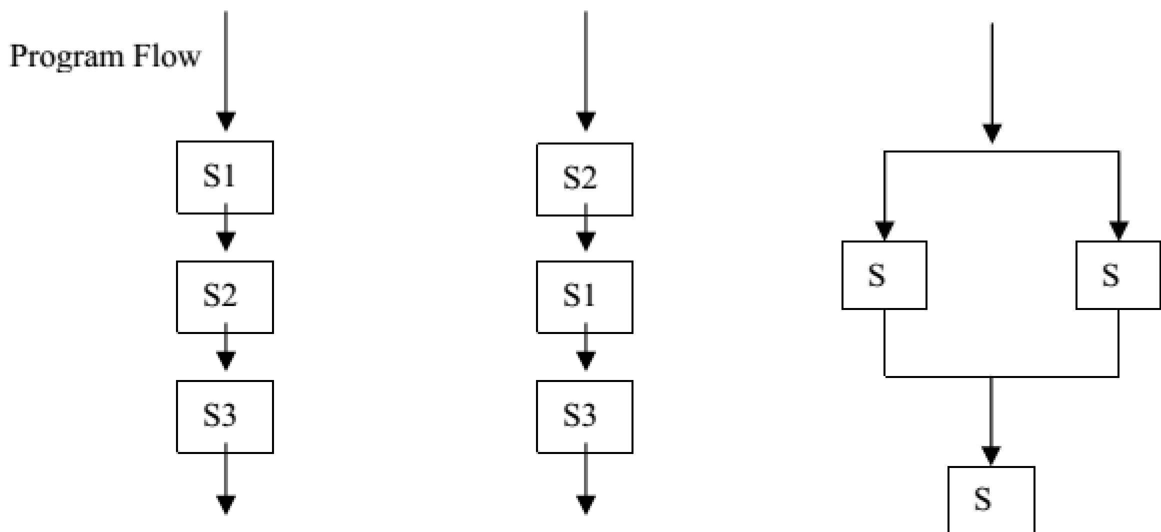


Figure 19. *Parallel execution of S1 and S2 blocks of instructions.*

On these bases, parallelism can be classified at various levels in a program. These parallelism levels form a hierarchy according to which, lower the level, the finer is the granularity of the process. The degree of parallelism decreases with increase in level. Every level according to a grain size demands communication and scheduling overhead, as it is summarized in Figure 20.

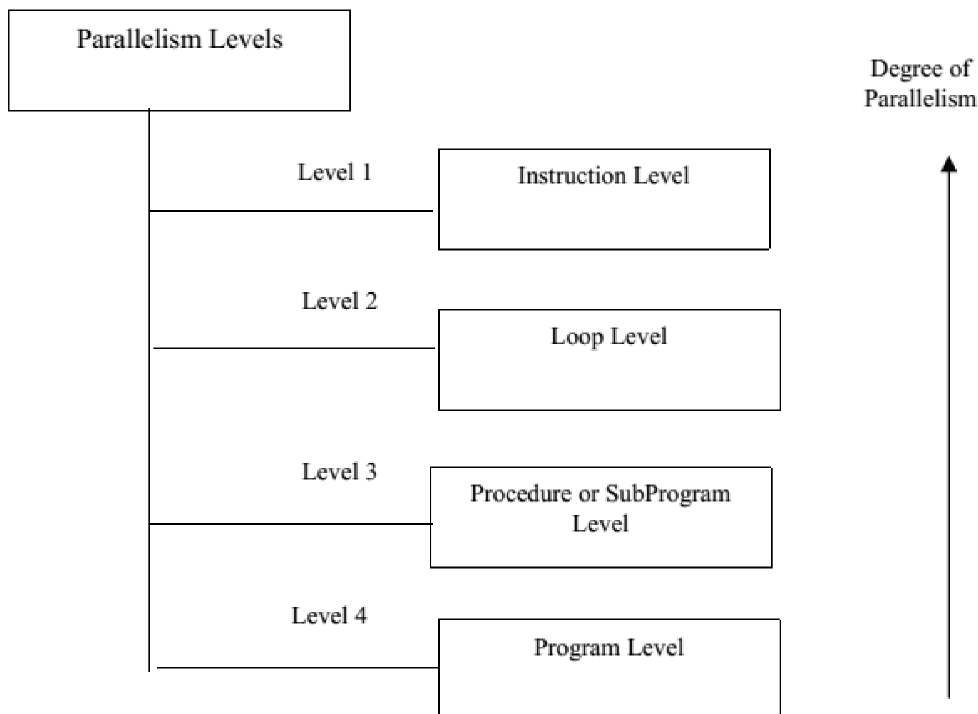


Figure 20. *Parallelism levels in a program.*

The instruction level is the lowest level and the degree of parallelism is the highest. The fine grain size is used at instruction or statement level as only few instructions form the grain size here. The fine grain size may vary according to the type of the program. For example, in scientific applications, the instruction level grain size could be higher. As a higher degree of parallelism can be achieved at this level, the overhead for a programmer will be more.

The loop level is another level of parallelism, where iterative loop instructions can be parallelized. Fine grain size is also used at this level. Simple loops in a program are easy to parallelize whereas the recursive loops are difficult. This type of parallelism can be achieved through compilers.

The procedure level consists of independent subroutines. Medium grain size is used at this level containing some thousands of instructions in a procedure. Multiprogramming is implemented at this level. Parallelism is exploited by programmers, but not through compilers.

The program level consists of independent programs. Coarse grain parallelism is used at this level. Time sharing is achieved through the operating system.

Traditionally, coarse grain parallelism is implemented in shared memory multiprocessors, while distributed memory computers are used to execute medium grain program segments. Fine grain parallelism is exploited in SIMD computers, as multicores or manycores architectures.

However, it is not sufficient to check for the parallelism between statements or processes in a program. The identification of possible parallelism and, eventually, the decision about which kind of parallelism would give the best performance, also depends on the number and types of processors available, the memory organization and simultaneous need for same data, control and resources by different processes, as it will pointed out in the next chapter [1].

1.2 Parallelism conditions

As discussed above, parallel computing requires that the segments to be executed in parallel must be independent of each other [1]. Therefore, for a correct parallel execution, all the conditions of parallelism between the segments must be analyzed. Three types of dependencies can be recognized, as in Figure 21.

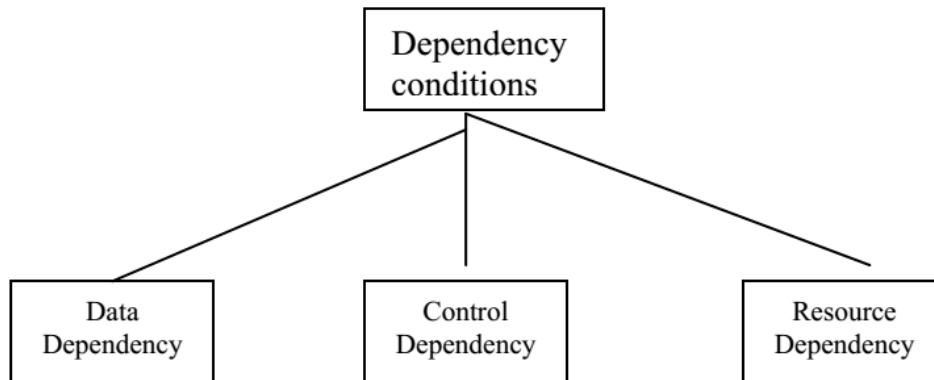


Figure 21. *Dependency conditions. All the conditions of parallelism between the segments must be analyzed for a correct parallel execution.*

Data dependency refers to the situation in which two or more instructions share the same data. The instructions in a program can be arranged based on the relationship of data dependency. In other words, the programmer should concentrate on how two instructions or segments are data dependent on each other. Four types of data dependencies are recognized:

- Flow dependence: if instruction 2 follows instruction 1, and output of 1 becomes input of 2, then 2 is "flow dependent" on 1
- Anti-dependence: when instruction 2 follows 1 such that output of 2 overlaps with the input of 1 on the same data
- Output dependence: when output of the two instructions 1 and 2 overlap on the same data, the instructions are said to be output dependent
- I/O dependence: when read and write operations by two instructions are invoked on the same file

The following program instructions are an example of data dependency:

S1: $a = b$

S2: $c = a + d$

S3: $a = c$

S1 and S2 are flow dependent, because variable a is generated by S1 as output, and it is used by S2 as input. Instructions S2 and S3 are anti-dependent because variable a is generated by instruction S3 but used by S2 and in sequence S2 comes first. Instruction S3 is flow dependent because of variable c. Instructions S3 and S1 are output dependent because variable a is generated by both instructions.

A control dependence could arise when instructions or segments in a program contain control structures. Therefore, dependency among the statements can be in control structures also, but the order of execution in control structures is not known before the run time. In this case, control structures dependency among the instructions must be analyzed carefully. As an example, the successive iterations in the following control structure (Figure 22) are dependent on one another.

```
For (i = 1; i ≤ n ; i + +)
 {
   if (x[i − 1] == 0)
         x[i] = 0
   else
         x[i] = 1;
 }
```

Figure 22. *Example of control dependency.*

Parallelism between the instructions may also be affected by conflicts due to shared resources. In particular, if two instructions are using the same shared resource, then it is a resource dependency condition. The contention of memory, caches, buses, registers, ports, and functional units could lead to runtime errors and incorrect results.

1.3 Amdahl-Gustafson's law

Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized [7]. Amdahl's law is used to evaluate the maximum expected improvement, called speedup, to an overall algorithm when a part of the algorithm is parallelized (Figure 24). In this situation, the speedup of a program using multiple processors is limited by the time needed for the sequential fraction of the program. Amdahl's law was initially intended for the special case of using n processors in parallel, as an argument for the single-processor approach's validity for achieving large-scale computing capabilities. Amdahl assumed that a fraction f of a program execution time was infinitely parallelizable without any scheduling overhead, while the remaining fraction $(1 - f)$ was totally sequential. The modern version of Amdahl's law states that, when an enhancement of a fraction f of the computation leads to a speedup S, the overall speedup is:

$$Speedup\ (f, S) = \frac{1}{\left((1-f)+\frac{f}{S}\right)}$$

Amdahl's law lead to important conclusions, such as:

- when f is small, optimizations will have little effect, i.e. some problems are difficult to parallelize

- as S approaches infinity, speedup is bound by $1/(1-f)$, i.e. there are other factors that limit the speedup

For example, if a program needs 20 hours using a single processor, and a particular portion of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (95% of sequential execution time) can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence the speedup is limited up to 20.

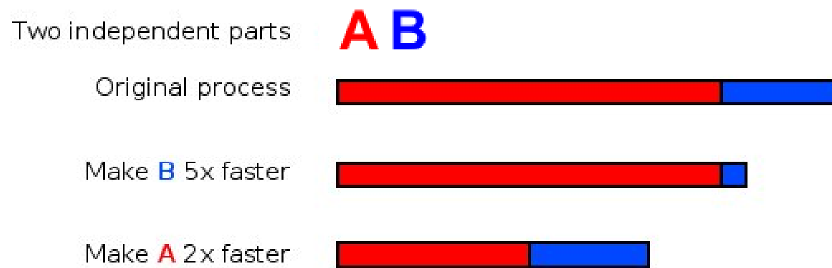


Figure 23. *A program should be analyzed carefully before planning a parallelization of its code. In this example, the B part performs much better in parallel executions than A, however its contribute to the overall speedup is low.*

Amdahl's law represent the law of diminishing returns about the return of adding more processors to a machine, when running a fixed-size computation that will use all available processors to their capacity. Each new processor, added to the system, will carry less usable computing power than the previous one. If the number of processors were doubled, the speedup ratio would diminish, as the total throughput heads toward the limit $1/(1-f)$. This analysis neglects other potential bottlenecks such as memory bandwidth and I/O bandwidth, that usually are not scaling with the number of processors.

Nevertheless, typically in scientific applications, a common user will not be interested in solving a fixed problem in the shortest possible interval of time, as Amdahl's law describes, but rather in solving the largest possible problem (Gustafson’s law), that is, the most accurate possible approximation in a fixed reasonable amount of time. If the non-parallelizable portion of the problem is fixed, or grows very slowly with problem size, then additional processors let to increase the maximum limit for the problem size.

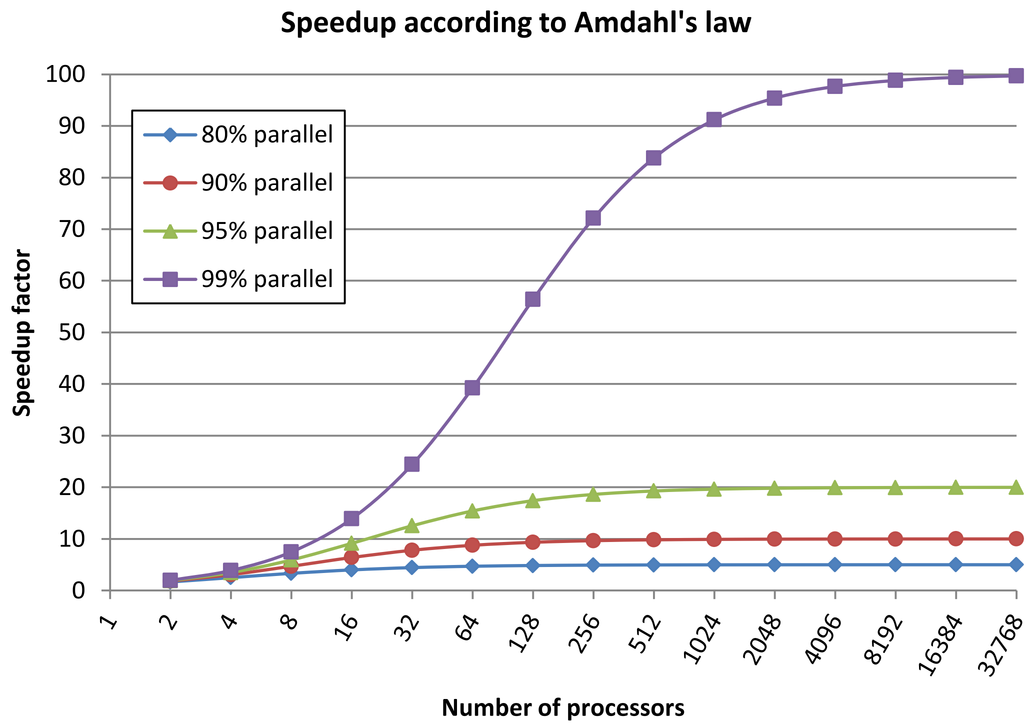


Figure 24. *Speedup according to Amdahl's law.*

2. Parallelization in shared memory computers

The shared memory model is the most used and easiest to grasp method for inter-thread communication [8]. Its main attractiveness is that it does not require major re-thinking of program structure, as it can be regarded as a natural extension of single-threaded applications. The concept itself is simple and straightforward: each thread that need to communicate holds a reference to a location in the memory that can be used to place information into; then, the same memory can be accessed by any other thread holding a reference to the same memory location. Most shared memory systems provide the appearance of a single memory space and a uniform computing environment, where any processor can access any memory location in the system.

There are however two fundamental issues with shared memory: in order to maintain the consistency of the memory content, access to the memory must be policed and all threads need to synchronize; for the same consistency reasons, local copies of the shared memory held in processor caches must be synchronized and updated, while the hardware must provide mechanisms for keeping the logical ordering of memory writes, the primary focus of memory consistency models.

The first issue, concerning the access policies to shared memory, relies entirely in the programmer ability. Traditionally, locks and/or semaphores were used, with a synchronization object associated with each logically related group of shared memory areas; any thread requiring to access the shared memory would need to acquire the ownership of the synchronization object before accessing the shared memory area.

The second issue, which concerns cache coherence and hardware ordering, is usually solved entirely in hardware through the implementation of memory consistency and cache coherence and validation protocols; however, these will have impact on software performance: the number of processor cycles needed for keeping the caches coherent and the size of the logic required to implement it (consequently, the power consumption) will increase with the number of cores that are concurrently accessing the same memory location. Optimizing for cache behavior requires extremely good understanding of the underlying hardware and often leads to application level solutions that are not immediately obvious, such as ‘un-natural’ partitioning or sizing of data or bundling of memory updates.

2.1 OpenMP

Shared-memory machines existed for a long time. In the past, each vendor developed its own ”standard” compiler directives and libraries, which allowed for a program to exploit advanced capabilities of their specific parallel machine. A standardization was not formally adopted, since no strong support of the vendors was existing and distributed memory machines, with their own

more standard message passing libraries PVM and MPI, appeared as a good alternative to shared-memory machines [9].

In 1996-1997, a new interest in a standard shared-memory programming interface appeared, mainly due to a renewed interest from the vendors side in shared-memory architectures, and the opinion by a part of the vendors that the parallelization of programs using message passing interfaces appeared cumbersome and long and that a more abstract programming interface would be desirable. OpenMP is the result of a large agreement between hardware vendors and compiler developers and is considered to be an "industry standard". The designers of OpenMP desired to provide an easy method to thread applications without requiring that the programmer know how to create, synchronize, and destroy threads or even requiring to determine how many threads to create. To achieve these ends, the OpenMP designers developed a platform-independent set of compiler pragmas, directives, function calls, and environment variables that explicitly instruct the compiler how and where to insert threads into the application. Most loops can be threaded by inserting only one pragma right before the loop. By leaving the details to the compiler and OpenMP, more time is left for determining which loops should be threaded and how to best restructure the algorithms for maximum performance. In practical cases, the maximum performance of OpenMP is realized when it is used to thread "hotspots", the most time-consuming loops in your application.

OpenMP also addresses the inability of previous shared-memory directive sets to deal with coarse-grain parallelism. Indeed, the former limited support for coarse grain work led to developers to think that shared-memory parallel programming was essentially limited to fine-grain parallelism.

As mentioned before, OpenMP is a collection of compiler directives, library routines and environment variables meant for parallel programming in shared-memory machines. The most important directive in OpenMP is the one in charge of defining the so called parallel regions. Such a region is a block of code that is going to be executed by multiple threads running in parallel. Since a parallel region needs to be created/opened and destroyed/closed, two directives are necessary, forming a so called directive-pair !$OMP PARALLEL/!$OMP END PARALLEL.

An example of their use is reported in Figure 25.

```
!$OMP PARALLEL
        write(*,*) "Hello"
!$OMP END PARALLEL
```

Figure 25. *Example of a parallel region. Each thread execute the same instruction.*

Since the code enclosed between the two directives is executed by each thread, the message *Hello* appears in the screen as many times as threads are being used in the parallel region. Before

and after the parallel region, the code is executed by only one thread, which is the normal behavior in serial programs. Therefore it is said, that in the program there are also so called serial regions. When a thread executing a serial region encounters a parallel region, it creates a team of threads, and it becomes the master thread of the team. The master thread is a member of the team and takes part in the computations. Each thread inside the parallel region gets a unique thread number which ranges from zero, for the master thread, up to $Np - 1$, where Np is the total number of threads within the team. In Figure 26, the previous example is represented in a graphical way to clarify the ideas behind the parallel region constructor. At the beginning of the parallel region it is possible to impose clauses which fix certain aspects of the way in which the parallel region is going to work: for example the scope of variables, the number of threads, special treatments of some variables.

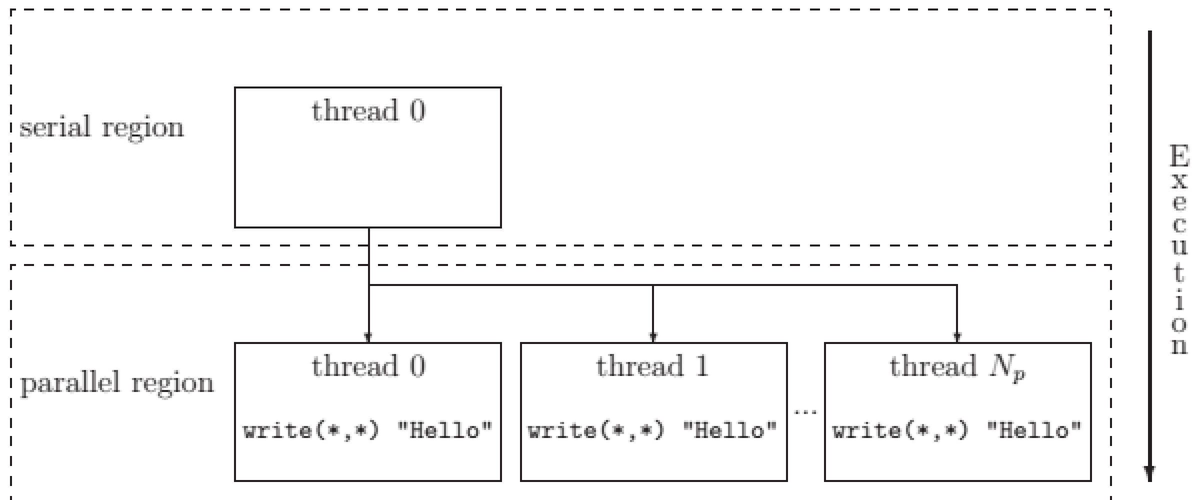


Figure 26. *Graphical representation of the example in Figure 25.*

When the !$OMP END PARALLEL directive is reached, all the variables declared as local to each thread are erased and all the threads are killed, except for the master thread which continues the execution of the program. It is necessary that the master thread waits for all the other threads to finish their work before closing the parallel region, otherwise information would get lost and work would not be done. This waiting is an implicit synchronization between the parallel running threads.

Data dependencies exist when different iterations of a loop read or write shared memory, and more specifically, a thread reads or modifies data that should be used by another thread [10]. Consider the example in Figure 27, that calculates factorials.

```
#pragma omp parallel for
for (i = 2; i < 10; i++)
{
```

$$factorial[i] = i * factorial[i-1];$$
}

Figure 27. *Example of data dependency in OpenMP parallel loops. Compilation fails because each loop iteration writes a value that a different iteration reads.*

The compiler will thread the loop, but it will fail because at least one of the loop iterations is data-dependent upon a different iteration. This situation is referred to as a race condition, and can occur only when using shared resources, i.e. memory, in a parallel execution. To address this problem it is necessary to rewrite the loop or, in worse cases, pick a different algorithm that does not contain the race condition.

Load balancing, the equal division of work among threads, is among the most important attributes for parallel application performance. Load balancing is extremely important, because it ensures that the processors are busy most, rather all, of the time. Without a balanced load, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities.

Within loop constructs, poor load balancing is usually caused by variations in compute time among loop iterations. It is usually easy to determine the variability of loop iteration compute time by examining the source code. In most cases, you will see that loop iterations consume a uniform amount of time. When that's not true, it may be possible to find a set of iterations that consume similar amounts of time. For example, sometimes the set of all even iterations consumes about as much time as the set of all odd iterations. Similarly it can happen that the set of the first half of the loop consumes about as much time as the second half. On the other hand, it may be impossible to find sets of loop iterations that have a uniform execution time. Regardless of which case an application falls into, you should provide this extra loop scheduling information to OpenMP so that it can better distribute the iterations of the loop across the threads (and therefore processors) for optimum load balancing [10].

OpenMP, by default, assumes that all loop iterations consume the same amount of time. This assumption leads OpenMP to distribute the iterations of the loop among the threads in roughly equal amounts and in such a way as to minimize the chances of memory conflicts due to false sharing. This is possible because loops generally touch memory sequentially, so splitting up the loop in large chunks, as the first half and second half in case of two threads, will result in the least chance for overlapping memory. While this may be the best choice for memory issues, it may be bad for load balancing. Unfortunately, the reverse is also true; what may be best for load balancing, may be bad for memory performance. Therefore, programmers must find a balance between optimal memory usage and optimal load balancing by measuring the performance through profiler tools, to see which method produces the best results [10].

By summarizing, OpenMP threaded application performance is largely dependent upon:

- The underlying performance of the single-threaded code
- CPU utilization, idle threads and poor load balancing
- The percentage of the application that is executed in parallel
- The amount of synchronization and communication among the threads
- The overhead needed to create, manage, destroy, and synchronize the threads, in particular by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions
- Performance limitations of shared resources such as memory, bus bandwidth and CPU execution units
- Memory conflicts caused by shared memory or false sharing.

2.2 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) are a set of low-level kernel subroutines that perform common linear algebra operations such as copying, vector scaling, vector dot products, linear combinations, and matrix multiplication [11,12]. They were first published as a Fortran library in 1979 and are still used as a building block in higher-level math programming languages and libraries.

The BLAS routines and functions are divided into the following groups according to the operations they perform:

- BLAS Level 1 Routines perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- BLAS Level 2 Routines perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- BLAS Level 3 Routines perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software. Furthermore, most of the BLAS library available via vendors are multithreaded and thread-safe [13]. In other words, they exploit parallelization on shared memory computers, i.e. through OpenMP programming, and the parallel code is correctly executed by each thread.

3. Parallelization in distributed memory computers

3.1 Message passing paradigm

The message passing paradigm is commonly considered to be the opposite of the shared memory model, as it usually relies on a “share nothing” approach [8]. In a message passing based system, threads have no shared memory areas, and all information exchange happens through messages that are passed between the threads. Message passing is usually considered the foundation for any thread isolation solution: shared memory means that if one thread fails, also the other threads may fail, as the content of the shared memory could be corrupted. A similar situation will not occur in a message passing based system, as threads can protect themselves through analysis and possible discarding of erroneous messages.

3.2 MPI – Message Passing Interface

High performance, scalability and portability features make MPI to be the standard message-based communication protocol in high performance computing in distributed memory systems. It consists of a language-independent communication protocol by exploiting both point-to-point and collective communication. The MPI standard (MPI-1, MPI-2 and MPI-3 versions) defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message passing programs in Fortran and C. MPI provides a clearly defined base set of routines that can be efficiently implemented by parallel hardware vendors. Therefore, upon this collection of standard low-level routines, hardware vendors can create higher-level routines for the distributed-memory communication environment supplied with their parallel machines. MPI provides a simple-to-use portable interface for the basic user, yet powerful enough to allow programmers to use the high-performance message passing operations available on advanced machines. The message passing paradigm is attractive because of its wide portability and can be used in communication for distributed-memory computers, shared-memory multiprocessors systems, networks of workstations, and a combination of these elements. The paradigm is applicable in multiple settings, independent of network speed or memory architecture [14].

Even if the first MPI-1 model did not implement any shared memory concept, and MPI-2 brings only a limited “distributed” shared memory approach, MPI programs are regularly run on shared memory computers. Furthermore, designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality.

The MPI interface provides essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings) and a few language-specific features. MPI programs always work with processes, but programmers

commonly refer to the processes as processors. Typically, for maximum performance, a single process will be assigned to each CPU (or each core in a multi-core machine). This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

MPI library functions include, but are not limited to, point-to-point "rendezvous-type" send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes accessible in a logical topology, and so on. Point-to-point operations come in synchronous, asynchronous, buffered, and ready forms, to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send. Many operations are possible in asynchronous mode, in most implementations.

MPI-1 and MPI-2 both enable implementations that overlap communication and computation. MPI also specifies thread safe interfaces, which have cohesion and coupling strategies that help avoid hidden state within the interface. It is relatively easy to write multithreaded point-to-point MPI code, and some implementations support such code.

MPI provides a rich range of functionalities. The following concepts help the programmer to decide what functionality to use in application programs:

- Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology
- A number of important MPI functions involve communication between two specific processes. An example is MPI_Send, which allows one specified process to send a message to a second specified process. Such point-to-point operations are particularly useful in patterned or irregular communication. For example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed
- Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast broadcast call. This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. For example, reduction operations are useful at the end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result

An example of basic MPI functions is given in Figure 28.

```
program hello
include 'mpif.h'
        integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
        call MPI_INIT(ierror)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
        print *, 'node', rank, ': Hello world'
        call MPI_FINALIZE(ierror)
end
```

Figure 28. *Example of MPI usage. Each process prints its name and the string "Hello world".*

3.2.1 MPICH library

MPICH is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard [15]. The goals of MPICH are:

- to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters (desktop systems, shared-memory systems, multicore architectures), high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet, Quadrics) and proprietary high-end computing systems (Blue Gene, Cray)
- to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations.

An MPI library typically requires thread services (e.g., thread creation, mutex locks), shared-memory services (for intranode communication), internode communication services (e.g., TCP/IP sockets), and OS process-management services. The APIs for these features can differ among operating systems and platforms. For portability, MPICH2 uses an internal abstraction layer for these services, which can be implemented selectively on different OS platforms [16].

The "CH" in name "MPICH" means Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems. There are two common ways to use MPI with multicore processors or multiprocessor nodes:

- Use one MPI process per core (recall, a core is defined as a program counter and some set of arithmetic, logic, and load/store units)
- Use one MPI process per node (recall, a node is defined as a collection of cores that share a single address space). Use threads or compiler-provided parallelism to exploit the multiple cores. OpenMP may be used in combination with MPI; for example, the loop-

level parallelism of OpenMP may be used with any implementation of MPI. This is sometimes called the hybrid programming model. MPICH automatically recognizes multicore architectures and optimizes communication for such platforms.

Before starting a parallel job, a process management system, as Simple Multipurpose Daemon (SMPD), must be configured on each host and each host must be connected into a ring. After that, all users of the network will be able to launch MPI jobs using mpiexec or a similar command. The simplest form of a command to start an MPI job is "$mpiexec - n\ 32\ hello.exe$".

4. State of the art of multicore systems

In 2001 the first general-purpose processor that featured multiple processing cores on the same CMOS die was released: the POWER4 processor of IBM. Since then, multi-core processors have become the norm and currently the only way to improve the performance for high-end processors is to add support for more threads, either in the number of cores, or through multithreading on cores to mask long-latency operations [8].

There are multiple reasons why the clock rate gains of the past cannot anymore be continued. The unsustainable level of power consumption implied by higher clock rates is just the most obvious and stringent reason; equally important is the fact that wire delays rather than transistor switching will be the dominant issue for each clock cycle.

Compared to single threaded processors, multi-core processors are highly different and the design space is enormous. The development of semiconductor technology let the integration of the idea of large supercomputers onto one single-chip multiprocessor, most often called a multi-core processor.

Current general-purpose multi-core processors are built on a homogeneous architecture. This means that the cores can execute the same binary code and that it does not really matter, from a functional point of view, on which core a program runs. Most of these homogeneous architectures also implement a shared global address space with full cache coherency. By contrast, a heterogeneous architecture features at least two different kinds of cores that may differ in architecture, functionality and performance.

All modern core designs are pipelined, where instructions are decoded and executed in stages in order to improve on overall throughput, although instruction latency is the same or even increased.

As the capabilities and speed of cores have been improving more rapidly than memory performance, the net result was that cores were idling for a significant share of the time while waiting on high latency memory operations to complete. This observation led to the implementation of hardware multi-threading, a mechanism through which a core could support multiple thread contexts in hardware (including program counter and registers, but sharing the cache) and fast switching between hardware threads whenever some of the threads stalled due to high latency operations. The usefulness of this technology also has its limit however. Recently, the gap between the speed of memory access and speed of cores started to narrow due to the decline in processor core frequencies; therefore latency-hiding techniques will yield smaller benefits. Considering the performance gain per watt, these techniques will be most likely replaced by other mechanisms, such as increased cache sizes.

Having multiple cores on a chip requires inter-core communication mechanisms. The historical way that the individual processors in a shared memory multiprocessor communicate has been through a common bus shared by all processors. This is indeed also the case in the early multi-core processors from general purpose vendors (Intel, AMD). In order not to flood the bus with memory and I/O traffic, there are local cache memories, typically one or two levels, between the processor and the bus. More recent designs are based on the realization that shared communication mediums such as buses are problematic both in latency and bandwidth. A shared bus has long electrical wires and if there are several potential slave units (as in a multicore processor all cores and memory sub-system are both slaves and masters) the capacitive load on the bus makes it even slower. Furthermore, the fact that several units share the bus will fundamentally limit the bandwidth from each core. As the number of cores will increase, on-chip communication networks will increasingly face scalability and power constraints.

In the high-performance computing area, shared memory has not been used since its current implementations do not scale to thousands of nodes used in the top-performing compute clusters. These clusters, however, are built out of shared memory nodes and although programmers may or may not use shared programming models within a node, the hardware of these nodes typically implement a shared address space. The best performing programs in these machines typically use a hybrid programming model using message-passing between nodes and a shared memory within a node. For these reasons, all general-purpose multi-core processors today support a shared address space between cores and maintain a cache-coherent memory system. By definition, a cache system is said to be coherent if and only if all processors, at any time, have a consistent view of what is the last globally written value to each location. The cache coherency mechanism (Figure 29 and Table 1) allows for a fast access to commonly used data in processor private caches while still maintaining consistency when some other processor updates shared variables. The most common implementation uses invalidation mechanisms where local copies are invalidated if a core updates a shared variable [8].

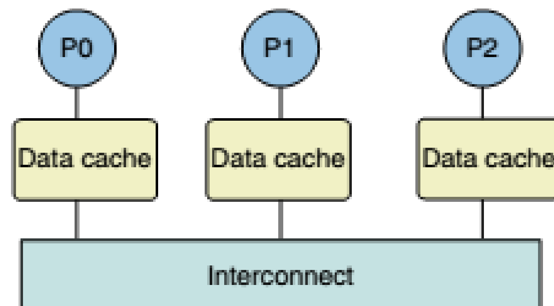


Figure 29. *Example of cache coherence mechanism. Initial state of block A is un-cached [8].*

Table 1. *Example of cache coherence mechanism, based on Figure 29 [8].*

Time	PO	P1	P2	Comments
T1	Load A	Load A		A copy in both 0 and 1 caches. The block is shared.
T2	Store A			The copy in cache 1 needs to be invalidated. The block is exclusive in cache 0.
T3	Load A			This load will succeed. No coherence action is needed.
T4		Load A		The newly updated value in cache 0 is transferred to cache 1. The block is shared. Memory could be updated, depending on cache coherence protocol used.
T5			Load A	Block is now also cached in cache 2.
T6		Store A		The ownership is transferred to P1 and the block is now exclusive in cache 1. The copies in both cache 0 and 2 are invalidated.

Furthermore, many transistors on the CPU are devoted to advanced communication and administration techniques like out-of-order execution and branch prediction [17,18]. In particular, branch prediction is a method used to guess the memory address of following instructions. With good guessing, a CPU can request the instructions from memory long before they are executed. This decreases the time the CPU wastes on waiting for data and instruction to arrive. Out-of-order execution is a queuing system for instructions on the CPU. Supported instructions have their respective resources with a physical location. If a processor is supporting out-of-order execution, instructions can line up for their resource and run as soon as it becomes available. The results are coordinated and synchronized automatically afterwards. Techniques of handling memory together with the support of task switching makes a CPU perform well when running operation systems.

Multicore systems will evolve in the near future towards the following concepts [8]:

- The next future is to start adding, though slowly, more cores to chip designs
- These cores should be complex and highly performing, in order to support single threaded applications
- Shared memory on hardware level will be enhanced by increasing the number of cores per chip
- The main issues to address will be memory access latencies, to be reduced through cache hierarchies and mechanisms that can improve efficiency of core usage.

4.1 Emerging technologies

After multi-core chips became a standard, several new issues emerged that required new solutions, while some existing, well-known problems have found innovative solutions. The problem of “memory wall”, initially defined as the gap between the speed of processors and the speed of memory access, slowly turned into a different problem: while the latency gap became smaller, with the increase in the number of cores, the need for memory bandwidth increased. Interconnections became another important issue: existing solutions became either too power requiring, as the transistor count raised up (the case of bus or ring solutions), or led to higher

latency (mesh networks), as the number of cores continues to increase. Finally, the availability of a larger number of cores calls into question the core usage efficiency, promoting the use of some of the cores (or at least hardware threads) as helpers [8].

There are essentially three different viewpoints followed by multi-core chip designs:

- fewer, but more complex, high performance cores with large, shared on-chip caches and cache coherence mechanisms; this design is represented by most server chip vendors (IBM, Intel, AMD and SUN) but also companies focusing on the mobile and low power design space
- large number of simple, low frequency, sometimes specialized cores with distributed on-chip memories (with or without cache coherence mechanisms) and scalable interconnections
- integration of multiple cores of different capabilities

The main argument for the first type of design is to keep support for single-threaded applications. Conversely, following the second approach, the overall performance per watt will be higher by using simpler but more cores. Finally, the third type of design usually targets specific domains where some measure of performance dominates all other factors.

Examples of processors based on low number of cores are Intel Server, IBM Power, SPARC and ARM processors. Examples of processors based on large number of cores are NVIDIA GPUs and Intel MIC.

4.1.1 Nvidia GPU

A Graphic Processing Unit is a specialized integrated circuit designed to manipulate and operate on graphical data, in order to accelerate the creation and rendering of images in a frame buffer, intended for output to a display. Modern GPUs has become very efficient at manipulating 3D computer graphics, and their highly parallel structure well fit algorithms where processing of large blocks of data can be done in parallel. In fact, operations as texture mapping, rendering of polygons, rotation and translation of geometries into different coordinate systems and video decoding, involve matrix and vector operations that can be effectively exploited in simulation software.

In contrast to a CPU, a GPU focuses mainly on floating-point multiplication and addition, together with the ability to execute thousands of threads in parallel. Memory is located physically closer to the processors (Figure 30, 31 and 32) and memory transfer has a large bandwidth within the graphics card (Table 2). However, GPUs do not integrate any system for choosing how the memory should be used for optimal performance [19]. Therefore, best performance can be achieved by programming specifically where and when data should be stored on the graphic card. The programmer also need to be aware of some hardware specifications, as parallel threads can only operate in blocks, by performing the same operation on different data. Each thread can operate only on its local memory, then a group of threads are grouped in a block, of dimension

related to the hardware configuration. Each block can operate on L1 cache. Similarly, a group of blocks called grid, can handle data in the global memory.

In conclusion, GPUs provide outstanding computing power on tasks characterized by a high degree of parallelism, but their arithmetic power results from a highly specialized architecture, and many applications exists for which GPUs are not well suited [8].

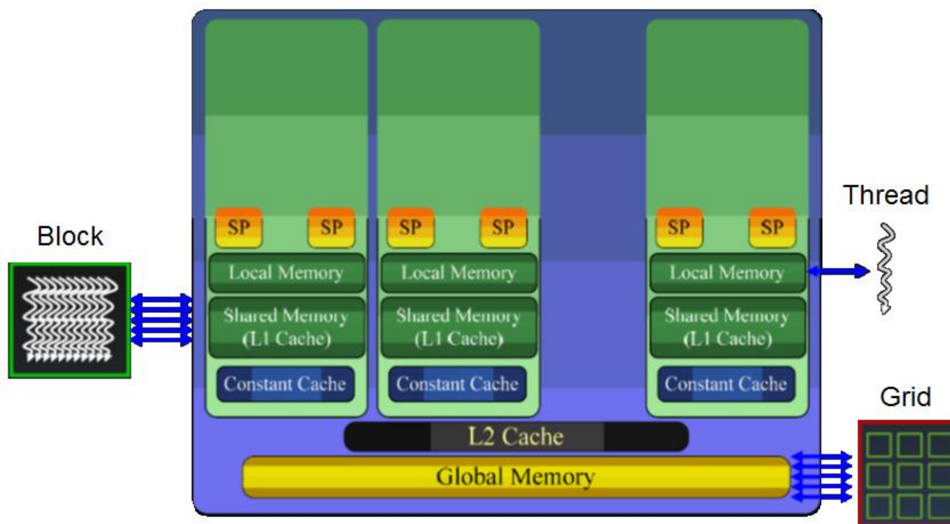


Figure 30. *GPU functional subdivision.*

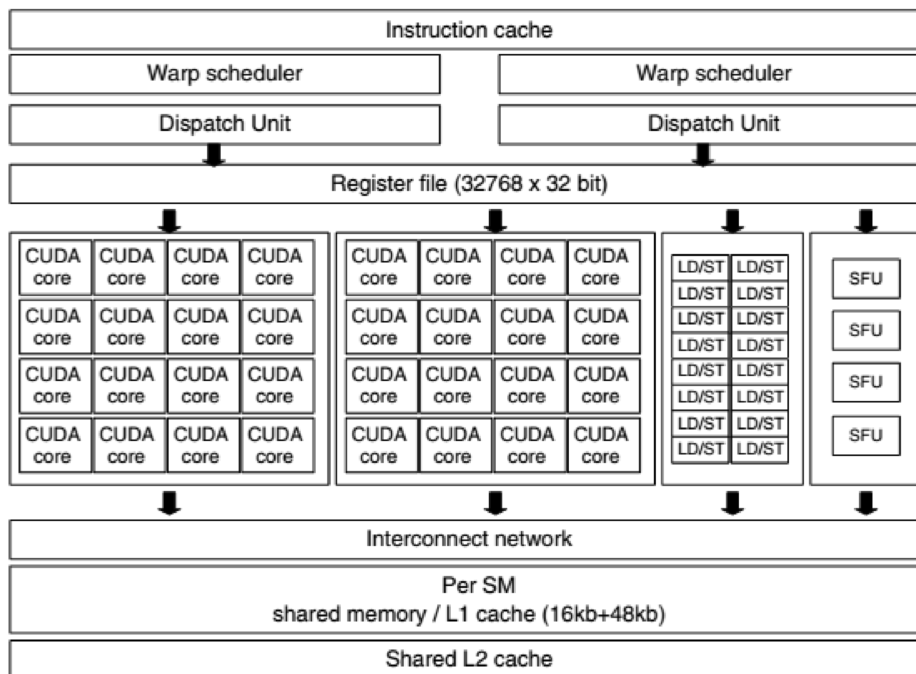


Figure 31. *Example of NVIDIA streaming multiprocessor architecture.*

Table 2. *Nvidia Tesla GPUs technical specifications. The same data on a recent Intel multicore processor are reported for reference [20,21].*

	Tesla C1060	**Tesla C2075**	**Tesla K20**	**Tesla K40**	**Intel Xeon E5-2687W v2**
Number of Streaming Processor cores (Scalar Processor)	240	448	2496	2880	8
Core frequency	1.3 GHz	1.15 GHz	706 MHz	745 MHz	3.4 GHz
Single Precision floating point performance (peak)	933 GFlops	1.03 TFlops	3.52 TFlops	4.29 TFlops	--
Double Precision floating point performance (peak)	78 GFlops	515 GFlops	1.17 TFlops	1.43 TFlops	108 GFlops
Floating Point Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754	
Total dedicated memory	4 GB	6 GB	5 GB	12 GB	256 GB
Memory Speed	800 MHz	1.5 GHz	2.6 GHz	3.0 GHz	1.86 GHz
Memory bandwidth (between cache and cores)	102 GB/s	144 GB/s	208 GB/s	288 GB/s	59.7 GB/s
Software Development Tools	C-based CUDA Toolkit	C-based CUDA Toolkit	C-based CUDA Toolkit		

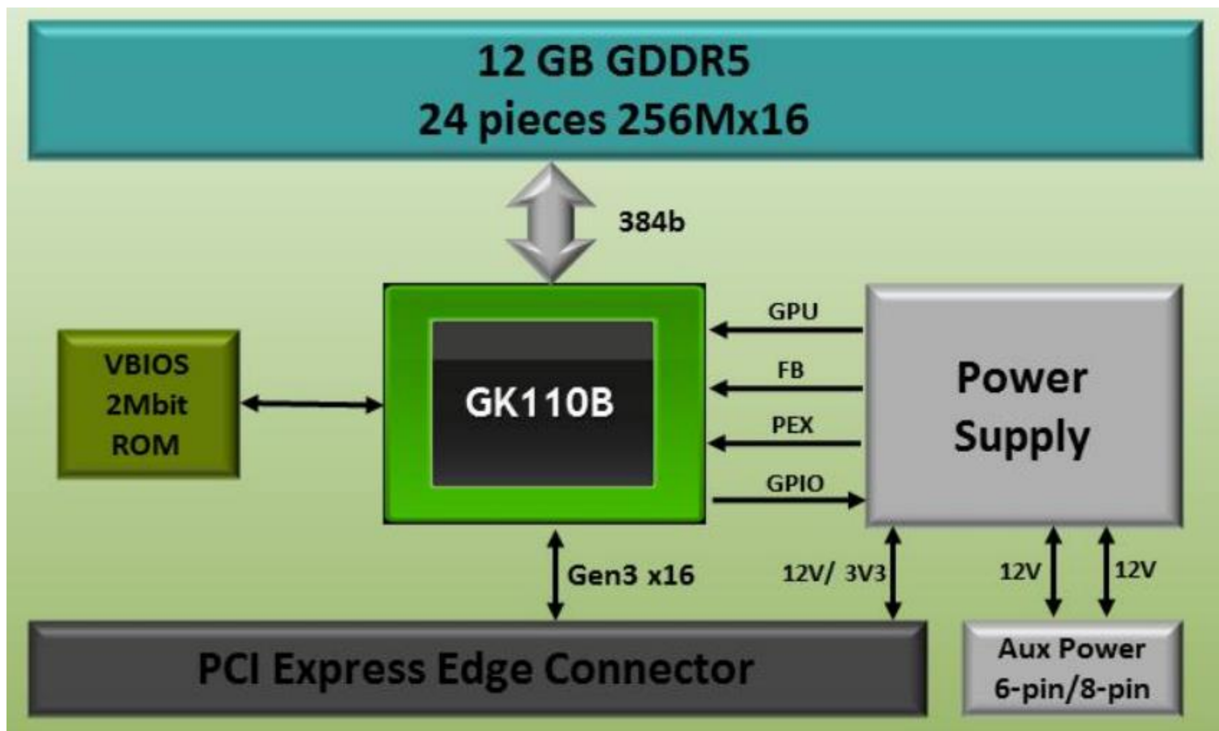


Figure 32. *NVIDIA Tesla K40 GPU.*

4.1.2 Intel MIC

The Intel MIC represents a step in the path from multicore to many-core. There are neither specialized accelerators or dedicated memory structures, but only relatively simple x86 processors with their attached vector-processing units and caches, the DRAM controllers, and the PCIe interface (Figure 33). All of the power of the chip is dedicated to executing x86 instruction code [8,22].

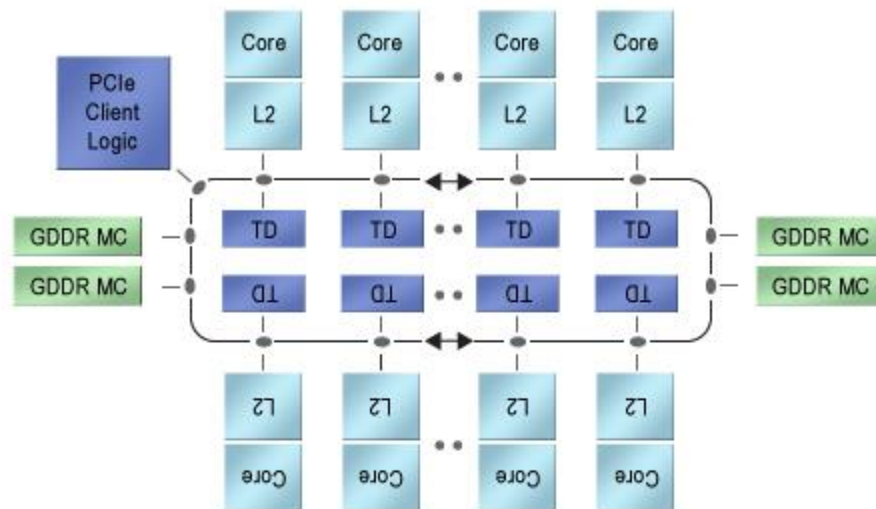


Figure 33. *Intel's Xeon Phi is a heterogeneous array of more than 50 x86 cores linked by a two-way racetrack interconnect scheme.*

The Intel chip shows the virtue of being homogeneous architectures, without special calls to hardware accelerators and without location-dependent performance. Furthermore, software designers can map any mixture of data-flow, multithread, or data-parallel machines onto the CPUs.

4.2 Scalability Issues for Many-core Processors

In the previous chapter, a brief outline has been carried out on well-established technologies used in building multi-core processors as well as the emerging technologies that are targeting some of the shortcomings of current technologies. On manufacturing technology level, major chip manufacturers estimate that current CMOS technologies will produce smaller components down to approximately the 6 nm manufacturing process (Figure 4). At that level, at least two major challenges will emerge [8]:

- As the size of transistors will be measured in a few tens of atoms at most, quantum effects will have to be taken into account and, consequently, unreliability of the hardware will increase, with components failing more often and, more importantly, intermittently
- There will be so many transistors on the chip that it will be impossible to switch all of these at the same time; this phenomenon, known as the dark silicon problem, will have a significant impact on future processors design

The unreliability of future hardware will likely lead to the implementation of redundant execution mechanisms. By following this approach, multiple cores will perform the same computation, in order to increase the probability that at least one will succeed; in some cases a voting scheme on the result (verifying if all the computations yielded the same result) may be used to guarantee correctness of the calculation. Even if such kind of mechanisms will be invisible to the software, they will impact the complexity of the design of logical processor cores.

The dark silicon problem is trickier to address. In particular, by lowering the frequency at which chips operate would only push the limit further [23], but finally it will become an issue. Alternative solutions include the adoption of “memristors” as building blocks (instead of transistors) and chip designs where, at any given time, only a subset of components would be active, depending on the type of application [24]. Optical interconnects within and between chips could also allow for building smaller chips, interconnected to build larger, cache coherent logical chips.

Another scalability bottleneck relates to the design of cache coherency protocols [25]. Synchronizing access to the same memory area from a large number of cores will increase the complexity of coherency design and will lead to increasing delay and latency in accessing frequently modified memory areas. To maintain a coherent view across hundred or thousand cores could be expensive and impractical.

Memory bandwidth will be another scalability bottleneck. The leveling of the core frequency will lead to reduced latency, but the increase in the number of cores will multiply the amount of data required by the cores and thus the total memory bandwidth that future chips will require. As an extension of Moore's law for an ever-increasing number of cores (in principle, following the same trend of doubling core-count every two years) a similar trend would need to be followed by memory bandwidth. This trend has not been followed by industry in past years.

On the basis of previous observations about future developments and bottlenecks, the following trends will dominate the design of future processors:

- A shift towards simple, low frequency, low complexity cores is expected, coupled with an increase of the core count to the level of several hundred within five to ten years; heterogeneity in core capabilities will be a factor, because it is an easy optimization gain
- Improvements will focus on novel memory technologies that can deliver higher bandwidth access; technologies such as 3D stacking, optical interconnections and memristors will be used in processor designs; especially optical interconnects facilitate chips structure and interconnection
- The size of on-chip memory will be remarkably increased, and some emerging technologies will reduce the footprint, power consumption and complexity of designing such solutions, similar to the development of the embedded DRAM technology; once again 3D stacking and memristors may be some of the future technologies
- Hardware accelerators will be abundant, due to the low footprint and low power consumption
- Aggressive power optimization mechanisms, as near threshold operation, power gating, frequency and voltage scaling, will become prevalent not only in traditionally low power domains, but also in most areas where processors are used.

5. Simulation of electromagnetic field problems

Model creation is at the very heart of physics and other sciences as well. The essence of a model is that it should be a simplified representation of real object or physical phenomena which serves a particular, and perhaps limited, purpose. The physical systems are often very complex, containing entities that cannot be considered and whose existence and properties can only be inferred or approximated.

The need for more detailed analysis during design and verification process, known as virtual prototyping, led to the diffusion of numerical methods finalized to the study of different physical phenomena. The study of new induction heating processes involves the determination of electric and magnetic fields distribution in a physical domain, as well as the thermal distribution inside heated parts. In most practical cases, an analytic approach would not be possible due to non-linear phenomena, since magnetic saturation in magnetic flux concentrators, iron core losses, radiative thermal exchanges could not be easily treated. Numerical models allow for the determination of electric current density distribution inside workpieces and inductors, even in case of complex geometrical and physical properties. By summarizing, through a numerical model it is possible:

- To determine electric and magnetic fields distribution, both from a local (quantities in particular positions in the domain) and a global point of view (i.e. magnetic flux, inductance, induced power)
- To reduce high field gradients, to avoid magnetic saturation conditions, to exploit anisotropic materials in a more profitable way
- To predict the behavior of an electromagnetic device or process with good accuracy
- To reduce costs of prototyping

5.1 Finite Element Method

Finite Element Method (FEM) is a numerical technique that allows for the solution of a vector field problem, in which non homogeneity, directional anisotropy, non-linearity and time-varying phenomena are present [26]. In mathematical terms, this translates as to find approximate solutions to boundary value problems for differential equations. This method uses a variational approach (calculus of variations) to minimize an error function and produce a stable solution. Analogous to the idea that connecting many tiny straight lines can approximate a larger circle, FEM encompasses all the methods for connecting many simple systems of element equations over many small subdomains, named finite elements, in order to approximate a more complex system of equations over a whole domain. The subdivision of the domain into simpler parts provides several advantages:

- An accurate representation of complex geometries
- Possibility to include different material properties
- Easy representation of the global solution
- Accounting for local effects.

A typical FEM procedure involves:

- dividing the domain of the problem into a collection of subdomains, with each subdomain represented by a set of element equations related to the original problem
- systematically recombining all sets of element equations into a global system of equations for the final calculation. The global system of equations has known solution techniques, and can be calculated after imposing proper boundary conditions to the original problem, in order to obtain a numerical solution.

In the first step above, the element equations are simple equations that locally approximate the original complex equations to be studied, where the original equations are partial differential equations [27]. To explain the approximation process, FEM is commonly introduced as a special case of Galerkin method. The process is to construct an integral of the inner product of the residual and the weight functions, and set the integral to zero. In simple terms, it is a procedure that minimizes the error of approximation by fitting the original PDE through trial functions. The residual is the error caused by the trial functions, and the weight functions are polynomial approximation functions. The process eliminates all the spatial derivatives from the PDE, thus approximating the PDE locally with:

- a set of algebraic equations for steady state problems,
- a set of ordinary differential equations for transient problems.

These equation sets, defined on each finite element, are the element equations.

By according to a mathematical approach, the method can be introduced starting from Rayleigh-Ritz principle (variational method), for which solving $\mathbf{Ax} = \mathbf{b}$ means to find the minimum of $P(x) = \frac{1}{2}\,\mathbf{x}^{\mathrm{T}}\mathbf{Ax} - \mathbf{x}^{\mathrm{T}}\mathbf{b}$ [28]. Consider the equation $-u'' = f(x)$, with boundary conditions $u(0) = u(1) = 0$. This problem is infinite-dimensional, where $\mathbf{A}$ is replaced by $-\,\mathrm{d}^2/\mathrm{d}x^2$ and $\mathbf{b}$ vector is replaced by a function f. The total energy $P(v)$, to be minimized in order to get the solution $\mathbf{x}$, is obtained by replacing inner products $v^T f$ with integrals of $v(x)f(x)$:

$$P(v) = \frac{1}{2}\,v^T A v - v^T f = \frac{1}{2}\int_0^1 v(x)\big(-v''(x)\big)dx - \int_0^1 v(x)f(x)dx$$

and $P(v)$ must be minimized in respect to all functions $v(x)$ that satisfy the boundary conditions $v(0) = v(1) = 0$. The function that minimizes $P(v)$ is $u(x)$. Then, the differential equation has been converted to a minimum problem. By integrating by part, it follows that:

$$\int_0^1 v(-v'')dx = \int_0^1 (v')^2 dx - [vv']_{x=0}^{x=1} \xrightarrow{yields} P(v) = \int_0^1 \left[\frac{1}{2}\left(v'(x)\right)^2 + v(x)f(x)\right]dx$$

The term $vv'$is null at the boundary, because $v = 0$, and $\int \left(v'(x)\right)^2 dx$ is positive, as $x^T Ax$. Thus, there is a minimum of $P(v)$. In order to calculate that minimum, Rayleigh-Ritz principle lets to bring back to a $n-$dimensional problem by choosing only n test functions $V_1(x), \dots, V_n(x)$. Among all the linear combination $V = y_1 V_1(x) + \cdots + y_n V_n(x)$, the solution is the particular U that minimize $P(V)$. In this way, it is enough to minimize P on a subspace of V instead of on all possible $v(x)$. Once the $U(x)$ function is found, it should be a good approximation of correct solution $u(x)$.

By replacing V with v, the quadratic form turns into:

$$P(V) = \frac{1}{2}\int_0^1 \left(y_1 V_1{}'(x) + \cdots + y_n V_n{}'(x)\right)^2 dx - \int_0^1 \left(y_1 V_1(x) + \cdots + y_n V_n(x)\right) f(x) dx$$

Test functions are chosen a priori. Unknowns $y_1, \dots, y_n$ are represented in a **y** vector, thus leading to the quadratic form $P(V) = \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{A}\mathbf{y} - \mathbf{y}^{\mathrm{T}}\mathbf{b}$. Entries A_{ij} in matrix **A** are $\int V_i' V_j'\, dx$, coefficients of $y_i y_j$. Components b_i are $\int V_j f\, dx$. It is possible to find the minimum of $\frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{A}\mathbf{y} - \mathbf{y}^{\mathrm{T}}\mathbf{b}$ by solving the linear system $Ax = b$. Thus, Rayleigh-Ritz method can be summarized in three steps:

- Choose test functions $V_1, \dots, V_n$
- Computation of A_{ij} and b_j entries
- Solution of $\mathbf{Ax} = \mathbf{b}$ in order to find $U(x) = y_1 V_1(x) + \cdots + y_n V_n(x)$.

In the first step, test functions must be carefully chosen because:

- If functions $V_j(x)$ were not simple enough, following steps could become too expensive or impractical to compute
- If some combinations of $V_j(x)$ could lead to a bad $u(x)$ approximation, the solution could be not as accurate as expected

In order to combine both computability and accuracy, element-wise polynomial functions are chosen as test functions $V(x)$.

The most simple and used finite element is the linear element shown in Figure 34. Internal nodes are filled in $x_1 = h, x_2 = 2h, \dots, x_n = nh$. Then, V_j is the test function that is 1 on node x_j, and 0 on other nodes. The function covers a small interval close to its node, called support, and it is null on all other nodes, including the boundaries. Each combination $y_1 V_1 + \cdots + y_n V_n$ shows a value y_j on node j, as reported in Figure 35.

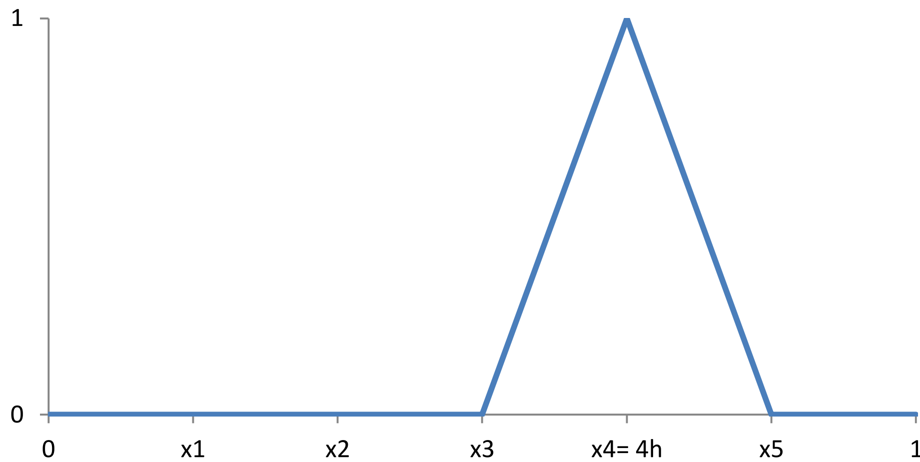


Figure 34. *Example of test function on a 1D linear finite element.*

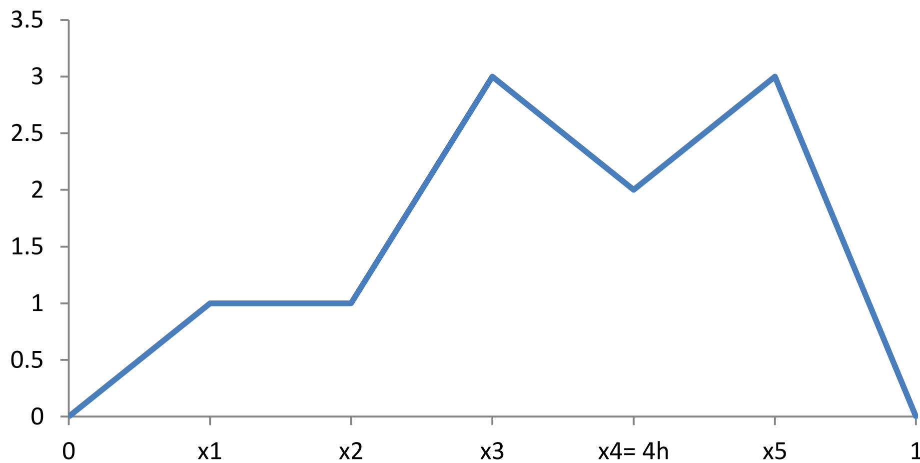


Figure 35. *Example of linear combination of test function on a discretized 1D domain.*

The second step is the computation of matrix entries $A_{ij} = \int V_i' V_j' \, dx$ in the "stiffness" matrix A. On x_j support, the slope V_j' is $1/h$ on the left of x_j, while it is $-1/h$ on the right. If functions from neighbor nodes are not superimposed, then $V_i' V_j'$ is zero and $A_{ij} = 0$. Each test function is superimposed to itself and to only two neighbor, thus:

Diagonal $\quad i = j \quad$ $A_{ij} = \int V_i' V_j' \, dx = \int \left(\frac{1}{h}\right)^2 dx + \int \left(-\frac{1}{h}\right)^2 dx = \frac{2}{h}$

Out of diagonal $\quad i = j \pm 1 \quad$ $A_{ij} = \int V_i' V_j' \, dx = \int \left(\frac{1}{h}\right)\left(-\frac{1}{h}\right) dx = -\frac{1}{h}$

The stiffness matrix A is obtained:

$$A = \frac{1}{h}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}$$

Usually, finite element method exploits more sophisticated test functions, as higher degree polynomials, built on triangles/tetrahedral or rectangles/brick elements. This method is a systematic approach for discretization of partial differential equations on irregular grids. In practical problems, this method leads to large square matrices, characterized by a strong sparsity factor (number of nonzeros entries to number of entries ratio). It is essential to use simple polynomials as test functions, because their derivatives are easy to find and integrate on the element. Finally, each of b_j entries is calculated as the average of f around the point x_j, such that $b_j = \int V_j f\, dx$.

In the third step, the linear system is solved in order to find the minimizing solution $U = y_1 V_1 + \cdots + y_n V_n$. By connecting all the y_j values on x_j nodes, the approximated solution $U(x)$ is obtained.

5.1.1 FEM in induction heating simulation

A big role in introduction of FEM in electromagnetism belongs to P.Silvester, M. Chari and other colleagues in 1970's [29]. While the first application of the method was in the area of electrical machine analysis, Silvester and Haslam extended FEM to induction heating problems by considering the eddy currents induced in a linear conductor [30]. The next year, in 1973, Chari extended the method to a ferromagnetic conductor subjected to an oscillating AC field [31], a problem much closer to the heart of induction heating. In 1975, A. Foggia extended FEM to the case of a ferromagnetic conductor in presence of a travelling magnetic field [32], providing the basis for metal strip heating applications and electromagnetic stirring in induction furnaces.

One of the key limitations of FEM is the need to include the far field region as part of the solution domain. An attempt to overcome this limitation led to the first hybrid formulations for a magnetic field problem. McDonald and Wexler [33] proposed to bind the eddy current and source regions, at an appropriate distance, by a surface on which a boundary integral formulation could be used. Since then, many improvements provided a firm theoretical basis for FEM to be effectively used in the simulation of induction heating processes.

In a generic induction heating electromagnetic problem, high frequency source fields are involved, and the time scale is in the order of milliseconds-microseconds, whereas the time scale of the thermal problem is in the range of seconds or minutes. FEM allows for the solution of the

quasi-stationary electromagnetic field problem, that is the determination of magnetic field distribution, induced current and power density distribution in the analyzed domain, eventually with Newton Raphson iterative loop, i.e. when accounting for non-linear field-dependent properties of ferromagnetic materials. A step of temperature calculation follows, with proper correction of temperature dependent properties of materials, for the updated time, and so on. This approach has been introduced by Holmdahl and Sundberg in [34]. By the same methodology, Lavers [35] provided one of the first solutions of a 2D induction heating problem involving a ferromagnetic, square section conductor. He also described the computational methods for the analysis of the molten metal confinement, the problem that includes the simulation of electromagnetic and magneto-hydrodynamic phenomena [36]. Massé described a predictor corrector scheme that allowed a time step refinement during the critical Curie transition [37].

In conclusion, at the very beginning of 90s, FEM was ready for a wide practical use in simulation of induction heating and melting processes and installations [38]. In later years, it has established as the preferred method due to its versatility and wide use in other technical fields, i.e. structural mechanics. Large commercial multipurpose packages appeared on the market and many companies and groups started to use them for induction heating simulations. Examples of these software are Cedrat Flux [39], Ansoft Maxwell [40], Vector Field Opera [41], Quick Field [42] and Comsol [43].

5.1.2 Nodal and edge elements

The numerical analysis of three-dimensional eddy current problems with the aid of the finite element method has been one of the main directions of research in computational electromagnetics during the 1990s. Available finite-element codes typically use vector and scalar potential functions to describe the field quantities. Finite element techniques using nodal based functions to approximate both scalar and vector potentials (known as nodal finite elements) were first to emerge [44-48]. Various possible formulations have been proposed by O. Birò in terms of different potential functions [53]. The most outstanding feature of these approaches is the incorporation of the Coulomb gauge on the vector potentials in the governing equations [49-50]. This technique is consistent with nodal finite elements and leads to robust numerical realizations.

Nevertheless, the use of nodal finite elements to approximate vector potentials leads to difficulties when applied to two specific types of problems. The first class of problems involves highly permeable conducting regions (i.e. iron parts) surrounded by non-magnetic and non-conducting domains (i.e. air) with a magnetic vector potential A used to describe the field inside the iron parts. The use of a magnetic scalar potential Φ in air in the vicinity of the iron/air interface is not desirable because of the weak coupling between the two potentials. The remaining option of employing A in air leads to considerable inaccuracies at the interface due to the continuity of the normal component A_n of the magnetic vector potential resulting in a natural interface condition which requires a jump in the divergence of A [51,52]. A second problem, not

easily treated in case of nodal finite elements, arises when singularities are approximated at sharp corners along which the tangential components of a vector potential vanish, and this vector potential describes the field on both sides of the interface constituted by the corner. Indeed, the continuity of the normal component renders the field obtained as the curl of the vector potential to be zero at the corner instead of an expected high value. A typical example is the modeling of thin flaws or cracks in a conducting piece, in which the eddy currents are derived from a current vector potential T, by setting the tangential components of T to zero.

The above problems are avoided if the vector potentials are approximated by edge (tangential vector) finite elements [54-56], because these elements enforce only the (physically necessary) continuity of the tangential components of the vector potentials, but not forcing the continuity of the normal component. However, A or T formulations cannot be gauged in the same way as in the corresponding nodal finite element formulations, since the lack of continuity of their normal components implies that their divergence cannot be defined at element interfaces. One of the most outstanding properties of edge finite elements is that the curls of the vector basis functions are not linearly independent although the test functions themselves are. This holds both for element shape functions, defined on each element, and for the global basis functions defined on the global mesh, leading to a tree-cotree method for gauging vector potentials [57].

5.1.3 Definition of the electromagnetic problem

An eddy current problem arises when an object, made of conducting material, is exposed to a time varying magnetic field, originated by a source [53]. In induction heating applications, the frequency range extends from a few Hertz to a few MHz, thus the displacement current density is assumed to be negligible, i.e. the differential equations of quasi-static fields hold. This leads to a static but time varying magnetic field in the non-conducting regions Ω_n surrounding the eddy current carrying conductors constituting the region Ω_c. In Ω_c, the current density distribution is unknown and an eddy current field is present.

The excitation can be of three types. The first possibility, called coil excitation, is realized by a coil with given current density J_0, situated in Ω_n. If the total current through a conductor is prescribed, but the current density distribution herein is unknown, a skin effect problem with current excitation is obtained. If, conversely, the voltage of the conductor is given, a skin effect problem with voltage excitation is defined. Skin effect problems are conveniently treated by different potential formulations.

The time variation will be assumed to be sinusoidal with an angular frequency of ω, and complex notation will be used. Hence, the differential equations to be solved are the following:

$$curl\,H = J_0 \qquad \text{in } \Omega_n$$

$$div\,B = 0 \qquad \text{in } \Omega_n$$

$$curl\ H = J \qquad \text{in } \Omega_c$$

$$curl\ E = -j\omega B \qquad \text{in } \Omega_c$$

$$div\ J = 0 \qquad \text{in } \Omega_c$$

$$div\ B = 0 \qquad \text{in } \Omega_c$$

Where H is the magnetic field intensity, B is the magnetic flux density, E is the electric field intensity and J is the current density.

The field quantities satisfy the following constitutive equations:

$$B = \mu H \qquad \text{in } \Omega_n \text{ and } \Omega_c$$

$$J = \sigma E \qquad \text{in } \Omega_c$$

Where μ is the magnetic permeability and σ is the electric conductivity.

The effect of a given current density in the coils is assumed to be described by an impressed field quantity T_0 satisfying:

$$curl\ T_0 = J_0$$

In this way, it is not necessary to model coil geometry in the finite element mesh. Usually, T_0 is chosen equal to H_s, that is the magnetic field in free space computed by integration of Biot-Savart law.

The boundary conditions of the problem (homogeneous for simplicity) can be given at artificial far boundaries (i.e. "infinite boxes"), or along symmetry/periodicity planes, as follows:

$$H \times n = 0 \quad \text{or} \quad B \cdot n = 0 \qquad \text{on } \Gamma_n \text{ (boundary of } \Omega_n\text{)}$$

Therefore, the normal component of B is continuous across the interface. Therefore the tangential component of H is continuous across the surface (since no surface current is considered).

$$H \times n = 0 \quad \text{or} \quad E \times n = 0 \qquad \text{on } \Gamma_c \text{ (boundary of } \Omega_c\text{)}$$

Therefore the tangential component of E is continuous across the interface.

The continuity conditions on the interface Γ_{nc} between the conducting and non-conducting regions are

$H \times n$ and $B \cdot n$ are continuous on Γ_{nc}.

These interface conditions will provide the coupling between the formulations in the non-conducting eddy current free region Ω_n and the conducting eddy current region Ω_c. The solution of the differential equations with the boundary and interface conditions is unique.

There are two ways to describe the magnetic field in the non-conducting eddy current free region Ω_n: either by a magnetic scalar potential or by a magnetic vector potential. The most used formulations are detailed in [53].

The ϕ-formulation

The magnetic scalar potential ϕ is defined by

$$H = T_0 - grad\ \phi$$

The magnetic scalar potential ϕ is usually called the reduced magnetic scalar potential because the source term is hidden in T_0.

Thus, the differential equation is:

$$div(\mu\ grad\ \phi) = div(\mu T_0) \text{ in } \Omega_n$$

With boundary conditions on Γ_n:

$\phi = 0$ Dirichlet condition

$\mu\ grad\ \phi \cdot n = \mu T_0 \cdot n$ Neumann condition

If no Dirichlet boundary conditions are prescribed, then $\phi = 0$ must be specified at some points in order to ensure the unicity of the reduced magnetic scalar potential. The scalar potential is approximated through nodal basis function. The ϕ-formulation is satisfactory if the permeability of the medium is not very high. In this case the simulated magnetic field $H = T_0 - grad\ \phi$ does not differ too much from the source magnetic field T_0. By increasing the permeability of the medium, e.g. when soft magnetic materials are modeled, the resultant magnetic field H is decreasing inside the magnetic material, e.g. $T_0 = grad\ \phi$, which is a strong disadvantage from numerical side. This leads to a cancellation error, i.e. when two almost equal quantities are subtracted from each other.

In order to overcome this penalizing situation, it is possible to derive the magnetic field intensity from a total magnetic scalar potential Ψ as

$$\mathrm{H} = -\mathrm{grad}\ \Psi$$

Since $curl\ H = 0$ and $J_0 = 0$ inside highly permeable regions.

This can be the situation in most practical cases, since the magnetic field intensity is generated by currents flowing in coils. The source coils are usually placed somewhere around the

ferromagnetic parts, but source currents are not flowing inside ferromagnetic materials. In this situation, it is possible to use the reduced magnetic scalar potential ϕ in the air region and the total magnetic scalar potential Ψ inside the highly permeable region and the two formulations must be coupled through the interface between the two regions.

As an alternative, it is also possible to avoid cancellation error inside ferromagnetic materials with high permeability by choosing an appropriate representation of the impressed current vector potential. A "compatible approximation" should be used to interpolate both the impressed field T_0 and the unknown $grad\ \phi$ part of H. In finite element method, this means that the vector field T_0 should be represented by edge elements and the scalar field ϕ should be interpolated by nodal elements and the order of them must be the same.

The A_r-formulation

In order to avoid a modeling of the coils, magnetic flux density is computed as the sum of impressed Biot-Savart field in free space $\mu_0 H_s$, and of the curl of a "reduced" vector potential A_r:

$$B = \mu_0 H_s + curl\ A_r$$

Then, the differential equation is:

$$curl\left(\frac{1}{\mu}\ curl\ A_r\right) = curl\ H_s - curl\frac{\mu_0}{\mu} H_s \quad \text{in } \Omega_n$$

With boundary conditions on Γ_n:

$$A_r \times n = 0 \qquad \text{Dirichlet condition}$$

$$\frac{1}{\mu}\ curl\ A_r \times n = 0 \qquad \text{Neumann condition}$$

The reduced vector potential can be approximated either by nodal or edge basis functions.

Similarly, two vector potential formulations can be used to describe the eddy current field in the conducting region Ω_c: either a magnetic vector potential or an current vector potential. The magnetic vector potential can be employed with or without an electric scalar potential, whereas the current vector potential description must be augmented by a magnetic scalar potential.

The A,V-formulation

The field quantities are derived from a magnetic vector potential A and an electric scalar potential V, defined as:

$$B = curl\ A$$

$$E = -j\ \omega\ A - j\ \omega\ grad\ V$$

Then, the differential equation is:

$$\begin{cases} curl\,\frac{1}{\mu}\,curl\,A + j\,\omega\sigma\,A + j\,\omega\,\sigma\,grad\,V = 0 \\ -div(j\,\omega\sigma\,A + j\,\omega\,\sigma\,grad\,V) = 0 \end{cases} \quad \text{in } \Omega_c$$

With boundary conditions on Γ_c:

$$A \times n = 0 \quad \text{or} \quad \frac{1}{\mu} curl\,A \times n = T_0 \times n$$

$$V = V_0 = constant \quad \text{or} \quad n \cdot (-j\,\omega\sigma\,A - j\,\omega\,\sigma\,grad\,V) = 0$$

Magnetic vector potential can be approximated either by nodal or edge basis functions, while scalar electric potential is approximated by nodal basis functions.

The T, ϕ formulation

The field quantities are derived from potentials as:

$$H = T_0 + T - grad\,\phi$$

$$J = curl\,T_0 + curl\,T$$

Then, the differential equation is:

$$\begin{cases} curl\,\frac{1}{\sigma}\,curl\,T + j\omega\mu\,T - j\omega\mu\,grad\,\phi = -\,curl\,\frac{1}{\sigma}\,curl\,T_0 - j\omega\mu\,T_0 \\ j\omega\,div\,(\mu\,T - \mu\,grad\,\phi) = -j\omega\,div\,(\mu\,T_0) \end{cases} \quad \text{in } \Omega_c$$

With boundary conditions on Γ_c:

$$T \times n = 0 \quad \text{or} \quad \frac{1}{\sigma} curl\,T \times n = -\frac{1}{\sigma} curl\,T_0 \times n$$

$$\phi = \phi_0 = constant \quad \text{or} \quad n \cdot (-\mu T - \mu\,grad\,\phi) = -n \cdot\,\mu\,T_0$$

Current vector potential can be approximated either by nodal or edge basis functions, while magnetic scalar potential is approximated by nodal basis functions.

5.2 Integral equations and hybrid methods in induction heating simulation

Concurrently to the development of FEM, other numerical formulations of eddy current problem, based on integral equations, have been explored [58-61]. This approach let to the development of several calculation methods such as Inductively Coupled Circuits (ICC), Volume Integral Methods (VIM), Boundary Element Method (BEM) and Partial Element Equivalent Circuit method (PEEC).

All integral equation methods are based on replacement of the real conductive and magnetic bodies with the primary and secondary (induced) field sources. Interaction of the selected sources at distance allows to compose equations for the secondary source calculation. A common action between primary and secondary sources must provide the same distribution of at least one of the electromagnetic field components, as it is in the real system. This is a necessary and sufficient condition for calculation of the secondary sources and, then, all the required parameters of the system. Secondary sources may be the real physical sources, such as currents in active conductors and electric charges, or artificial sources introduced for the calculation purposes only. In magnetic bodies, the secondary sources describing an action of the surface and volumetric magnetization currents must be added. Mayergoiz strongly contributed to the theory of integral equations for eddy current problems [62-65] by describing a complete theory of the secondary sources and the ways to build the computational models.

The main advantages of integral methods are small area of discretization (only conductive and magnetic bodies) and, therefore, less number of elements and no necessity to use any artificial external borders for the open space systems. The main drawbacks lay in the difficult treating of complex geometries and the unsuitability to calculate fields in non-linear magnetic regions. It must be also mentioned that integral methods give rise to fully populated matrices and each entry must be calculated by analytical procedures or numerical quadrature algorithms.

A different approach, originated from ICC, was proposed by V. Nemkov. It is based on a combination of Integral Equations for the “external” area, where the known sources are located, with an Impedance Boundary Condition (IBC) on the surface of the heated part. The surface impedance of the body may be calculated analytically in case of pronounced skin effect [66].

The Boundary Element Method is based on the fact that reaction of the body with linear properties (magnetic permeability, dielectric permittivity and electric resistivity) on the whole system field distribution may be adequately described by properly distributed secondary sources on the surface (the boundary) of the body. The influence of one element to another one is described through Green functions [67,68]. BEM could be applied to linear 2D and 3D problems, with complex geometries. An extension of this method to non-linear materials has been proposed in [69,70].

The key idea behind Partial Element Equivalent Circuit method (PEEC) is to attribute a contribution to the total impedance to each part of a structure [71]. On the basis of Maxwell's equations, the method model connections of an electric structure by using an electric circuit with lumped components (L, R and M). PEEC is a method which makes it possible to calculate precisely the resistance, the partial inductance and the mutual partial inductances of rectilinear conductors of rectangular cross-section, by taking into account conductors of any cross-section either by using numerical integration techniques or by subdividing the cross-section in rectangular basic elements (mesh concept in Figure 36). The computation of resistances and partial inductances is based on the Ohm’s law and on the line integral of the vector potential

along the conductors respectively; it requires that the conductors be placed in a region where there are no nonlinear magnetic materials and that the current density on the conductor cross-section be uniform. This last condition is incompatible with the frequency approach. Then, in order to take into account the effects of the frequency on the current non-uniformity on the cross-section of conductors, this area is divided in small sections. PEEC is not used to simulate induction heating applications, but it is very effective in EMC studies, for example in PCB design. A PEEC software, able to simulate also parasitic capacitance effects, is Cedrat InCa3D [39].

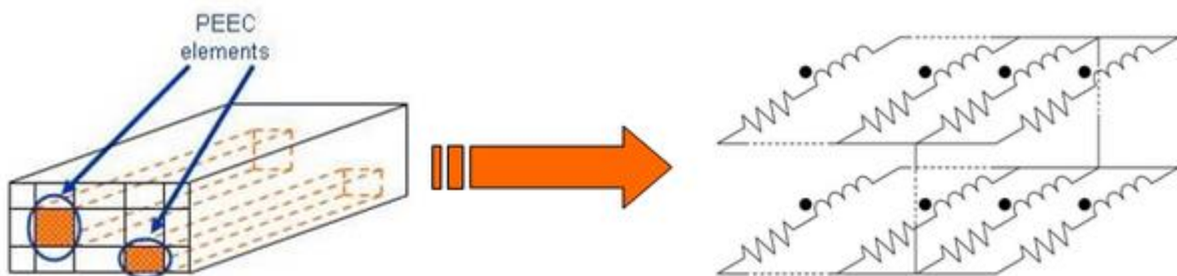


Figure 36. *Representation of conductors in a PEEC equivalent circuit.*

6. Sparse linear systems: direct and iterative solvers

Solving linear systems of equations lies at the heart of many problems in computational science and engineering. In many cases, particularly when discretizing continuous problems by differential numerical methods, the system is large and the associated matrix A is sparse. This holds in application of finite element method electromagnetic, thermal or structural problems, and of volume element method in fluid dynamics problems. Furthermore, depending on the particular formulation of the physical problem, the matrix could be symmetric or non-symmetric, positive definite or indefinite.

Graph theory is an ideal tool for representing the structure of sparse matrices and for this reason it is exploited in sparse matrix techniques. For example, graph theory is the key aspect used in unraveling parallelism in sparse Gaussian elimination or in most preconditioning techniques. As an example of the use of graph models, parallelism in Gaussian elimination can be extracted by finding sets of unknowns that are independent at a given stage of the elimination process. These unknowns do not depend on each other, thus the rows corresponding to such unknowns can be used as pivots simultaneously. If the matrix is diagonal, then all unknowns are independent. Conversely, when a matrix is dense, each unknown will depend on all other unknowns. Sparse matrices lie between these two limit situations [77].

Permuting the rows or the columns, or both the rows and columns, of a sparse matrix is a common operation. In fact, reordering rows and columns is one of the most important elements used in parallel implementations of both direct and iterative solution techniques. When the rows of a matrix are permuted, the order in which the equations are written is changed. Otherwise, when the columns are permuted, the unknowns are relabeled, or reordered. Two-sided permutations, that is to apply the same permutation to both the columns and the rows of the matrix, are also a common operation in sparse matrix techniques. This is related to the fact that the diagonal elements in linear systems play an important role. For instance, diagonal elements are typically large in matrices arising from PDE discretization and it may be useful to preserve this important property in the permuted matrix. The interpretation of the symmetric permutation is quite simple and straightforward. The resulting matrix corresponds to renaming/relabeling/reordering the unknowns and then reordering the equations in the same manner [77].

The storage of matrices is also an important factor to be considered. If the coefficient matrix A is sparse, large-scale linear systems of the form $\mathbf{Ax} = \mathbf{b}$ can be efficiently solved if zero elements of A are not stored. Thus, sparse storage schemes allocate contiguous storage in memory for the nonzero entries of the matrix and a limited number of zeros. This requires a scheme for knowing where the elements fit into the full matrix.

There are many methods for storing the data in order to take advantage of the large number of zero elements. The main goal is to represent only the nonzero entries, though allowing for good performance in common matrix operations [72].

The simplest storage scheme for sparse matrices is the coordinate format (COO). The data structure consists of three arrays:

- a real array AA containing all the real or complex values of the nonzero elements of A in any order
- an integer array JR containing their row indices
- a second integer array JC containing their column indices.

All three arrays are of length N_z, the number of non-zero elements.

For example, the matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

Will be represented in COO format as:

$$AA = (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12)$$

$$JR = (1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 3 \quad 4 \quad 4 \quad 5)$$

$$JC = (1 \quad 4 \quad 1 \quad 2 \quad 4 \quad 1 \quad 3 \quad 4 \quad 5 \quad 3 \quad 4 \quad 5)$$

In the previous case, the elements are listed by row. Another common ordering is by column. COO format is useful for its flexibility, since it is used as an "entry" format for most sparse matrix software packages [73].

However, the array JC contains redundant information, thus it could be replaced by an array of pointers to the beginning of each row, in order to save memory. The new data structure, called Compressed Sparse Row storage (CSR), is still composed by three arrays:

- a real array AA contains the nonzero entries of the matrix, stored row by row, from row 1 to n-th. The length is N_z;
- an integer array JA contains the column indices of nonzero elements as stored in AA, and its length is N_z;
- an integer array IA contains the pointers to the beginning of each row in array AA and JA. The content of $IA(i)$ is the position in AA and JA where the i-th row starts. The length of

IA is $n + 1$, and $IA(n + 1)$ contains the value $IA(1) + N_z$, that is the memory address of the fictitious row $n + 1$.

By following the example above, the matrix A is stored in CSR format as:

$$AA = (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12)$$

$$JA = (1 \quad 4 \quad 1 \quad 2 \quad 4 \quad 1 \quad 3 \quad 4 \quad 5 \quad 3 \quad 4 \quad 5)$$

$$IA = (1 \quad 3 \quad 6 \quad 10 \quad 12 \quad 13)$$

The CSR scheme is preferred over COO because it is more useful for performing typical computations on sparse matrices, i.e. sparse matrix vector products, which is an important operation for iterative solutions of sparse linear systems.

In Figures 37 and 38, an example for the matrix-by-vector product in CSR storage is given.

```
DO I = 1,N
   K1  = IA(I)
   K2 = IA(I + 1) - 1
   Y(I) = DOTPRODUCT( A(K1:K2),X(JA(K1:K2)))
END DO
```

Figure 37. *Example code of sparse matrix-vector product for CSR format matrices.*

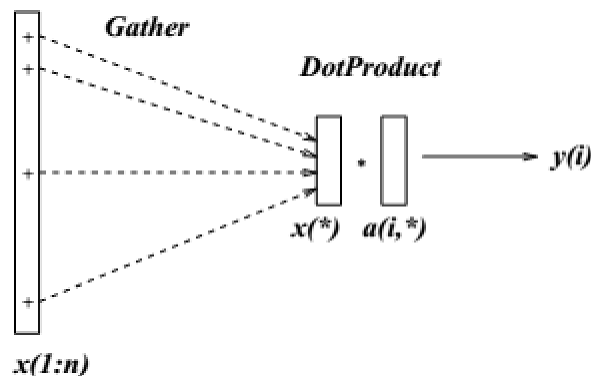


Figure 38. *Illustration of the row-oriented sparse matrix-vector multiplication.*

Notice that each iteration of the loop computes different component of the resulting vector. This is an advantage because each of these components could be computed independently on a parallel

hardware. However, data need to be accessed in a non-patterned way, and limited memory bandwidth could lead to a bottleneck. Other techniques are available, such as data prefetching, vectorization and other advanced techniques, depending on the hardware.

Solving a lower or upper triangular system is another important “kernel” in sparse matrix computations. The code in Figure 39 shows a simple routine for solving a unit lower triangular system $\mathbf{Lx} = \mathbf{y}$ for the CSR storage format.

```
X(1) = Y(1)
DO I = 2,N
    K1 = IAL(I)
    K2 = IAL(I + 1) - 1
    X(I) = Y(I) - DOTPRODUCT(AL(K1:K2),X(JAL(K1:K2)))
END DO
```

Figure 39. *Example code for the solution of a lower triangular system in CSR format.*

At each step, the inner product of the current solution x with the i-th row is computed and subtracted from $y(i)$. This gives the value of $x(i)$. The dotproduct function computes the dot product of two arbitrary vectors $u(k1{:}\,k2)$ and $v(k1{:}\,k2)$. The vector $AL(K1{:}\,K2)$ is the i-th row of the matrix L in sparse format, and $X(JAL(K1{:}\,K2))$ is the vector of the components of X gathered into a short vector which is consistent with the column indices of the elements in the row $AL(K1{:}\,K2)$.

Compressed Sparse Column format (CSC) is similar to CSR, except that values are read first by column, a row index is stored for each value, and column pointers are stored. Recently, new formats emerged in order to fit particular classes of problems or to exploit in a better way new hardware.

Another important aspect related with the solution of large sparse linear systems is the conditioning of a system, that is to determine its stability. In particular, when the linear system arises from the discretization of physical problem models, both the system matrix **A** and the right hand side **b** are generally disposed to errors. When a small variation on **A** or **b** is reflected into a great variation on the solution **x**, the linear system is badly conditioned.

The conditioning number of a square matrix **A** is the real number $\chi(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$. The problem and related matrix **A** could be:

- well-conditioned, if $\chi(\mathbf{A})$ is almost equal to 1, i.e. if the relative error reflected on the solution is of the same order as the source error in B vector
- badly conditioned, if $\chi(\mathbf{A})$ is large

- badly formulated, if $\chi(\mathbf{A})$ is infinite

The conditioning number could be deteriorated by the following factors:

- Physical characteristics: very different numerical values of constants or of physical coefficients; in a ratio higher than 10^{-5}, i.e. computation domain characterized by regions of low-conducting materials and regions of high-conducting materials, or electric circuit that contains resistances or/and inductances characterized by both low and high values
- Poor quality mesh, with flattened elements, with a very high ratio between the size of the smallest mesh element and that of the largest mesh element, or a very high number of mesh elements
- Type of interpolation: a high degree of interpolation functions, or edge element interpolation. As an example, in Cedrat Flux software nodal FEM leads to invertible linear systems while edge elements leads to noninvertible linear systems by construction. A tree-cotree gauge technique can lead to invertible matrices but it deteriorates the matrix conditioning.

In order to decrease the value of the conditioning number, the initial system is transformed into an equivalent system having better a priori properties, through a preconditioning technique. In particular, preconditioning consists in replacing the system to be solved $\mathbf{Ax} = \mathbf{b}$ with an equivalent system $\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$. In general, a preconditioning algorithm build a matrix $\mathbf{M}^{-1}$ as approached to $\mathbf{A}^{-1}$ as possible, so that $\mathbf{M}^{-1}\mathbf{A}$ matrix is well-conditioned.

Finally, the scaling operation of linear systems can improve the conditioning by the so-called normalization techniques. Typically, solving a non-scaled linear system can lead to the divergence of the iterative process or to the slow convergence of this process, in case of iterative solvers, or it can generate solutions of bad quality, in case of direct solvers.

6.1 Direct methods

Since the early 1990s, many new algorithms and a number of software packages have been developed, in order to efficiently solve sparse symmetric and non-symmetric systems [74]. In particular, direct methods for solving a sparse linear system $\mathbf{Ax} = \mathbf{b}$ involve the explicit factorization of the system matrix $\mathbf{A}$ (or a permutation of $\mathbf{A}$) into the product of lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{U}$. In the symmetric case, for positive definite problems $\mathbf{U} = \mathbf{L}^T$ (Cholesky factorization) or, generally, $\mathbf{U} = \mathbf{DL}^T$, where $\mathbf{D}$ is a block diagonal matrix with 1×1 and 2×2 blocks. Forward elimination followed by backward substitution completes the solution process for each given right-hand side $\mathbf{b}$.

Direct methods are important because of their generality and robustness. Indeed, for badly conditioned linear systems arising from some applications, they are currently the only feasible solution methods. In many other cases, direct methods are often the method of choice because, on the other hand, finding and computing a good preconditioner for an iterative method can be

computationally more expensive than using a direct method. Furthermore, direct methods provide an effective mean of solving multiple systems with the same **A** but different right-hand sides **b**, as in this case the factorization step needs to be performed only once.

Sparse direct methods solve systems of linear equations by factorizing the coefficient matrix A, generally employing graph models to try to minimize both the storage needed and the amount of calculation performed. Sparse direct solvers follow a number of distinct phases (Figure 40). Although the exact subdivision depends on the algorithm and software being used, a common subdivision is given by:

1. An analysis phase that investigates the matrix structure to determine a pivot sequence and build a dependency graph, a data structure needed for efficient factorization. Finding a good pivot sequence can significantly reduce both memory requirements and the number of floating-point operations required for next steps. Most advanced codes have different capabilities, i.e. permutations and scaling of the matrix prior to the factorization, as well as data partitioning and mapping for distributed parallel computing and memory usage estimation for In-core/Out-of-Core computation
2. A factorization phase that uses the pivot sequence to factorize the matrix
3. A solve phase that performs forward elimination followed by back substitution, by using the stored factors. The solve phase may include an iterative refinement for the improvement of the solution quality, as well as an error analysis.

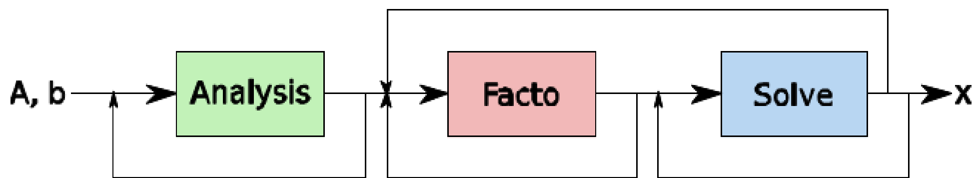


Figure 40. *Block diagram representing the three solution phases of a sparse direct solver [88].*

In a serial implementation, the factorization is usually the most time-consuming among the different phases, while the solve phase is generally significantly faster. This situation is still true for parallel executions both on distributed memory systems and shared memory computers.

6.2 Frontal solvers

A frontal solver [76] is an approach to the solution of sparse linear systems that automatically avoids a large number of operations involving zero terms. A frontal solver builds a LU or Cholesky decomposition of a sparse matrix, given as the assembly of element matrices, by assembling the matrix and eliminating equations only on a subset of elements at a time. This subset is called front and it is essentially the transition region between the part of the system already processed and the part to be processed. In the original method, the whole sparse matrix is

never created explicitly and only parts of the matrix are assembled as they enter the front. Processing the front involves dense matrix operations, which use parallel hardware efficiently. In a typical implementation, only the front is in memory, while the factors in the decomposition are written into files. The element matrices are read from files or created as needed and discarded.

In an unifrontal scheme, the factorization proceeds as a sequence of partial factorizations and eliminations on dense submatrices, called frontal matrices. For assembled systems, the frontal matrices can be written as:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

where all rows in frontal matrices are fully summed (no further contributions are expected to these rows) and the first block column is fully summed. This means that the pivots can be chosen from anywhere in the first column, and numerical pivoting with arbitrary row interchanges can be accommodated within these columns [77,78].

Let pivots to be chosen from a square matrix $\mathbf{A}_{11}$. Then $\mathbf{A}_{11}$ is factored and the Gaussian elimination overwrites $\mathbf{A}_{21}$, and the following Schur complement:

$$\mathbf{A}_{22} - \mathbf{A}_{21}{\mathbf{A}_{11}}^{-1}\mathbf{A}_{12}$$

is computed by using dense matrix kernels. The submatrices, consisting of rows and columns of the frontal matrix from which pivots have not been chosen yet, form the contribution block. In the case above, this is the same as the Schur complement matrix. At the next stage, further rows from the original matrix are assembled with the Schur complement to form another frontal matrix. The entire sequence of frontal matrices is retained in the same working array. Data movement is limited to assembling rows of the original matrix into the frontal matrix, and storing rows and columns as they become pivotal. One important advantage of the method is that only this single working array needs to reside in memory.

An example is shown in Figure 41, where two pivot steps have already been performed on a 5×7 frontal matrix (by computing the first two rows of $\mathbf{U}$ and columns of $\mathbf{L}$, respectively) and the columns are in pivotal order. Entries in $\mathbf{L}$ and $\mathbf{U}$ are shown in lower case. Row 6 has just been assembled into the current 4×7 frontal matrix, shown as a solid box. Columns 3 and 4 are now fully summed, because a column is fully summed if it has all of its nonzero elements in the frontal matrix, and can be eliminated. After this step, rows 7 and 8 must both be assembled before columns 5 and 6 can be eliminated (the dashed box, a 4×6 frontal matrix containing rows 5 through 8 and columns 5, 6, 7, 8, 9, and 12). The frontal matrix is stored without the zero columns, that is, columns 6 and 7 in the dashed box. The dotted box shows the state of the frontal matrix where the next four pivots can be eliminated. To factor the 12×12 sparse matrix in Figure 41, a 5×7 dense working array is sufficient to hold all the frontal matrices. The unifrontal

method works well for matrices with small pattern. But, for matrices with large pattern, the frontal matrices may be very large and an unacceptable amount of fill-in may occur.

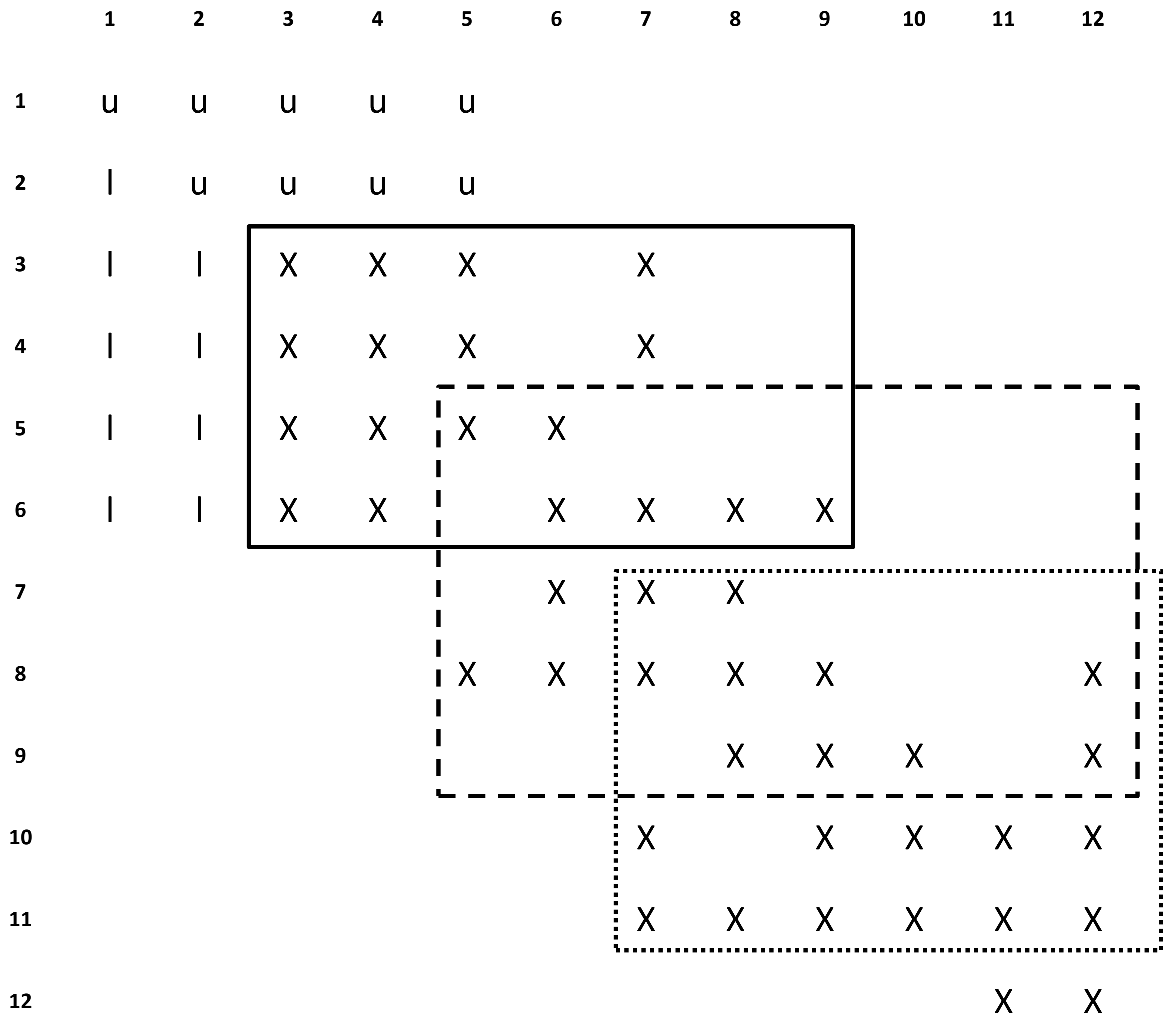


Figure 41. *Factorization of a 12 × 12 sparse matrix. Two pivot steps have already been performed on a 5 × 7 frontal matrix [78].*

6.3 Multifrontal solvers

A multifrontal solver, proposed by Duff and Reid [79], is an improvement of the frontal solver that uses several independent fronts at the same time. The fronts can be elaborated by different processors on parallel hardware, by exploiting a high-level parallelism. Nonetheless, as in the unifrontal method, the multifrontal method exploits low-level parallelism and vectorization techniques using the dense matrix kernels on frontal matrices. However, the frontal matrices in

the multifrontal method are generally smaller and denser than in the unifrontal method. The following three phases are performed:

- a reordering phase, through a minimum degree algorithm, in order to reduce number of zeroes stored in factor matrices (fill-in). The ordering is combined with a symbolic analysis to generate an assembly tree [77].
- a numerical factorization, based on the generated assembly tree
- a solve step, by forward substitution and backward elimination

The meaning of elimination tree can be derived from the graph theory, along to how this is related to Gaussian elimination and parallelization.

A non-directed graph (V, E) is associated to any sparse symmetric matrix $\mathbf{A}$, of order n (Figure 42). V is a set of nodes and E is the set of edges, that express the connections between the elements of V.

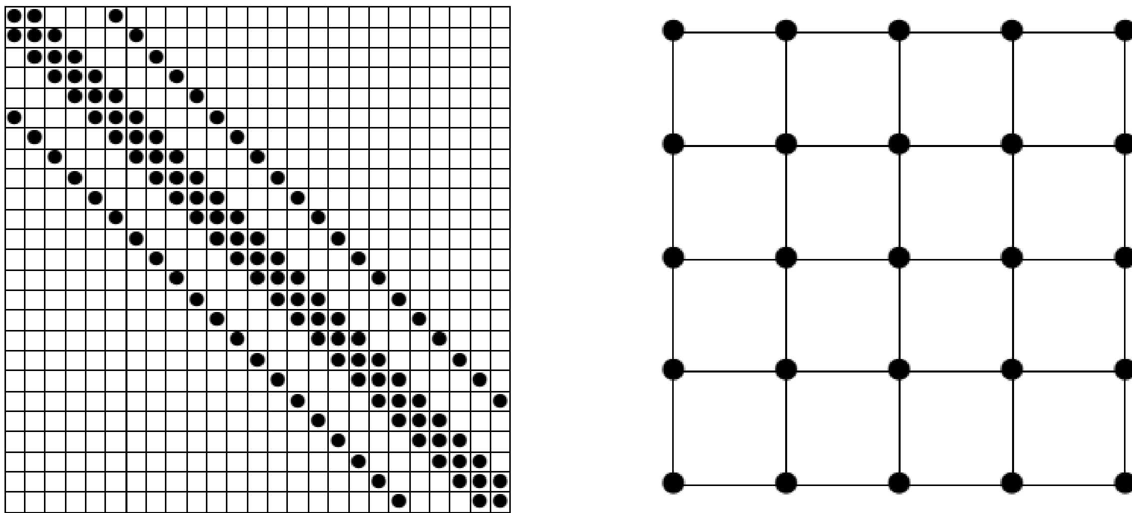


Figure 42. A coefficient in matrix A correspond to a node in the graph G. For each nonzero a_{ij} in A, there exist an edge between nodes i and j in the graph G [82].

Nodes can be labeled from 1 to n, such that each pair $(i, j) \in E \iff a_{ij}$ (or a_{ji}) is nonzero. A sequence of distinct edges (i_1, i_2), (i_2, i_3), …, (i_{k-1}, i_k) is a path. A path is a cycle if $i_k = i_1$ for > 2 . An acyclic graph is a graph without cycles. If there is a path between any pair of nodes, the graph is connected. A tree is defined as an acyclic connected graph. A rooted tree is defined by choosing a node, called root, which gives an orientation to the tree. For each tree, father nodes, child nodes and leaf nodes (nodes without descendants) can be defined.

A tree associated with Gaussian elimination is called an elimination tree. Several properties of trees associated with a tree generation algorithm are also given in [81]. Let $\mathbf{A}^{(k)}$ be the modified

matrix from rows and columns k to n after elimination by pivots $1, 2, 3, \ldots, k-1$. The essence of the multifrontal approach to Gaussian elimination is that the elimination operations corresponding to pivot k can be performed as soon as all the entries in the first row and first column of $\mathbf{A}^{(k)}$ have been calculated. Thus, it is possible to eliminate several variables at a node, without waiting for all the entries in $\mathbf{A}^{(k)}$ to be computed.

The entries which are modified by the descendants of node k are included in a submatrix of $\mathbf{A}^{(k)}$ *determined by the rows and columns with nonzeros in row k and column k of* $\mathbf{A}^{(k)}$.

Thus, the entries that are modified by the child nodes correspond to nonzeros in the child's pivot row and column. It is equivalent to assert that, for any child j of node k, the sparsity structure of column j in rows k to n is included in the sparsity structure of column k of $\mathbf{A}^{(k)}$.

Computation corresponding to tree nodes which are not ancestors or descendants of one another are independent.

In this way, it is possible to exploit the tree parallelism. An example of an assembly tree corresponding to the following sum:

$$\mathbf{A} = [[\mathbf{B}_1 + \mathbf{B}_2] + [\mathbf{B}_3 + \mathbf{B}_4]] + [[\mathbf{B}_5 + \mathbf{B}_6] + [\mathbf{B}_7 + \mathbf{B}_8]]$$

is given in Figure 43.

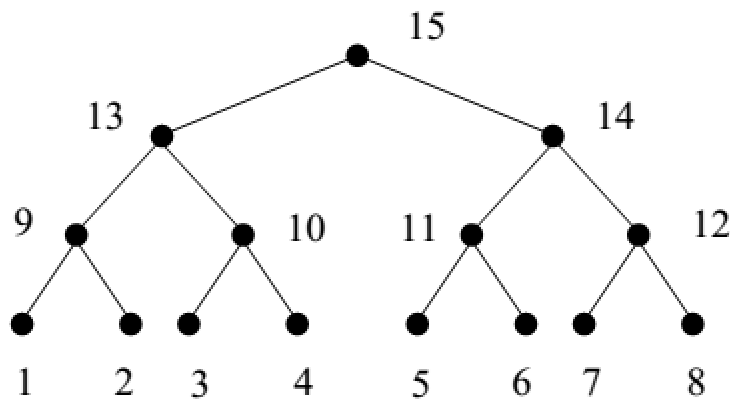


Figure 43. *Assembly tree corresponding to the sum A [78,82].*

The multifrontal method associates an operation, based on the properties of the tree, to each node/edge of the elimination tree [80]. In particular, each edge of the elimination tree corresponds to summing contributions from the children of the node with the rows and columns of the node itself. Consider a node k, which is not a leaf node. Then the contributions of the children of this node will change the entries in $\mathbf{A}^{(k)}$ corresponding to the nonzero entries in column k. A matrix (frontal matrix) is associated to each node k,. The frontal matrix is determined by the nonzero elements in the pivot row and column, respectively, such that the

contributions from the previous eliminations of the descendants of node k can be summed in the matrix. The outer products from the elimination by pivot k are then performed within the frontal matrix of node k.

An example of a matrix and its associated assembly tree is reported in Figure 44. From the initial matrix, an assembly tree with three nodes (each corresponding to one supernode) is derived. The two first independent leaf nodes contribute to the computation of the third.

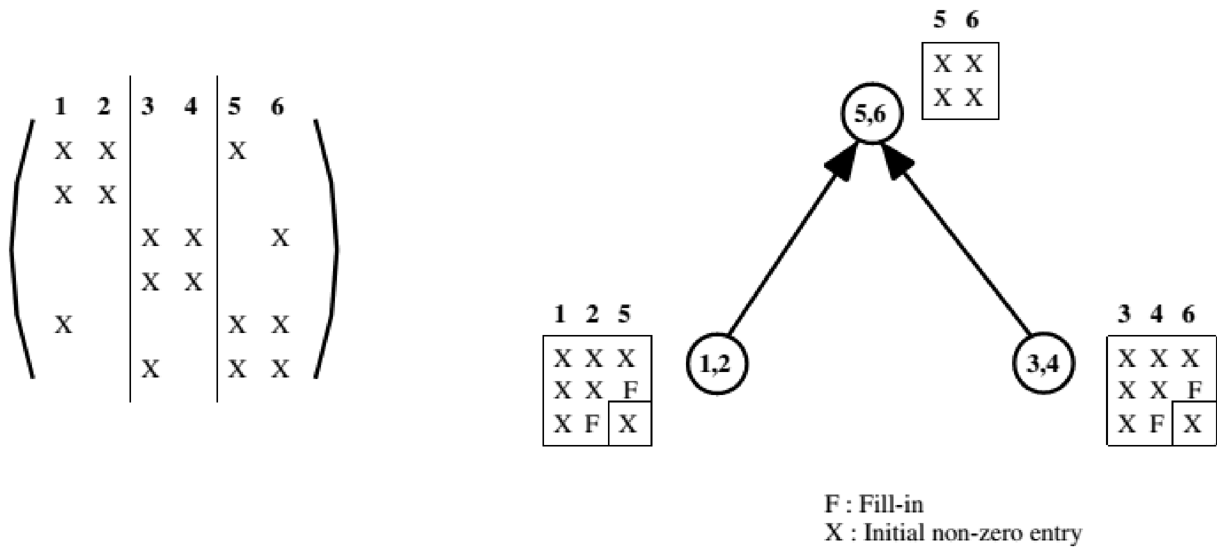


Figure 44. *A matrix with three supernodes* $(\{1,2\},\{3,4\},\{5,6\})$ *and the associated assembly tree [84].*

The order of a frontal matrix is given by the number of nonzeros below the diagonal in the first column of the supernode associated with the tree node. Each frontal matrix is divided into two parts:

- the factor block, called fully summed block, which corresponds to the variables factorized when the elimination algorithm processes the frontal matrix
- the contribution block which corresponds to the variables updated when processing the frontal matrix.

Once the partial factorization is complete, the contribution block is passed to the parent node. When contributions from all children are available on the father node, they can be assembled (i.e. summed with the values contained in the frontal matrix of the father), as in Figure 45.

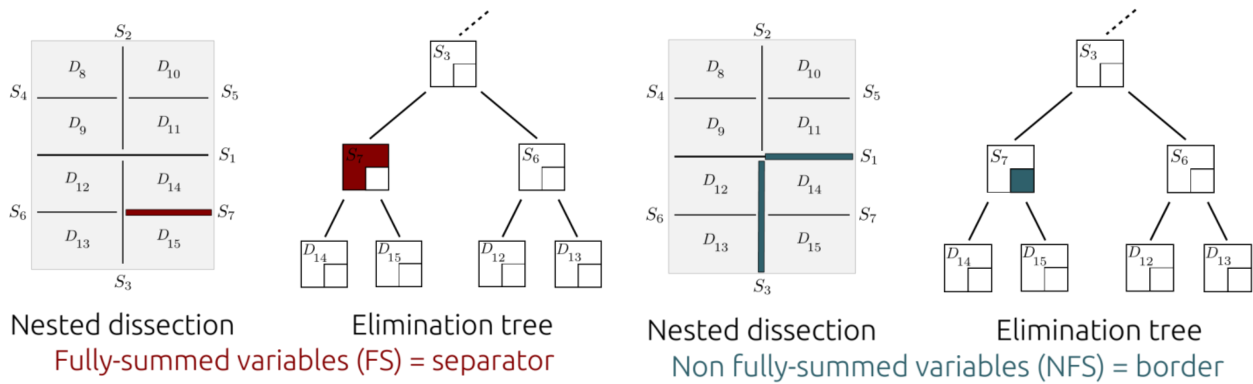


Figure 45. *Nested dissection generates the elimination tree [84].*

The elimination algorithm exploits a postorder traversal search of the assembly tree, in which father nodes are not processed before their children [82]. It uses three areas of storage in a contiguous memory space, one for the factors, one to stack the contribution blocks, and another one for the current frontal matrix [85]. During the tree traversal, the memory space required by the factors is always growing, while the stack memory (containing the contribution blocks) varies depending on performed operations. When the partial factorization of a frontal matrix is processed, a contribution block is stacked and the size of the stack increases; conversely, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are popped out of the stack and its size decreases. The stack memory is thus very dependent on the assembly tree topology.

It is well known that, despite their good reliability and numerical robustness, sparse direct solvers pose heavy requirements in terms of computational and memory resources. The reason is that the factors of a sparse matrix are much denser due to the appearance of fill-in, as new nonzero coefficients are introduced during the factorization process [82].

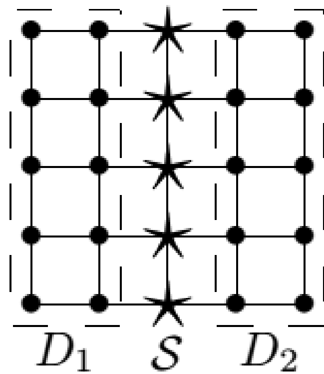
Reordering (i.e., renumbering the unknowns of a sparse linear system) is a well-known technique to reduce the fill-in in the final factor matrices, and this has a significant impact on the active memory size. In [85], the effect of reordering techniques on the active memory size is reported. In particular, the active memory size is largely dependent on the shape of assembly trees resulting from the reordered matrix.

Many methods and algorithms have been proposed in order to reduce the amount of fill-in and, among them, one of the most commonly used is the nested dissection method [86]. This method can be summarized in three steps:

- assembly of an undirected graph, in which nodes represent rows and columns of the matrix, and each edge represents a nonzero entry in the sparse matrix representing the system

- recursively partitioning of the graph into subgraphs using separators, defined as small subsets of nodes that, when removed, allow for the graph to be partitioned into subgraphs with almost the same fraction of the number of nodes.
- factorization, by ordering the elimination of the variables based on the recursive structure of the partition: in particular, each of the two subgraphs formed by removing the separator is eliminated first, and then the separator vertices are eliminated.

Given the sparse, symmetric matrix $\mathbf{A}$ of size n, reported as an example in Figure 46, its structure can be represented with an adjacency graph, i.e., a graph $G(V, E)$ containing n nodes (one for each coefficient in $\mathbf{A}$) and edges (i, j) for all $a_{ij} \neq 0$. The fill-in can then be easily modeled by using this graph. Specifically, eliminating a variable of $\mathbf{A}$ lead to the elimination of the associated node from G along with all incident edges. At the same time, new edges that connected the neighbors of the eliminated nodes must be built. These newly introduced edges represent fill-in coefficients. Then, assume that a node separator S of G is computed, i.e., a subset of nodes which, if removed, splits the graph into two subgraphs D_1 and D_2. Assume that $\mathbf{A}$ is permuted in such a way that all the variables in D_1 are eliminated first, then all variables in D_2 are eliminated and, finally, all variables in S are eliminated. Because all the neighbors of vertices in D_1 are either in D_1 or in S, no fill-in will be generated inside the submatrix that connects the variables in D_1 to variables in D_2.



(a) One level of nested dissection based on a 5x5 square grid mesh.

$A_{1,1}$		
0	$A_{2,2}$	
$A_{S,1}$	$A_{S,2}$	$A_{S,S}$

(b) Matrix A is reordered with respect to the nested dissection.

Figure 46. *One level of nested dissection [82].*

Because fill-in that occurs inside the diagonal blocks $\mathbf{A}_{1,1}$ and $\mathbf{A}_{2,2}$ may still be excessive, this procedure can be recursively and independently applied to the two subgraphs corresponding to D_1 and D_2, until the fill-in inside diagonal blocks can be considered negligible.

The application of nested dissection algorithm to a graph generates a separators tree which implicitly defines a fill-in reducing permutation of the input matrix. This is shown in Figure 47,

where $S_i = D_i$ is assumed for the leaves of the tree, i.e. S_i is a separator that splits D_i into two empty subdomains.

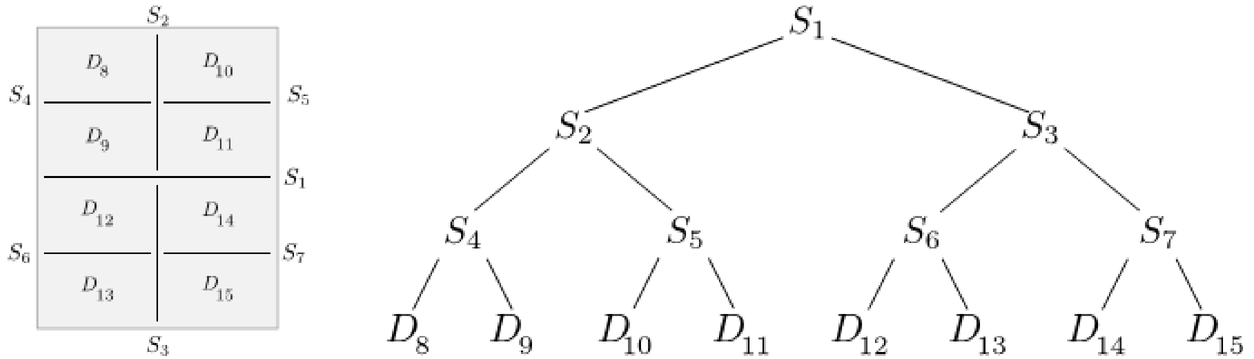


Figure 47. Each separator splits the graph in two subgraphs, until the fill-in inside diagonal blocks can be considered negligible (leaf nodes) [82].

This procedure describes the key idea of the nested dissection method although modern nested dissection ordering tools employ more sophisticated algorithms [86].

The separators tree can also be regarded as an elimination or assembly tree that defines the dependencies between the variables of a matrix and thus, implicitly, the order in which they have to be eliminated. Specifically, the elimination tree states that the elimination of the variables associated with a node (or with the corresponding separator) only affects variables associated with ancestor nodes, i.e. nodes along the path that connects the eliminated node to the root of the tree. Therefore, all the pivotal orders defined by topological traversals of the tree are equivalent in the sense that they produce the same amount of fill-in.

Each frontal matrices is associated with each node of the tree and is formed by coefficients related to the variables associated with that node as well as to their neighbors. Note that the neighbors include also variables that have become such due to fill-in introduced by the elimination of previous variables.

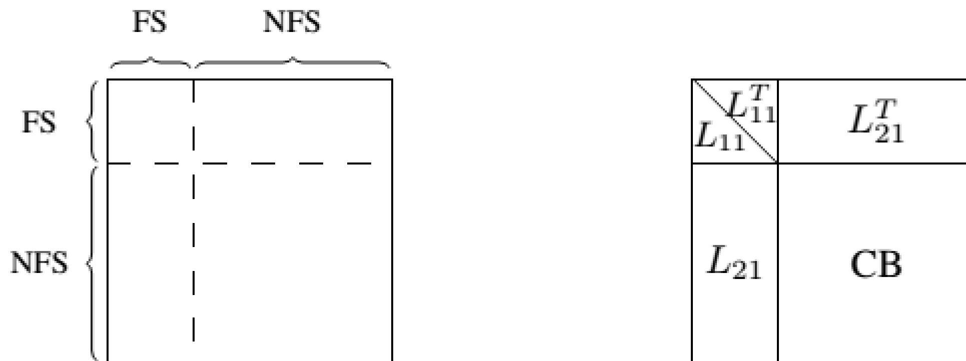
Based on the introduced concepts, the multifrontal factorization consists in a topological order traversal of the tree (i.e., bottom-up) where, each time a node is visited, two operations are performed:

- assembly: the frontal matrix is formed by summing the coefficients in the rows and columns of the variables associated with the tree node with coefficients produced by the factorization of its children.
- factorization: once the frontal matrix is formed, a partial Cholesky factorization is performed on it in order to eliminate the variables associated with the tree node. The result of this operation is a set of rows of the global factor and a Schur complement, also

commonly referred to as contribution block (CB), containing coefficients that will be assembled into the father node.

As shown in Figure 48, the set of variables in a frontal matrix can be split into two subsets:

- fully-summed (FS) variables are the variables associated with the tree node or, equivalently, with the corresponding separator. They are called fully-summed because the rows and columns in the front are up to date with respect to previously eliminated variables, that is, variables that have been eliminated at descendant nodes.
- non fully-summed (NFS) variables: this subset contains the neighbors of the fully-summed variables and is formed by pieces of separators belonging to ancestor nodes which form a border around the fully-summed variables. For the frontal matrix at node S_7 of the tree in Figure 47, this border is formed by half of the S_1 separator and the entire S_3 separator; variables in this border have become neighbors of those in S_7 due to fill-in coefficients introduced by the eliminations of variables in S_{14} and S_{15}.



(a) Before elimination: the front is assembled. FS variables are ready to be eliminated. NFS variables will be updated.

(b) After elimination: FS variables are eliminated, CB is the Schur complement that will be assebled into the father front.

Figure 48. *Structure of a front [82].*

6.5 MUMPS software package

MUMPS (MUltifrontal Massively Parallel Solver) is a software package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric [87-91]. MUMPS implements a direct method based on a multifrontal approach which performs a direct factorization:

$$\mathbf{A} = \mathbf{LU}$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ an upper triangular matrix.

If the matrix is symmetric, then the factorization

$$\mathbf{A} = \mathbf{LDL}^T$$

where $\mathbf{D}$ is block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed.

The parallel version of MUMPS requires MPI for message passing communication and use BLAS, BLACS, and ScaLAPACK [13,14,15] libraries as dense factorizations kernels. Other features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, out-of-core capability, parallel analysis, detection of null pivots, basic estimate of rank deficiency and null space basis for symmetric matrices, and computation of a Schur complement matrix. MUMPS offers several built-in ordering algorithms, a tight interface to some external ordering packages such as PORD [92], SCOTCH [94] or METIS [93], and the possibility for the user to input a given ordering. MUMPS is available for real or complex arithmetic, both in single and double precision computations. Recent experimental functionalities involve the computation of the determinant, the computation of some entries in the inverse of the matrix and the possibility to exploit sparsity of the right-hand sides, in order to reduce the amount of floating-point operations and accesses to the factor matrices.

From a functional point of view, MUMPS distributes the work tasks among the processors, and an identified processor (the host) is in charge of most of the analysis phase, to map and broadcast the matrix to the other processors (slaves) in the case where the matrix is centralized, and to collect the solution [87-91]. The system $\mathbf{Ax} = \mathbf{b}$ is solved in three main steps:

- **Analysis**:
  A preprocessing phase is performed, through an ordering according to the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$ and a symbolic factorization. Both parallel and sequential implementations of the analysis phase are available. A mapping of the multifrontal computational graph is then computed and used to estimate the number of operations and memory necessary for factorization and solution. Matrix $\mathbf{A}_{pre}$ denotes the preprocessed matrix.
- **Factorization**:
  During factorization, a direct factorization $\mathbf{A}_{pre} = \mathbf{LU}$ or $\mathbf{A}_{pre} = \mathbf{LDL}^T$ is computed, depending on the symmetry of the preprocessed matrix. The original matrix is initially distributed onto the available processors by following the mapping of the dependency graph, the so called elimination tree. Then, the numerical factorization is performed as a sequence of dense factorizations on frontal matrices. In addition, to standard threshold pivoting and two-by-two pivoting, an option allows for static pivoting. The elimination tree also expresses independency between tasks and enables multiple fronts to be processed in parallel (multifrontal approach). After the factorization, the factor matrices are kept distributed, both in local memory or on hard disks memory, ready to be used in the following solution phase.
- **Solution:**

The solution of $\mathbf{LU}\mathbf{x}_{pre} = \mathbf{b}_{pre}$ or $\mathbf{LDL}^T\mathbf{x}_{pre} = \mathbf{b}_{pre}$, where $\mathbf{x}_{pre}$ and $\mathbf{b}_{pre}$ are respectively the transformed solution $\mathbf{x}$ and right-hand side $\mathbf{b}$ associated to the preprocessed matrix $\mathbf{A}_{pre}$, is obtained through a forward elimination step:

$$\mathbf{Ly} = \mathbf{b}_{pre} \quad \text{or} \quad \mathbf{LDy} = \mathbf{b}_{pre}$$

followed by a backward elimination step

$$\mathbf{U}\mathbf{x}_{pre} = \mathbf{y} \quad \text{or} \quad \mathbf{L}^T\mathbf{x}_{pre} = \mathbf{y}$$

The right hand side $\mathbf{b}$ is firstly preprocessed and then broadcasted from the host to the working processors. Sparse right-hand sides might be used to limit the volume of data exchange during this step. A forward elimination and a backward substitution are then performed using the (distributed) factors, computed during factorization, to obtain $\mathbf{x}_{pre}$. The solution $\mathbf{x}_{pre}$ is finally post-processed to obtain the solution $\mathbf{x}$ of the original system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ can be either assembled on the host or kept distributed on the working processors. Iterative refinement and backward error analysis are among post-processing options of the solution phase.

Each of these phases can be called separately and several instances of MUMPS can be handled simultaneously. MUMPS allows for the host processor to cooperate to the factorization and solve phases, as any other processor. For both the symmetric and the unsymmetric algorithms, a fully asynchronous approach with dynamic scheduling of the computational tasks is exploited. Asynchronous communication is used to enable overlapping between communication and computation. Dynamic scheduling was initially chosen to accommodate numerical pivoting in the factorization. The other important reason for this choice was that, with dynamic scheduling, the algorithm can adapt itself at execution time to remap work and data to more appropriate processors. In fact, the main features of static and dynamic approaches are combined. In particular, based on the estimation obtained during the analysis step, a part of computational tasks is mapped. Other tasks are dynamically scheduled at execution time. Also the main data structures (the original matrix and the factors) follow the same scheme and are partially mapped during the analysis phase.

6.6 Improvements of shared memory parallelism

In the context of a joint project developed by the MUMPS team in Bordeaux, Lyon and Toulouse, and supported by many French institutions, as CERFACS, CNRS, ENS Lyon, INPT(ENSEEIHT)-IRIT, INRIA and University of Bordeaux, new functionalities that exploit shared memory parallelism have been implemented and developed by MUMPS team. This chapter summarizes these results.

There are typically two sources of parallelism in multifrontal methods. From a coarse-grain parallelism point of view, elimination trees are directed acyclic graphs (DAG) that define dependencies between their fronts. Then, the structure of tree offers an inner parallelism, which consists in factorizing different independent fronts at the same time. This is called tree

parallelism. From a fine-grain parallelism point of view, the partial factorization of a frontal matrix at a given node of the elimination tree can also be parallelized: this is called node parallelism. In a distributed-memory environment, the MUMPS solver implements both these types of parallelism. However, tree parallelism decreases near the root, where node parallelism generally increases because frontal matrices become bigger [96].

Preliminary results on how to mix distributed memory (as MPI processes) with shared memory models (OpenMP threads on multicore computers) show that using 4 to 8 threads per MPI process is a good compromise between performance and scalability issues. On usual matrix benchmarks, this combination can lead to a speedup of 6 on an eight-core machine. Furthermore, it has been observed that unsymmetric $\mathbf{LU}$ and Cholesky $\mathbf{LL}^T$ factorizations are generally more efficient than $\mathbf{LDL}^T$ factorization. These results have been obtained on a 96-core machine at INRIA, Bordeaux, France. More details are reported in [96,97].

The set of test problems used in mentioned experiments is given in Table 3, where N is the order of the matrix and NNZ its number of nonzero entries. The matrices come from Tim Davis collection (University of Florida) [98], from the GridTLSE collection (University of Toulouse) [99] and from geophysics applications [100,101]. For symmetric matrices, the associated unsymmetric problem is solved, although the value of NNZ reported only represents the number of nonzeros in the lower triangle matrix. A nested dissection ordering, METIS [93], is used to reorder the matrices and, by default, double precision computation is enabled. Five different areas can be defined in Table 3, separated by the horizontal lines. The first area corresponds to matrices for which there are very large fronts, arising from 3D finite element problems. The third area relays on matrices with many small fronts, i.e. arising from circuit simulation matrices. The second area corresponds to an intermediate situation between the two extremes. Last two zones corresponds to 3D and 2D geophysics applications, respectively.

Table 3. *Details about matrices used for performance tests [97].*

Matrix	Symmetry	Arithmetic	N	NNZ	Application field
3DSpectralwave	Symmetric	Real	680943	30290827	Materials
AUDI	Symmetric	Real	943695	77651847	Structural
Sparsine	Symmetric	Real	50000	1548988	Structural
Ultrasound80	Unsymmetric	Real	531441	33076161	Magneto-Hydro-Dynamics
DielFilterV3Real	Symmetric	Real	1102824	89306020	Electromagnetism
Haltere	Symmetric	Complex	1288825	10476775	Electromagnetism
ECL32	Unsymmetric	Real	51993	380415	Semiconductor device
G3_Circuit	Symmetric	Real	1585478	7660826	Circuit simulation
Qimonda07	Unsymmetric	Real	8613291	66900289	Circuit simulation
GeoAzur_3D_32_32_32	Unsymmetric	Complex	110592	2863288	Geo-Physics
GeoAzur_3D_48_48_48	Unsymmetric	Complex	262144	6859000	Geo-Physics
GeoAzur_3D_64_64_64	Unsymmetric	Complex	512000	13481272	Geo-Physics
GeoAzur_2D_512_512_512	Unsymmetric	Complex	278784	2502724	Geo-Physics
GeoAzur_2D_1024_1024_1024	Unsymmetric	Complex	1081600	9721924	Geo-Physics
GeoAzur_2D_2048_2048_2048	Unsymmetric	Complex	4260096	38316100	Geo-Physics

A multi-core based computer has been used for these tests, performed by Jean-Yves L'Excellent and Wissam Sid-Lakhdar [97]:

- Processor: 2x 4-Core Intel Xeon Processor E5520 2.27 GHz (Nehalem)
- Memory: 16 GigaBytes.
- Compiler: Intel compilers (icc and ifort) version 12.0.4 20110427.
- BLAS: Intel(R) Math Kernel Library (MKL) version 10.3 update 4.
- Location: ENSEEIHT-IRIT, Toulouse.

In Table 4, the effect of using threaded BLAS and OpenMP directives on the factorization time are reported; a comparison between these results with an MPI parallelization using MUMPS 4.10.0, with different combinations of threads per process for a total of 8 cores, is reported. In case of multiple threads per MPI process, threaded BLAS and OpenMP directives are used within each MPI process, in order to set the total number of threads to 8, that is, only one thread per core has been used.

On the first set of matrices (3Dspectralwave, AUDI, Sparsine, Ultrasound80), the ratio of large fronts over small fronts in the associated elimination tree is high. In this case, the best performance is achieved by setting more threads per MPI process, because node parallelism and the underlying multithreaded BLAS routines can reach their full potential on many fronts. On the second set of matrices, the ratio of large fronts over small fronts is medium; thus, the best computation times are generally reached when mixing tree parallelism at the MPI level with node parallelism at the BLAS level. On the third set of matrices, where the ratio of large fronts over small fronts is very small (that is, most fronts are small), using only one core per MPI process is often the best solution. To this extreme cases, tree parallelism determine the performance, whereas node parallelism does not bring any improvement, because parallel BLAS are not efficient on small fronts where there is not enough work for all threads working in a team. For Geoazur series of matrices, tree parallelism is more critical on 2D problems than on 3D problems: on 2D problems, the best results are obtained by increasing the number of MPI processes, and by using fewer threads per MPI process.

In general, OpenMP directives improve the amount of node parallelism (compare columns "Threaded BLAS" and "Threaded BLAS + OpenMP directives" in the "1 MPI x 8 threads" configuration), but the improvements are very limited. With the increasing number of cores per machine, this approach is only scalable when most of the work is done on very large fronts, as in the case of a very large 3D problem [97].

In other cases, tree parallelism is necessary. As message passing in MUMPS was mainly designed in order to exploit parallelism on a network of computing nodes rather than in multi-core processors, the availability of high performance multithreaded BLAS libraries could allow for both node and tree parallelism to be exploited at the shared memory level. The path consists in introducing multithreaded tree parallelism, as in [97].

__Table 4.__ Factorization times with different core-process configurations. Times are in seconds. N/A: the factorization failed for out of memory error. For each matrix, the best time obtained appears in bold [97].

Matrix	Sequential	Threaded BLAS only	Threaded BLAS + OpenMP directives			Pure MPI
	1 MPI x 1 thread	1 MPI x 8 threads	1 MPI x 8 threads	2 MPI x 4 threads	4 MPI x 2 threads	8 MPI x 1 thread
3DSpectralwave	2061.95	372.83	**371.87**	392.98	387.57	N/A
AUDI	1270.20	251.14	**249.21**	250.87	300.43	315.85
Sparsine	314.58	62.52	**61.87**	82.01	80.22	94.42
Ultrasound80	441.84	**89.05**	89.16	95.67	124.07	124.10
DielFilterV3Real	271.96	60.69	59.31	52.13	**47.85**	61.92
Haltere	691.72	121.29	120.81	**115.18**	140.34	145.55
ECL32	3.00	1.13	1.05	**0.93**	0.98	0.94
G3_Circuit	16.99	8.84	8.73	6.24	4.21	**3.61**
Qimonda07	25.74	27.42	28.49	18.21	9.63	**5.54**
GeoAzur_3D_32_32_32	75.74	16.09	**15.84**	16.28	18.68	19.62
GeoAzur_3D_48_48_48	410.78	73.90	72.96	**69.71**	**95.02**	106.86
GeoAzur_3D_64_64_64	1563.01	254.47	**254.38**	276.98	303.15	360.96
GeoAzur_2D_512_512_512	4.48	2.3	2.33	1.46	**1.4**	1.56
GeoAzur_2D_1024_1024_1024	30.97	11.54	11.65	8.38	6.53	**6.21**
GeoAzur_2D_2048_2048_2048	227.08	64.41	64.27	49.97	**43.33**	43.44

A few strategies have been implemented in order to overcome the limitations of the previous approach on small frontal matrices, where limited gains from node parallelism are observed. Even when there are large frontal matrices near the top of the tree, node parallelism may be insufficient close to the bottom of the tree and tree parallelism could be exploited instead. A solution is to introduce tree parallelism at thread level, allowing for different frontal matrices to be treated by different threads. Many algorithms for shared memory systems, such as proportional mapping and Geist-Ng algorithms, have been widely used in sparse direct methods. Variants of these methods have been adapted to MUMPS for the mapping algorithm in distributed memory environments. The final goal is to achieve load balancing and a high degree of concurrency among the processors, while reducing the amount of processor-to-processor data communication, through tree parallelism [102]. The Geist-Ng algorithm has been adapted to current multicore processors and hierarchical cache structures.

The AlgFlops algorithm is based on Geist-Ng algorithm is aimed to load balancing among threads. Given an arbitrary tree and P available processors, the main idea is to find the smallest set of branches in the tree such that the set can be partitioned into exactly P subsets, each one requiring approximately the same amount of work. The algorithm consists of two steps:

- Finding a layer L_{th} that separates the bottom of the tree from the top. This layer identifies a set of branches in the bottom of the tree that can be mapped onto the processors with an acceptable load balance
- Factorizing the elimination tree

The algorithm consists in an iterative procedure that, by starting from the root of the tree, replaces the largest subtrees by their children until finding a satisfactory layer, as shown in Figure 49. A detailed description of the algorithm is reported in [97].

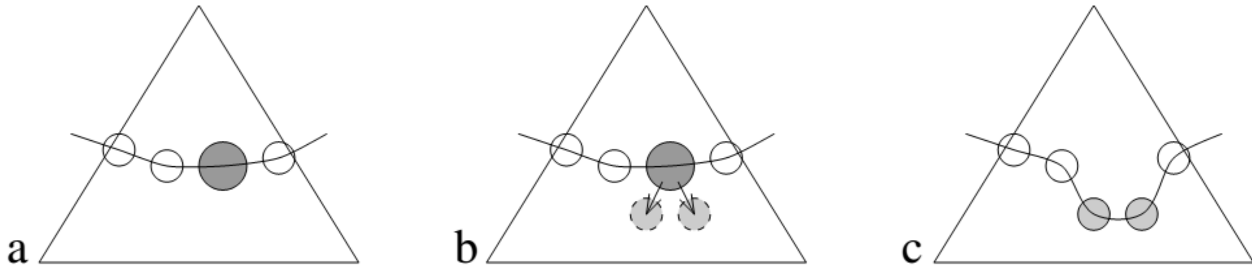


Figure 49. *One step in the construction of the layer L_{th}. AlgFlops replaces the largest subtrees by their children until finding a satisfactory layer, in which all branches imply a similar load for each processor.*

As concerning the numerical factorization, each thread picks the heaviest untreated subtree under L_{th} and factorizes it. If no more subtrees remain, the thread goes idle and waits for the others to finish. Whereas the Geist-Ng algorithm only used tree parallelism, the proposed AlgFlops algorithm uses tree parallelism under L_{th} but also node parallelism above it, because

- there are fewer nodes near the root of a tree, and more nodes near the leaves
- fronts near the root are usually larger than fronts near the leaves
- this approach matches pros and cons of each kind of parallelism granularity

Other improvements are introduced in AlgTime algorithm, where mono-thread factorizations are performed under L_{th}, with inexpensive synchronization of threads on L_{th}.

Some experimental results for L_{th}-based algorithm are reported in the Table 5. The application of these algorithms leads to further reductions in the factorization time on all tested sparse matrices, in addition to the improvements obtained by multithreaded BLAS dense algebra operations. Furthermore, in a shared memory hardware, L_{th}-based algorithm shows better performance than when message-passing only is used.

Table 5. *Experimental results with L_{th}-based algorithms on an eight-core hardware [97].*

Matrix	Serial reference	Threaded BLAS + OpenMP	Lth-based algorithms	
			AlgFlops	AlgTime
3DSpectralwave	2061.95	371.87	343.64	**339.78**
AUDI	1270.20	249.21	225.82	**210.10**
Sparsine	314.58	61.87	59.46	**57.91**
Ultrasound80	441.84	89.16	**77.06**	77.85
DielFilterV3Real	271.96	59.31	46.10	**44.52**
Haltere	691.72	120.81	102.17	**99.51**
ECL32	3.00	1.05	3.07	**0.72**
G3_Circuit	16.99	8.73	8.82	**3.02**
Qimonda07	25.74	28.49	27.34	**4.26**
GeoAzur_3D_32_32_32	75.74	15.84	13.02	**12.87**
GeoAzur_3D_48_48_48	410.78	72.96	64.14	**62.48**
GeoAzur_3D_64_64_64	1563.01	254.38	228.69	**228.12**
GeoAzur_2D_512_512_512	4.48	2.33	0.88	**0.84**
GeoAzur_2D_1024_1024_1024	30.97	11.65	5.37	**5.02**
GeoAzur_2D_2048_2048_2048	227.08	64.27	35.47	**34.56**

The gains of the proposed algorithms are very important on matrices arising from 2D finite element or circuit simulation problems, whose elimination trees typically present many nodes at the bottom and few medium-sized nodes at the top. In 3D finite element cases, even if the proposed L_{th}-based algorithms could offer some advantages, achieved gains are generally much smaller than in 2D cases, because in the 3D case most of the time is spent above L_{th} layer where the fronts are very large, while in 2D cases a remarkable part of the work is performed on small frontal matrices under L_{th} layer.

In conclusion, on a shared memory hardware, better performance are achieved when threads work on separate tasks in parallel, instead of setting them to cooperate on the same task, even at the price of a strong synchronization. Further improvements are obtained by implementing memory allocation policies, such as interleaving allocation policy on NUMA systems [97].

An existing fully-featured, distributed memory code has been adapted to shared-memory architectures, by using efficient multithreaded BLAS libraries in combination with the message-passing based tree parallelism, as in the original MUMPS 4.10.0. Furthermore, a new multithreading model has been introduced in main computation kernels through OpenMP models, at node parallelism level. As a further step, a higher-level tree parallelism has been exploited by taking advantage of the task graph arising from matrix factorizations. Since the task graph is a tree, serial kernels are effectively used to process independent subtrees in parallel, and multithreaded kernels are then applied to process the nodes from the top of the tree. An algorithm, based on a performance model of individual tasks, is used to determine the switching criterion from tree parallelism to node parallelism. This switching implies a cost for synchronization, that can be reduced by dynamically re-assigning idle CPU cores to active tasks. The performance of this algorithms depends also on compilation of the code and used BLAS libraries, thus a careful configuration is needed.

Large performance gains have been observed, while reusing the existing computational kernels and memory management algorithms, and keeping the existing numerical functionalities. Although this study is focused on computations with a single MPI process, MUMPS can be easily utilized in a hybrid distributed-memory/shared memory, by taking advantage of modern clusters with multi-core nodes. A preliminary experiment shows that, on 8 nodes with 8 cores each of the *bonobo* machine from the *plafrim* platform at Inria-Bordeaux [103], the factorization time of a GeoAzur 3D matrix of size 96x96x96 using 64 MPI processes (one core per MPI process) takes 657 seconds and falls down to 297 seconds by using 8 MPI processes with 8 cores per MPI [96]. There is still significant room for improvements, and the optimization of such a hybrid MPI-OpenMP approach will be the object of future work of MUMPS development team.

6.7 Improvements by Low-Rank approximation techniques

In the context of a joint project developed by the MUMPS team in Bordeaux, Lyon and Toulouse, and supported by many French institutions, as CERFACS, CNRS, ENS Lyon, INPT(ENSEEIHT)-IRIT, INRIA and University of Bordeaux, new functionalities that exploit low-rank approximation techniques have been implemented and developed by MUMPS team. This chapter summarizes the results obtained by Patrick Amestoy and Clement Weissbecker [80,83].

A low-rank matrix can be represented in a form which decreases its memory requirements and the complexity of involved basic linear algebra operations, such as matrix-matrix products. This is formalized in [109].

Let $\mathbf{A}$ be a matrix of size $m \times n$. Let k_ε be the approximated numerical rank of $\mathbf{A}$ at accuracy ε. A is a low-rank matrix if there exist three matrices $\mathbf{W}$ of size $m \times k_\varepsilon$, $\mathbf{Z}$ of size $n \times k_\varepsilon$ and $\mathbf{E}$ of size $m \times n$ such that:

$$\mathbf{A} = \mathbf{W} \cdot \mathbf{Z}^T + \mathbf{E}$$

Where $\|\mathbf{E}\|_2 \leq \varepsilon$ and $k_\varepsilon(m+n) < mn$.

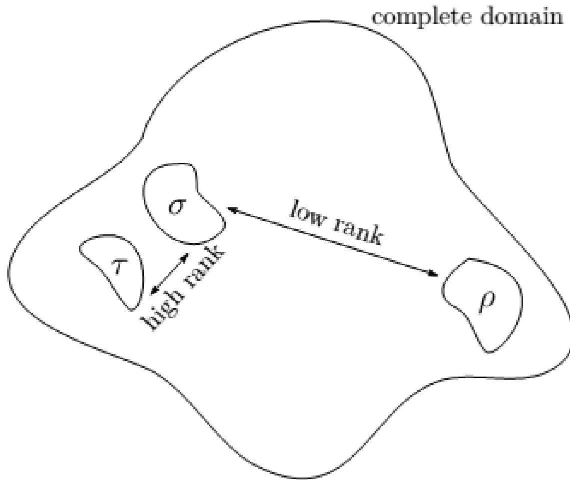
The numerical rank at precision ε, k_ε, and matrices $\mathbf{W}$ and $\mathbf{Z}$ can be computed with a Rank-Revealing QR (RRQR) factorization. Low-rank approximation techniques rely on the idea of neglecting $\mathbf{E}$ and of representing $\mathbf{A}$ as the product of $\mathbf{W}$ and $\mathbf{Z}^T$, at accuracy ε. Thus, $\mathbf{A} = \mathbf{W} \cdot \mathbf{Z}^T$ is the low-rank form of $\mathbf{A}$, and its accuracy can be controlled by a numerical parameter ε, the low-rank threshold. The choice of ε is critical, as k_ε should be small enough to guarantee that the low-rank form of $\mathbf{A}$ could use less memory than the standard form. Analogous criteria can be defined in the case where the objective is to reduce the complexity of operations involving the factorization of $\mathbf{A}$. Table 6 shows the requirements on the rank k_ε in order to reduce the operation count of the matrix-matrix product and of the triangular system solve, when the resulting matrix is stored in a dense form.

Table 6. *Cost of dense and low-rank basic linear algebra operations. A, L, $A_1 = W_1 \cdot {Z_1}^T$ and $A_2 = W_2 \cdot {Z_2}^T$ are $n \times n$, L is lower triangular, W_1, Z_1, W_2, Z_2 are $n \times k_\varepsilon$. The bracketing is critical to exploit the smaller dimension of each matrix [83].*

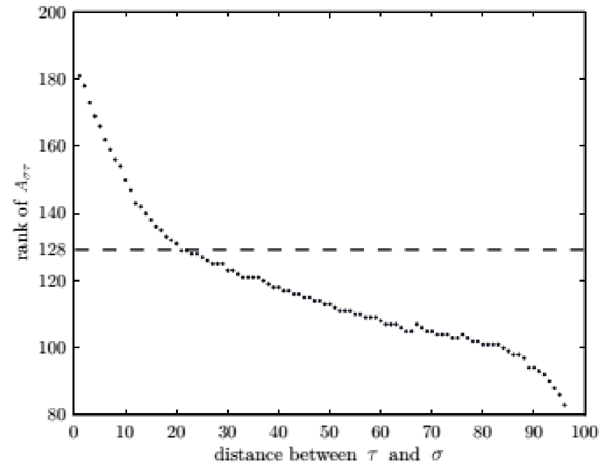
Operation type		Dense	Low-rank	Rank requirement
Cholesky factorization	LL^T	$n^3/3$	-	-
RRQR compression	$W_1{Z_1}^T$	$6kn^2 - 6k^2n + \frac{10}{3}k^3$	-	-
Triangular solve	$W_1({Z_1}^T L^{-T})$	n^3	$3kn^2$	$k < n/3$
Matrix-matrix product	$W_1({Z_1}^T W_2){Z_2}^T$	$2n^3$	$2kn^2 + 4k^2n$	$k < n/2$

In practice, matrices coming from applicative problems are not low-rank and cannot be directly approximated in a low-rank form. However, low-rank approximations can be performed on sub-blocks defined by an appropriately chosen partitioning of matrix indices [109,110]. Theoretical studies dealing with mathematical properties of the underlying operators, as well as geometrical and physical properties of the domain where the problem is defined, lead to the definition of a heuristic admissibility condition for low rank sub-blocks. The admissibility condition relies on the intuition that variable sets that are far away in the problem domain usually show weak interactions. In this case, the corresponding matrix block has a low rank [82].

An example is reported in Figure 50(a), where strong and weak interactions among different part of the domain are shown. Figure 50(b) shows that the rank of block $\mathbf{A}_{\sigma\tau}$ is a decreasing function of the geometrical distance between clusters σ and τ. This experiment has been done on a top-level separator of a 3D (128^3) wave propagation problem called Geoazur128, with square clusters of dimension 16x16, so that each sub-block has size 256. It shows that depending on the distance between clusters, a potential compression could be exploited. The dashed line at $y = 128$ sets a tradeoff for the storage of sub-blocks by a low-rank representation [83].



(a) Strong and weak interactions in the domain



(b) Correlation between graph distance and full accuracy block rank

Figure 50. *The rank of block $A_{\sigma\tau}$ is a decreasing function of the geometrical distance between clusters σ and τ [83].*

In practical cases, admissibility conditions according to geometrical or physical aspects of the problem cannot be easily exploited. In [82], a technique based on algebraic properties of the matrix is presented, by taking into account, for example, the efficiency of basic linear algebra kernels on the $\mathbf{W}$ and $\mathbf{Z}$ matrices of the low-rank form.

In particular, a new format called Block Low-Rank (BLR) has been studied and compared to other low-rank formats, as Hierarchical (H) matrices and Hierarchically Semi-Separable (HSS) matrices [110]. This structure, based on a non-hierarchical blocking of the matrix, can be exploited within internal data structures of the multifrontal method, and it has been implemented in an experimental version of MUMPS software in order to decrease the memory consumption and the operation count of the solver [111].

Figure 51 shows the global structure of the BLR representation of a dense Schur complement of order 128 x 128 corresponding to the top level separator of a 128 x 128 x 128 Laplacian problem, with a low-rank threshold set to 10^{-14}.

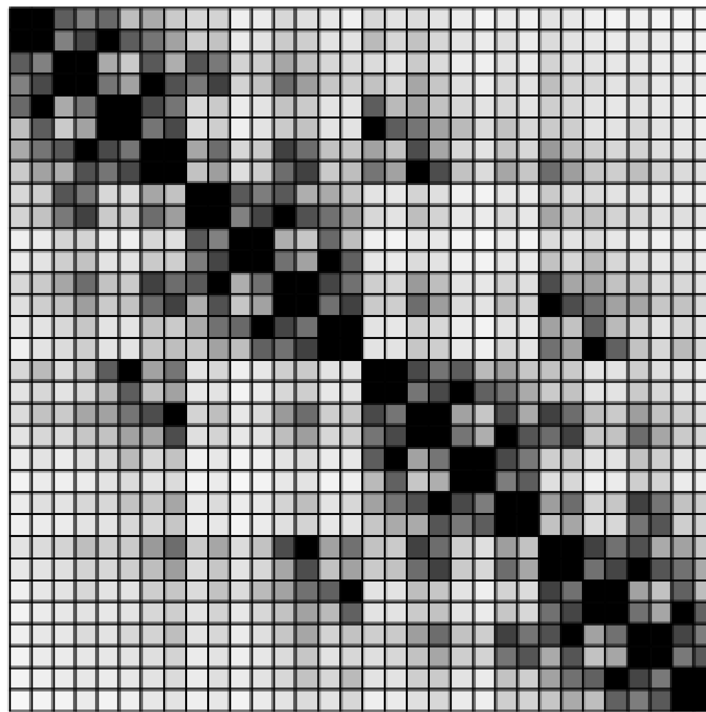


Figure 51. *Structure of a BLR matrix. The darkness of a block is proportional to its storage requirement. The lighter a block is, the smaller is the memory needed to store it. Each block in the matrix is of size 512 x 512 [83].*

An experimental comparison in [82,83] shows that the gains achieved with the BLR format are comparable to those obtained with other hierarchical formats. The cost of constructing the low-rank representation is much lower in the case of the BLR format with respect to the HSS and H

approaches; this provides an additional property which is very favorable in the context of a sparse, multifrontal solver involving operations with complex data access patterns.

6.8 PARDISO software package

The PARDISO package [118] offers a parallel solver for the direct solution of unsymmetric and symmetric sparse linear systems on shared memory multiprocessors. PARDISO employs a combination of left- and right-looking Level 3 BLAS supernode techniques [119] and has been included in Intel Math Kernel Library in 2006. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. Intel MKL version of PARDISO is based on PARDISO version from 2006, thus many new features and improvements of PARDISO 5.0.0, such as distributed memory factorization for some classes of matrices, are not available in the Intel MKL library. The PARDISO solver performs three tasks, an analysis and symbolic factorization step, a numerical factorization step, and a forward and backward substitution by optionally including iterative refinement.

For unsymmetric matrices, that are considered in this thesis, the solver first computes a non-symmetric permutation $\boldsymbol{P}_{MPS}$ and scaling matrices $\boldsymbol{D}_r$ and $\boldsymbol{D}_c$ with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [120]. In the second step, the solver computes a fill-in reducing permutation $\boldsymbol{P}$ based on the matrix $\boldsymbol{P}_{MPS}\boldsymbol{A} + (\boldsymbol{P}_{MPS}\boldsymbol{A})^T$, followed by the parallel numerical factorization:

$$\boldsymbol{QLUR} = \boldsymbol{P}\boldsymbol{P}_{MPS}\boldsymbol{D}_r\boldsymbol{A}\boldsymbol{D}_c\boldsymbol{P}$$

Where $\boldsymbol{Q}$ and $\boldsymbol{R}$ are supernode pivoting matrices. When the factorization algorithm reaches a point where it cannot factor the supernodes with such pivoting strategy, a pivoting perturbation strategy is used [121]. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

The PARDISO solver enables to use a combination of direct and iterative methods [122] to accelerate the linear solution process for transient simulations, as in thermal transient simulations (not considered in this thesis), in which many consecutive solutions of sparse systems are required. In such particular class of problems, the values of the nonzero coefficients in the matrix gradually change, thus keeping the same identical sparsity pattern. The analysis phase of the solvers has to be performed only once and the numerical factorizations become the important time-consuming steps during the simulation. PARDISO solver computes a numerical factorization $\boldsymbol{A} = \boldsymbol{LU}$ for the first system solution, then it applies the factors $\boldsymbol{L}$ and $\boldsymbol{U}$ for the next steps in a preconditioned Krylov-Subspace iteration. If the iteration does not converge, the solver automatically switches back to the actual numerical factorization.

6.9 Iterative methods

The term "iterative method" refers to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step [28]. In other words, an iterative method is a solution algorithm that, by starting from an arbitrary chosen initial guess $\mathbf{x_0}$, produces a better approximation $\mathbf{x_{k+1}}$ of the exact solution $\mathbf{x}$, at each k-th iteration. With an iterative method, it is possible to obtain a good approximation of the exact solution. An iterative method can be obtained by splitting of the matrix $\mathbf{A}$. If $\mathbf{A} = \mathbf{S} - \mathbf{T}$, then $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{Sx} = \mathbf{Tx} + \mathbf{b}$ and, at each iteration from $\mathbf{x_k}$ to $\mathbf{x_{k+1}}$, the solution of $\mathbf{Sx_{k+1}} = \mathbf{Tx} + \mathbf{b}$ should return $\mathbf{x_{k+1}}$.

A good $\mathbf{S} - \mathbf{T}$ decomposition should observe the following requirements:

- The vector $\mathbf{x_{k+1}}$ should be easy and inexpensive to compute. Thus, $\mathbf{S}$ should be a simple and invertible matrix, i.e. a diagonal or tridiagonal matrix
- The $\mathbf{x_k}$ sequence should converge to the exact solution $\mathbf{x}$. In other words, the error between current solution and the exact solution should decrease to zero

The requirements above are in conflict. By choosing $\mathbf{S} = \mathbf{A}$ and $\mathbf{T} = 0$, only one iteration could lead to the solution, but it could be difficult to solve the obtained linear system. The $\mathbf{S}$ matrix is named preconditioner, and a proper choice usually makes the difference between a fast convergence and a stagnation of the iterations.

Some simple choice of $\mathbf{S}$ could be:

- $\mathbf{S}$ = diagonal of $\mathbf{A}$ (Jacobi method)
- $\mathbf{S}$ = triangular part of $\mathbf{A}$ (Gauss-Seidel method)
- $\mathbf{S}$ = a combination of both (SOR method)
- $\mathbf{S} = \mathbf{L}_0\mathbf{U}_0$ (incomplete $\mathbf{LU}$ factorization)

Among iterative methods, projection methods are the most used [72]. The idea of projection techniques is to extract an approximated solution $\mathbf{x_m}$ to the $\mathbf{Ax} = \mathbf{b}$ problem from a subspace $\mathbf{x}_0 + \mathbf{K}_m$ of dimension m (subspace of candidate approximants). A typical technique is to impose orthogonality constraints to the residual vector $\mathbf{b} - \mathbf{A}\mathbf{x}_m$, at each iteration. In this family, Conjugate Gradient Algorithm (CG) and Generalized Minimal Residual Method (GMRES) are the best known iterative algorithms.

In general, the reliability of iterative techniques, when dealing with various applications, depends much more on the quality of the preconditioner than on the particular Krylov subspace accelerators used. A preconditioner is any form of implicit or explicit modification of an original linear system which makes it "easier" to solve by a given iterative method. In fact, since the rate at which an iterative method converges depends significantly on the spectrum of the coefficient matrix, iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called a preconditioner. A

good preconditioner improves the convergence of the iterative method, sufficiently to overcome the extra cost of computing and applying the preconditioner.

One of the simplest ways of defining a preconditioner is to perform an incomplete factorization of the original matrix A. This involves a decomposition of the form $\mathbf{A} = \mathbf{LU} - \mathbf{R}$, where L and U have the same nonzero structure as the lower and upper parts of $\mathbf{A}$, and $\mathbf{R}$ is the residual of the factorization. This incomplete factorization, known as $ILU(0)$, is inexpensive to compute, but it often leads to rough approximation that may cause many iteration to converge. As a general rule, more accurate ILU factorizations require fewer iteration to converge, but the preprocessing cost to compute factors are higher.

The first step when implementing iterative algorithms on a high-performance parallel computer is to identify the main operations, or kernels, that they require [72]. Five types of operations can be identified, which are:

- Setting up the preconditioner
- Matrix vector products
- Vector updates
- Dot products and reduction operations
- Preconditioning operations

In this list, the potential bottlenecks are setting up the preconditioner $\mathbf{M}$ and solving linear systems with $\mathbf{M}$, i.e., the preconditioning operation. A detailed review on how to implement these methods efficiently on shared memory computers and distributed memory computers is reported in [72].

Iterative methods are very different from direct methods in this respect. While the performance of direct methods, both for dense and sparse systems, is essentially determined by the factorization of the matrix relying on dense algebra suboperations, this operation is not present in iterative methods (although preconditioners may require a setup phase). Since such operations can be executed at very high efficiency on most computer architectures, a lower Flop rate for iterative than for direct methods will be achieved. Dongarra and Van der Vorst [73] give some experimental results about this, and provide a benchmark code for iterative solvers. Furthermore, the basic operations in iterative methods often use indirect addressing, depending on the data structure. Such operations also have a relatively low efficiency of execution, and are limited by hardware capabilities as memory bandwidth and latency. However, a lower efficiency of execution does not imply a long solution time for a given system. Furthermore, iterative methods are usually simpler to implement than direct methods, and since no full factorizations have to be stored in memory, they can handle much larger systems than direct methods.

It should be pointed out that not every method will work on every problem type, so knowledge of matrix properties is the main criterion for selecting an iterative method. Furthermore, different methods involve different kernels, and this may rule out certain methods depending on the

problem or target computer architecture. Below are short descriptions of each of the main iterative methods, along with brief notes on the classification of the methods in terms of the class of matrices for which they are most appropriate [75].

The following methods belong to the so-called stationary methods:

- **Jacobi**
  The Jacobi method is based on solving for every variable locally with respect to the other variables. One iteration of the method corresponds to solving for every variable once. The resulting method is easy to understand and implement, but convergence is slow. This method is easy to use, but unless the matrix is "strongly" diagonally dominant, this method is probably best only considered as a preconditioning in a nonstationary method.
- **Gauss-Seidel**
  The Gauss-Seidel method is similar to the Jacobi method, except that it uses updated values as soon as they are available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, but in general not competitive with the nonstationary methods. It can be applied to strictly diagonally dominant, or symmetric positive definite matrices. Parallelization properties depend on structure of the coefficient matrix. Different orderings of the unknowns have different degrees of parallelism; multi-color orderings may give almost full parallelism.
- **SOR**
  Successive Over-Relaxation (SOR) can be derived from the Gauss-Seidel method by introducing an extrapolation parameter ω. For its optimal choice, SOR may converge faster than Gauss-Seidel by an order of magnitude.
- **SSOR**
  Symmetric Successive Over-Relaxation (SSOR) has no advantage over SOR as a standalone iterative method; however, it is useful as a preconditioner for nonstationary methods.

The following methods belong to the so-called nonstationary methods:

- **Conjugate Gradient (CG).**
  The conjugate gradient method derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors. These vectors are the residuals of the iterations. They are also the gradients of a quadratic functional, the minimization of which is equivalent to solving the linear system. CG is an effective method, valid for symmetric positive definite matrices, since storage for only a limited number of vectors is required. Speed of convergence depends on the conditioning number. Inner products acts as a bottleneck in a parallel environment, because such reduction operations acts as synchronization points. The efficiency of this method on parallel hardware depends

weakly on the coefficient matrix, but strongly on the structure of the eventual preconditioner.

- **Minimum Residual (MINRES) and Symmetric LQ (SYMMLQ).**
  These methods are alternatives to CG for coefficient matrices that are symmetric but possibly indefinite. SYMMLQ will generate the same solution iterates as CG if the coefficient matrix is symmetric positive definite.
- **Conjugate Gradient on the Normal Equations: CGNE and CGNR.**
  CGNE solves the system $(\mathbf{A}\mathbf{A}^T)\mathbf{y} = \mathbf{b}$ for y and then computes the solution $\mathbf{x} = \mathbf{A}^T\mathbf{y}$. CGNR solves $(\mathbf{A}^T\mathbf{A})\mathbf{x} = \tilde{\mathbf{b}}$ for the solution vector $\mathbf{x}$ where $\tilde{\mathbf{b}} = \mathbf{A}^T\mathbf{b}$. When the coefficient matrix $\mathbf{A}$ is nonsymmetric and nonsingular, the normal equations matrices $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$ will be symmetric and positive definite, thus CG could be applied. The convergence may be slow, since the spectrum of the normal equations matrices will be less favorable than the spectrum of $\mathbf{A}$.
- **Generalized Minimal Residual (GMRES).**
  The Generalized Minimal Residual method computes a sequence of orthogonal vectors (as MINRES), and combines these through a least-squares solve and update. This method can be applied to general non-symmetric matrices. However, unlike MINRES and CG, it requires storing the whole sequence, so that a large amount of storage is needed. Furthermore, even if GMRES leads to the smallest residual for a fixed number of iteration steps, these steps become increasingly expensive to compute. For this reasons, restarted versions of this method are used, in which computation and storage costs are limited by specifying a fixed number of vectors to be generated. The restart point depends on matrix $\mathbf{A}$ and the right-hand side $\mathbf{b}$, and its correct setting requires skill and experience. GMRES requires only matrix-vector products involving the coefficient matrix, and the number of inner products grows linearly with the iteration number, up to the restart point. In an implementation based on a simple Gram-Schmidt process, the inner products are independent one another, thus leading to only one synchronization point. A more stable implementation, based on a modified Gram-Schmidt orthogonalization, needs one synchronization point per inner product.
- **BiConjugate Gradient (BiCG).**
  The Bi-Conjugate Gradient method generates two CG-like sequences of vectors, one based on a system with the original coefficient matrix $\mathbf{A}$, and one on its transposed $\mathbf{A}^T$. Instead of orthogonalizing each sequence, they are made mutually orthogonal, or "bi-orthogonal". As CG, this method uses limited storage. It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular, and there is a possibility that the method will break down. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration. This excludes all cases where the matrix is only given implicitly as an operator, since usually no corresponding transpose operator is available in such cases. Parallelization properties are similar to those for CG; the two matrix vector products (as well as the preconditioning steps) are

independent, so they can be done in parallel, or their communication stages can be packaged.

- **Quasi-Minimal Residual (QMR).**
  The Quasi-Minimal Residual method applies a least-squares solve and update to the BiCG residuals, thus leveling the irregular convergence behavior of BiCG. Also, QMR avoids the breakdown that can occur in BiCG. On the other hand, it does not effect a true minimization of either the error or the residual, and while it converges smoothly, it does not essentially improve on the BiCG. This method is applicable to non-symmetric matrices. Computational costs per iteration are similar to BiCG, but slightly higher. The method requires the transpose matrix-vector product, and parallelization properties are similar to BiCG.
- **Conjugate Gradient Squared (CGS).**
  The Conjugate Gradient Squared method is a variant of BiCG that applies the updating operations for the $\mathbf{A}$-sequences and the $\mathbf{A}^T$-sequences both to the same vectors. Ideally, this would double the convergence rate, but in practice convergence may be much more irregular than for BiCG. A practical advantage is that the method does not need the multiplications with the transpose of the coefficient matrix. Unlike BiCG, the two matrix-vector products are not independent and the number of synchronization points in a parallel environment is larger.
- **Biconjugate Gradient Stabilized (Bi-CGSTAB).**
  The Biconjugate Gradient Stabilized method is a variant of BiCG, like CGS, but using different updates for the $\mathbf{A}^T$-sequence in order to obtain smoother convergence than CGS. This method is applicable to non-symmetric matrices. Computational costs per iteration are similar to BiCG and CGS, but the method does not require the transpose matrix. In some cases, this method avoids the irregular convergence patterns of CGS while maintaining about the same speed of convergence.
- **Chebyshev Iteration.**
  The Chebyshev Iteration recursively determines polynomials with coefficients chosen to minimize the norm of the residual in a min-max sense. This method requires some explicit knowledge of the spectrum of the coefficient matrix; in the symmetric case the iteration parameters are easily obtained from the two extremal eigenvalues, which can be estimated either directly from the matrix, or from applying a few iterations of the Conjugate Gradient Method. The computational structure is similar to that of CG, but this method has the advantage of requiring no inner products, thus there are no synchronization points. An Adaptive Chebyshev method can be used in combination with methods as CG or GMRES, to continue the iteration once suitable bounds on the spectrum have been obtained from these methods.

In conclusion, stationary methods are older, simpler to understand and implement, but usually not as effective. In practical problems, preconditioned nonstationary methods are used.

7. Application of parallel direct solvers to FEM software

With the multicore era of 2000-2010, the cost of desktop parallel computing has been brought down. As a consequence, the diffusion of low-cost multicore and many-core hardware made parallel computing approachable not only to research institutes but also to industry. Furthermore, open source FEM packages integrated advanced parallel solvers for the solution of large sparse linear systems, largely using optimized numerical libraries targeting to different hardware, i.e. multicore computers, distributed memory computers, GPUs and others.

Following the success of the open source code, in recent years there has been a great effort by simulation software vendors in order to develop parallel versions of their commercial codes. In commercial FEM codes there are only few, but effective, potential points of intervention. Among all, the most important is the linear system solution, which can be seen as the inner and most frequent operation in the solution loop.

As an example, a typical transient simulation of an induction heating process might be represented as in Figure 52. A thermal transient study requires the solution of a sequence of time steps. At each time step, one or more iterations between a thermal problem solution and a time-harmonic electromagnetic problem solution are performed in order to achieve a refinement of the solution and to avoid propagation of errors. In general, both thermal and electromagnetic problem solutions could be non-linear, thus involving a Newton-Raphson process of successive approximation, through a sequence of linearized system solutions.

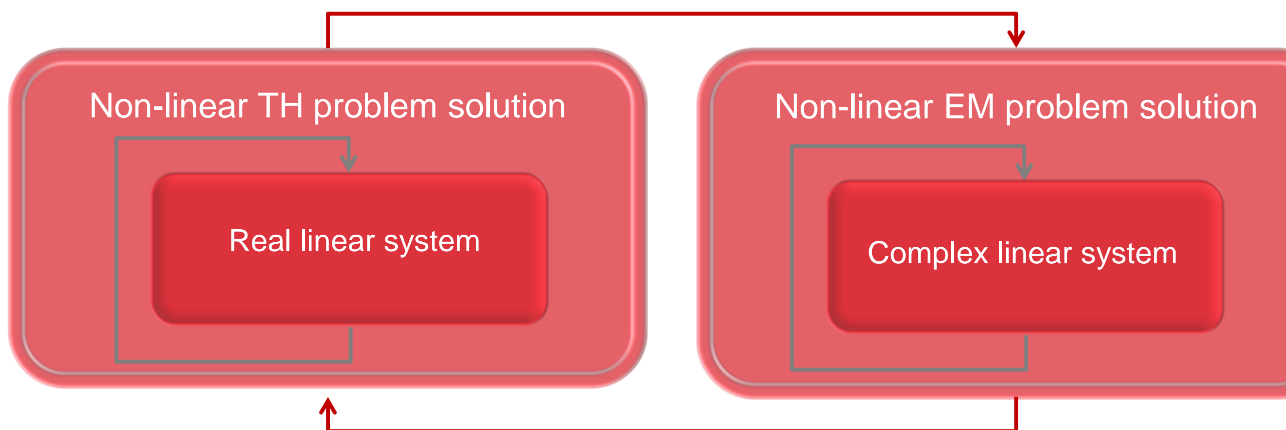


Figure 52. *Typical solution loop for a simulation of induction heating processes. A transient thermal problem is iteratively coupled to a steady-state time-harmonic electromagnetic problem. In general, both problems could be non-linear.*

- Non-Linear System : $[A(X)][X]=[B]$
- Principle : Solve $[R(X)]=[A(X)][X]-[B]=0$
- Non-Linear Iterations :
  - Solving : $\left[\frac{dR}{dX^T}(X_k)\right]\Delta X=-R(X_k)$
  - Incrementing solution : $X_{k+1}=X_k+\alpha\,\Delta X$

    Where α is relaxation coefficient
  - Convergence criterion : $\frac{\|X+\alpha\,\Delta X\|^2}{\|\Delta X\|^2}\leq\varepsilon$

Figure 53. *Solution scheme of a non-linear problem by Newton-Raphson method.*

Note that an optimization and parallelization of the "linear system solution" kernel can lead to a remarkable reduction on the overall simulation time. This idea has been exploited in cooperation with Cedrat Group [39], a company that provides electromagnetic simulation tools for electrical applications design, and MUMPS development team [87]. A study on how to accelerate the linear system solution has been carried out in this thesis, leading to the integration of different parallel solvers in Cedrat FEM simulation tool, named Flux [95]. In particular, two state-of-the-art parallel direct solvers for sparse linear systems solution have been tested on a set of large matrices, arising from the simulation of typical industrial application of induction heating processes.

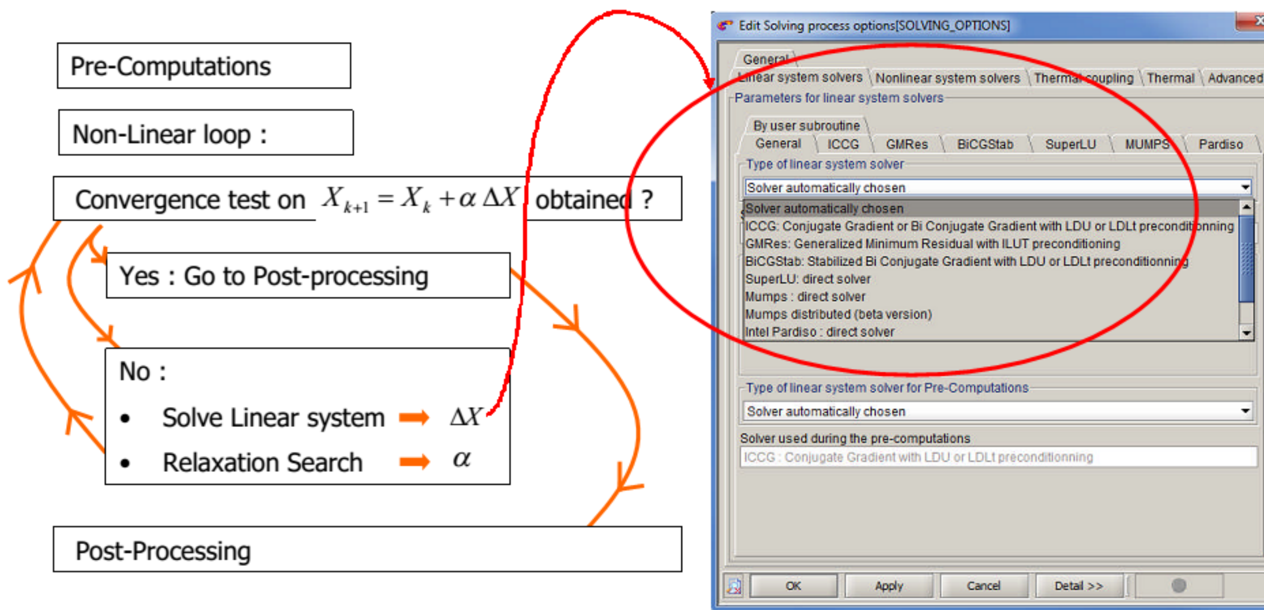


Figure 54. *Implementation of non-linear problem solution scheme in Cedrat Flux software. The work presented in this thesis contributed to the introduction of parallel computing capabilities for Cedrat Flux software.*

In order to compare the solver performance on different hardware configurations, all tests have been carried out for the solution of a quasi-stationary time-harmonic electromagnetic problem with constant material properties. However, reported conclusions can be considered of general validity, and they can easily be extended to both transient and steady state regime, electromagnetic, thermal or multiphysics coupled problem solutions.

Although iterative methods are widely applied to the solution of this kind of problem, the performance of direct solvers is investigated because this kind of solvers are the best suited in case of badly conditioned problems, as it usually happens when dealing with induction heating simulations.

Memory usage and factorization times for MUMPS (MUltifrontal Massively Parallel sparse direct Solver) [87], configured with multithreaded BLAS library on a multicore computer, have been compared with another popular sparse direct solver, Intel PARDISO (PARallel Direct Solver) [13]. Intel PARDISO targets shared memory multicore computers only, while MUMPS 4.10.0 targets distributed memory systems.

With the aim to build an efficient direct algorithm for the solution of large sparse complex matrices, further work has been carried out by MUMPS team, that developed an experimental version of the library. Tests have been performed by MUMPS team, both on workstation-size multicore hardware and on a modern cluster with multicore nodes. The author provided all tested matrices obtained by finite element models of induction heating processes. Different parallelization strategies and memory policies have been adopted for each different grain size level. Furthermore, the use of low-rank compression techniques allowed for a remarkable reduction of memory consumption during the factorization phase of the "linear system solution" kernel [82].

7.1 Benchmark matrices from induction heating simulations

A set of linear system solutions have been performed on matrices arising from finite element modeling of induction heating industrial applications, in order to characterize the behavior of MUMPS and PARDISO in terms of memory usage and factorization times, when run on different hardware configurations. Finite element models that simulate the heating of a graphite susceptor by a pancake coil and the induction hardening of a gear are chosen as test benchmarks [104,105,106]. Geometric model design, meshing and matrix building are performed by a commercial software [39]. Starting from the same geometry (in the following, named "Pancake" series or "Gear" series), model meshes are gradually refined in order to solve the same physical problems on different mesh sizes, leading to complex linear systems ranging from $320k$ to $2.5M$ degrees of freedom.

First benchmark: heating of a graphite susceptor by a pancake coil

In order to evaluate the performance of different sparse solvers for complex coefficient linear systems, the model of a pancake inductor and a graphite susceptor has been taken as the first benchmark problem. This system is actually used in a furnace for the production of solar grade silicon by the Directional Solidification System I-DSS [105]. In Figure 55, the model geometry is presented and Figure 56 shows its mesh. The numerical solution of the electromagnetic problem has been carried out by adopting a nodal A, AV formulation and by using the complex representation of the sinusoidal quantities [107, 53]. A formulation in terms of the magnetic vector potential A coupled with the scalar electric potential V has been chosen because it guarantees a precise solution of the eddy current distribution in models where highly permeable conducting region are not present [53, 108]. Usually, such magnetic vector potential formulation originates systems with more degrees of freedom than the (dual) scalar formulation. This model has been set up by using four different types of discretization, with a number of elements ranging from $600k$ up to $2.9M$ volume elements, leading to complex sparse systems from $320k$ to $1.5M$ unknowns. The sparsity pattern related to the matrix for case 1 of Pancake series is shown in Figure 57. Isovalues plot representing the eddy current distribution induced in the susceptor is shown in Figure 58.

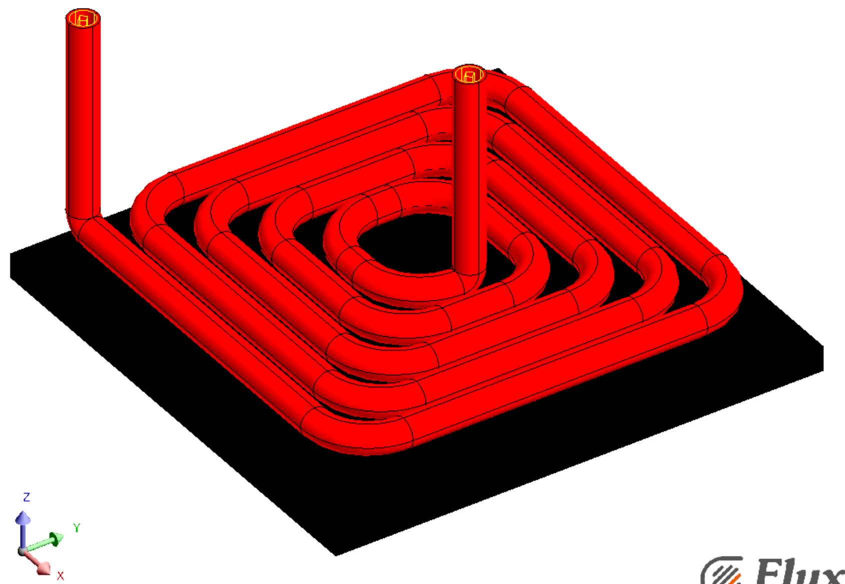


Figure 55. *Geometry of the pancake coil and graphite susceptor.*

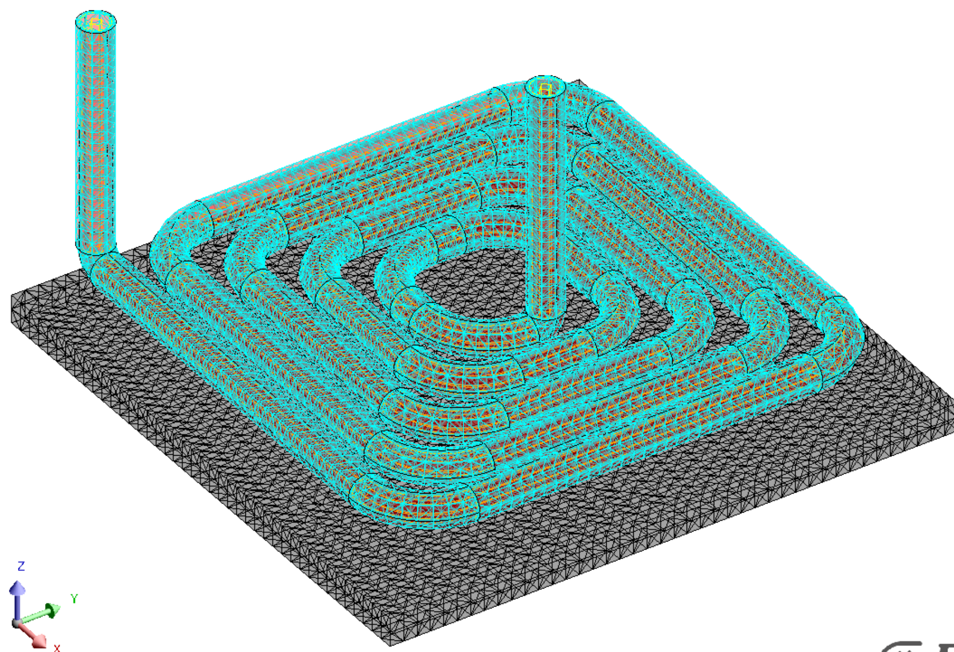


Figure 56. *Mesh of the pancake coil and graphite susceptor.*

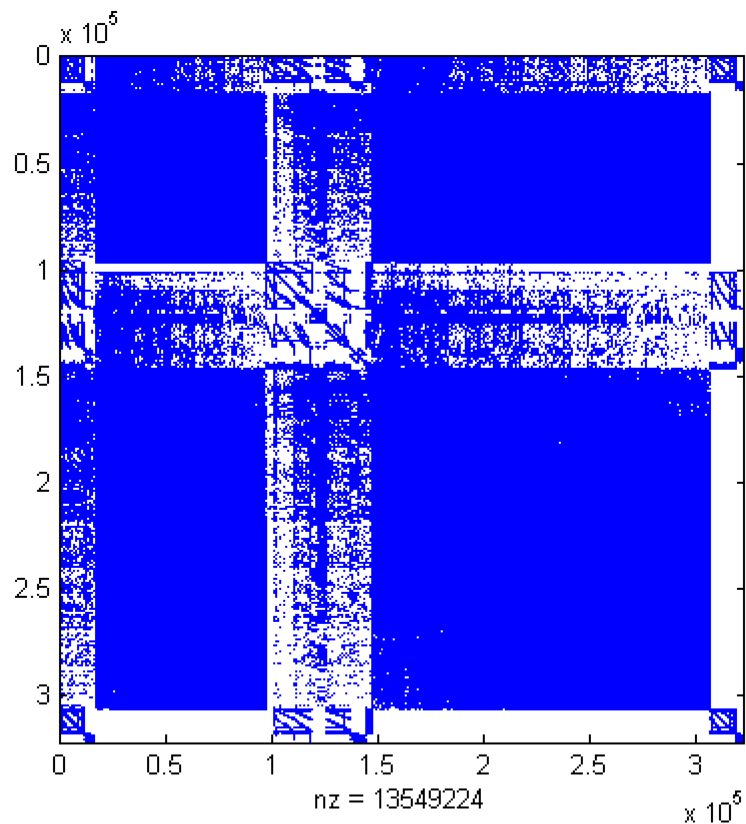


Figure 57. *Matrix originated from Pancake 1 benchmark.*

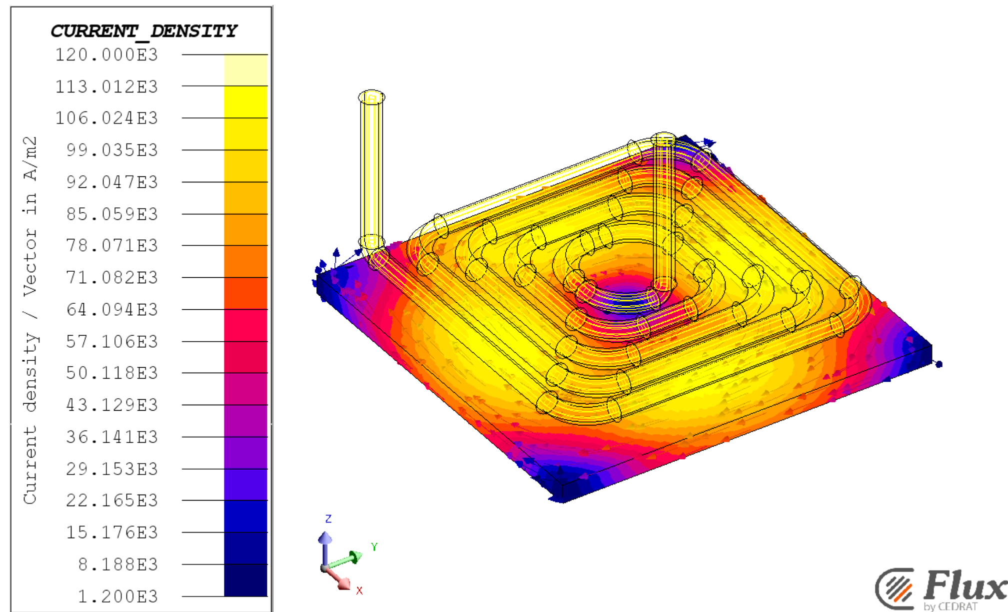


Figure 58. *Eddy current distribution in the graphite susceptor.*

Second benckmark: induction hardening of a gear

Simulations of induction hardening process is one of the most challenging tasks for multi-physics simulation. A complete process description requires not only to compute an electromagnetic and thermal coupled problem but also to couple these results with other different physics, able to calculate the metallurgical transformation and residual mechanical stresses [106]. In order to obtain reliable results from such a combined simulations, the solution of electromagnetic and thermal coupled problems must be as accurate as possible. Then, a very fine, possibly mapped mesh should be built mostly in the heat affected region in order to compute eddy currents distribution properly. In general, the model should consider nonlinear material properties of steel, and consequently the solution should be computed iteratively, through a non-linear solution method. Furthermore, to properly describe material properties dependency on temperature, the time dependent thermal problem must be solved for several time steps. Therefore, all these requirements lead to very long computation time.

For the purpose of this work, only the 3D steady state electromagnetic problem is solved, as it is usually the most time expensive step. Linear magnetic properties are considered. The benchmark model includes a slice of the whole system, as represented in Figure 59. The eddy current problem is solved by means of a nodal $T - T_o - \phi$ formulation. The model mesh, reported in

Figure 60, contains $500k$ volume elements. With this discretization, the dimension of linear system is $370k$ or $2.5M$ by using first and second order elements, respectively. The sparsity patterns related to the matrices from cases 1 and 2 of Gear series are shown in Figures 61 and 62, respectively. Isovalues plot representing the eddy current distribution induced in the gear is shown in Figure 63.

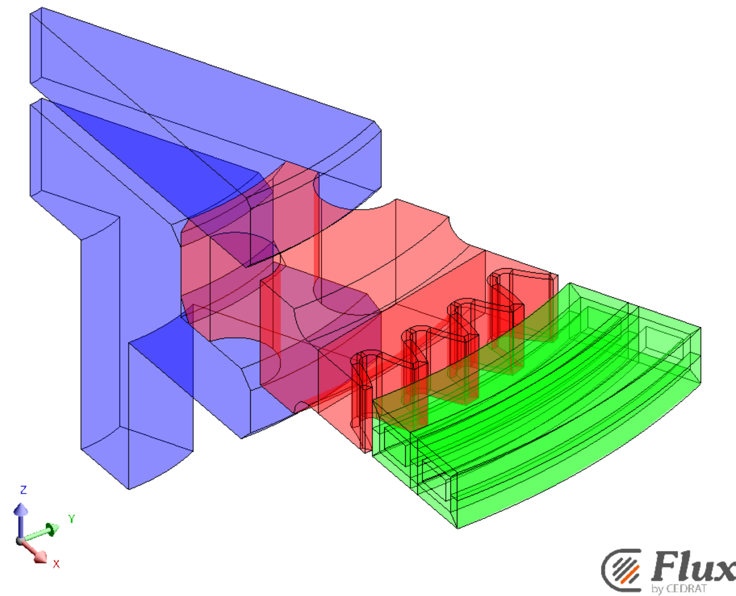


Figure 59. *The figure represents a slice of the whole inductor workpiece geometry because of symmetries.*

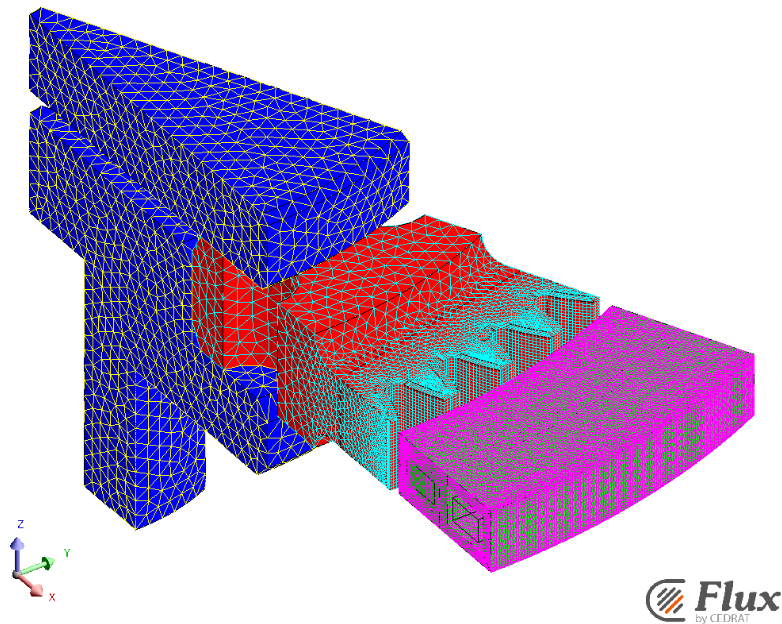


Figure 60. *Mesh of inductor, workpiece and clampers.*

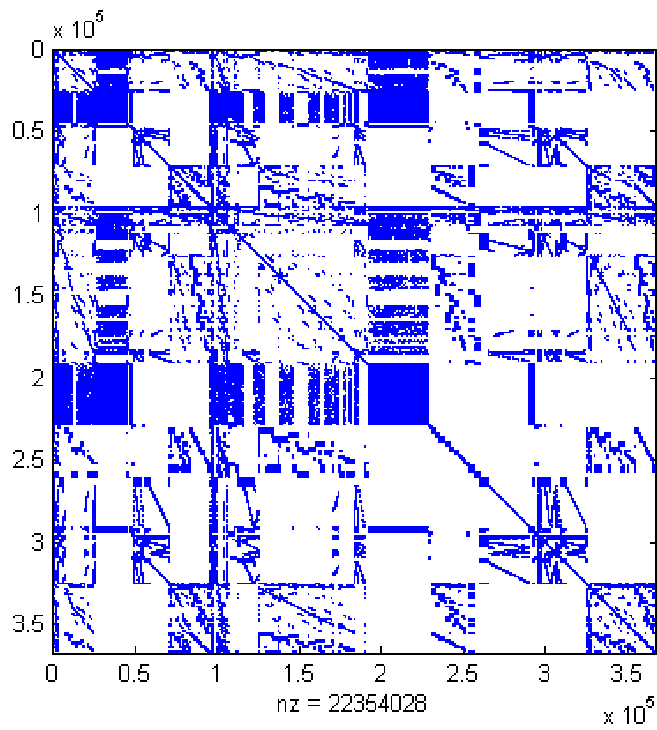


Figure 61. *Matrix originated from Gear 1 benchmark.*

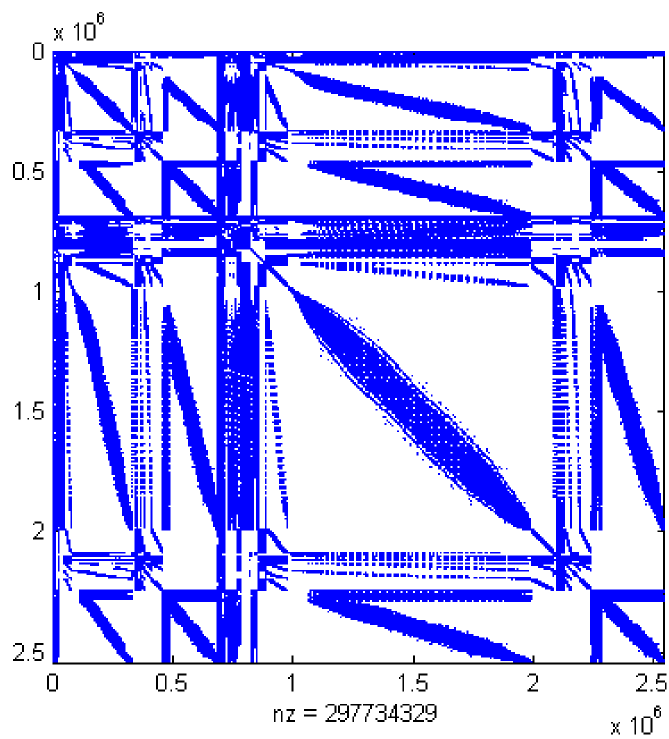


Figure 62. *Matrix originated from Gear 2 benchmark.*

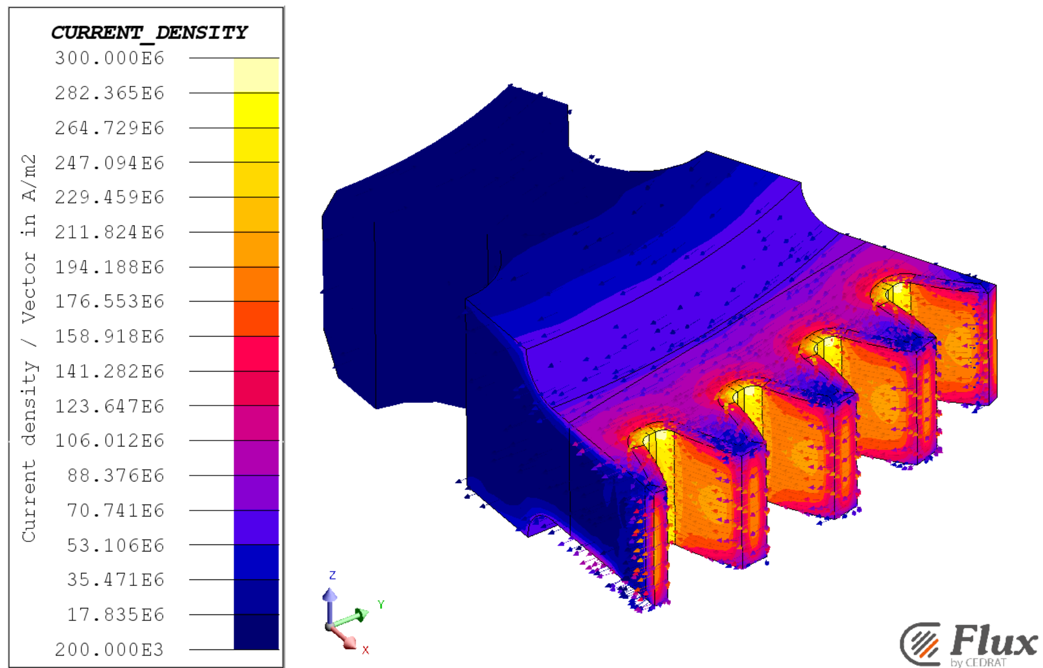


Figure 63. *Eddy current distribution in the gear.*

7.2 Experimental results

The hardware platform for the evaluation of computing performance is equipped with two Intel Xeon (X5670, six cores) and 96 Gb RAM. MUMPS version 4.10.0 and Intel MKL PARDISO version 10.3.9 are compared, both in the In-Core (IC, factors are stored on memory) and the Out-Of-Core mode (OOC, factors are stored on hard disks). The effect of swapping temporary data on virtual memory is also evaluated.

Although MUMPS is based on a message passing model for parallel computations, in this context a single MPI process with multithreaded BLAS libraries is run, because a better performance with such a configuration has been observed on multicore architectures in large 3D problems solution. Both MUMPS and PARDISO are executed on the set of matrices in Table 7, by using 12 OpenMP threads. The corresponding factorization times are reported in Figure 64 and Table 8. The IC factorization times are slightly different with MUMPS and PARDISO, leading to very close behavior when run on 12 cores. Memory usage for IC executions is also very similar among the two solvers. In the OOC mode, MUMPS is slightly slower than PARDISO on small matrices, but uses less memory since PARDISO does not perform I/O when memory is found to be sufficient, as for Pancake 3.

Table 7. Details about matrices used for performance tests.

Matrix	Symmetry	Arithmetic	Mesh elements	N	Number of LU factors
Pancake 1	Unsymmetric	complex	600k	320k	990M
Pancake 2	Unsymmetric	complex	1.1M	630k	2.1G
Pancake 3	Unsymmetric	complex	1.9M	1M	5.2G
Pancake 4	Unsymmetric	complex	2.9M	1.5M	8.7G
Gear 1	Unsymmetric	complex	500k	370k	700M
Gear 2	Unsymmetric	complex	500k (2nd order)	2.5M	12.5G

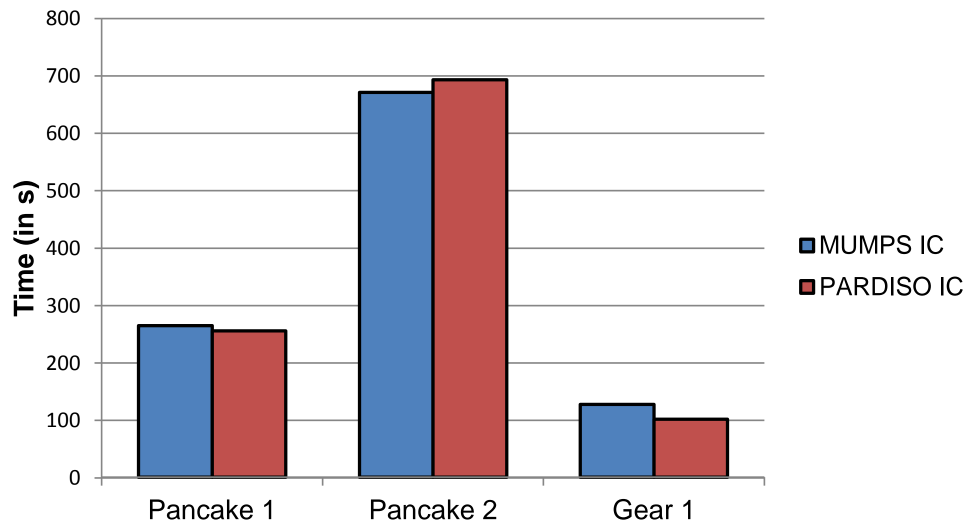


Figure 64. *In-Core factorization time (12 cores, small cases)*

Table 8. Memory in GB, needed for the factorization of the largest matrices.

Matrix	Number of LU factors	MUMPS OOC	PARDISO OOC	MUMPS IC + swap	PARDISO IC + swap
Pancake 3	5.2G	21	87	93	86
Pancake 4	8.7G	30	Error	155	147
Gear 2	12.5G	35	Error	218	214

The missing values for PARDISO in OOC mode (Table 8 and Figure 65) are due to an error during the factorization step in OOC mode with multiple threads. Timings on largest problems show that MUMPS is less penalized by memory swapping than PARDISO.

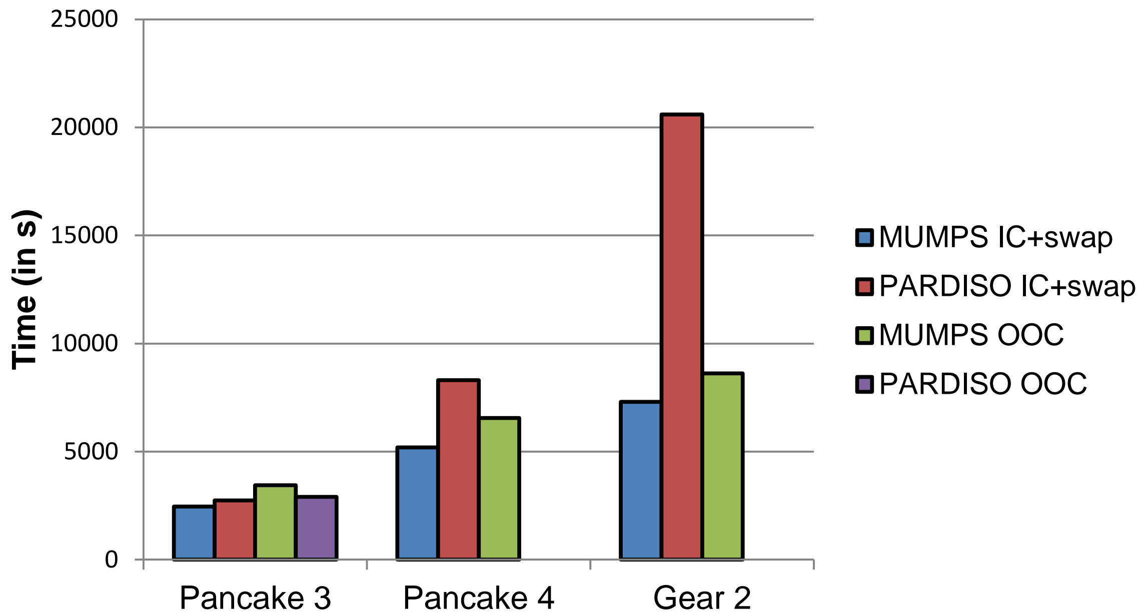


Figure 65. *IC (with swap) and OOC factorization times for large cases.*

Experimental results show that performance of both PARDISO and MUMPS are similar. However, MUMPS was initially proposed for parallel computation on distributed memory systems, thus the performance of a hybrid (distributed-shared) implementation of MUMPS is very encouraging.

Several tests have been carried out in order to identify the key factors that affect the performance of MUMPS, in terms of factorization time when a solution is computed on a shared memory system through a total number of 12 threads. Results in Table 9 are obtained on the previously described testing environment, for different combinations of number of MPI processes and number of threads for each MPI process.

Table 9. *Factorization times on a 12-core multicore system, with different core-process configurations. Times are in seconds. For each matrix, the best time obtained appears in bold.*

Matrix	Sequential	Threaded BLAS only				Pure MPI
	1 MPI x 1 thread	1 MPI x 12 threads	2 MPI x 6 threads	3 MPI x 4 threads	6 MPI x 2 threads	12 MPI x 1 thread
Pancake 1	2097	260	**238**	253	283	378
Pancake 2	5737	**678**	815	849	774	894

Improvements of shared-memory parallelism

In the context of a cooperation between MUMPS team and Department of Industrial Engineering – University of Padova, the MUMPS team provided some new functionalities of the code, and a preliminary version of such functionalities have been tested on a shared memory system. In particular, executions of the code involved only 1 MPI process, by exploiting:

- node parallelism through BLAS libraries for factorization kernels
- tree parallelism through a fork-join model of parallelism, based on OpenMP programming

The development of such functionalities and all tests have been performed by MUMPS team [96,97]. A detailed description is given in chapter 6. Benchmark matrices, originated from induction heating finite element models, have been provided by Department of Industrial Engineering.

As described in the previous chapter, the initial goal was to exploit tree parallelism in shared memory environments, by adopting algorithms commonly used in distributed memory environments and by rearranging them to a multithreading model. The AlgL0 algorithm consists in finding a separating layer in the tree, called L0, such that node parallelism will still be applied above it, but tree parallelism will be applied under it through OpenMP.

Furthermore, the use of adequate memory mapping policies allows for performance improvements of MUMPS on SMP (Symmetric Multi Processor) and NUMA (Non-Uniform Memory Access) architectures [104]. The "localalloc" policy, which consists in mapping the memory pages on the local memory of the processor that first touches them, is applied on data structures used under L_0, in order to achieve a better data locality and cache exploitation. The interleave policy, which consists in allocating the memory pages on all memory banks in a round-robin fashion such that the allocated memory is spread over all the physical memory, has been used on data structures above L_0 in order to improve the bandwidth.

Both MUMPS 4.10.0 and MUMPS 4.10.0 with new AlgL0 algorithm have been tested on the set of test matrices by MUMPS team, with the stated memory allocation techniques. As reported in Table 10, this approach brings a remarkable reduction of computational time on all tested matrices. The benefit tends to decrease on large matrices, because the fraction of workload in the top of the tree (above L0) gradually increases in comparison to the workload in the bottom of the tree. Furthermore, this approach should benefit from the use of the interleave policy.

Table 10. *Time saving by using MUMPS 4.10.0 with ALGL0 algorithm.*

Matrix	Time saving
Pancake 1	13 %
Pancake 2	11 %
Pancake 3	8 %
Gear 1	21 %

In conclusion, significant improvements of shared-memory parallelism in MUMPS have been verified also in 3D numerical simulation of induction heating industrial applications. Results show a remarkable reduction in computing time, when both tree and node parallelism are exploited.

Improvements by Low-Rank approximation techniques

A method for computing the blocking of frontal matrices for the BLR format has been developed by MUMPS team. A detailed description is given in chapter 6. The method has been effectively tested on practical benchmarks arising from electromagnetic, mechanical and thermal problems in [82,83]. This technique does not require any knowledge of geometry and physics of the problem and can be run at a minimal cost, if compared to the cost of the analysis phase of a multifrontal solver.

In the context of a cooperation between MUMPS team and Department of Industrial Engineering – University of Padova, the MUMPS team provided an experimental version of MUMPS with such new functionality. Several experimental tests have been conducted by MUMPS team on matrices originated from induction heating finite element models. Results show that considerable gains can already be achieved at very accurate approximation levels, in which case the final solution backward error is comparable to the one obtained with a standard, full-rank solver. Experimental results also show that the BLR format is tolerant with respect to variations in the size of the blocks. This property may be used to accommodate the BLR format of frontal matrices to data distribution in a parallel environment as well as to classical pivoting techniques[82,83]. The BLR multifrontal solver has been tested on matrices in Table 7, arising from induction heating applications.

Table 11. *Improvements by low-rank approximation techniques.*

Matrix	LR threshold ε	Memory saved for factors storage	Operations saved for factorization	Scaled residual
Pancake 2	10^{-8}	35%	60%	2.3×10^{-16}
Pancake 4	10^{-10}	34%	60%	1.0×10^{-12}
	10^{-14}	14%	29%	3.5×10^{-16}

Experimental results in Table 11 show that the method is more efficient on the largest problems, a good property in the context of large scale computing (in case of Pancake 2, no compression was achieved with $= 10^{-10}$). This behavior confirms the experimental tests performed on MUMPS team benchmarks coming from other applications. For Pancake 2, significant gains are obtained by setting the low-rank threshold to $\varepsilon = 10^{-8}$: necessary memory is reduced up to 35% and the numerical complexity is almost halved. In this case, a few steps of iterative

refinement are performed to recover full precision from the original approximated precision of 10^{-10}. For Pancake 4, solved by setting a low threshold precision of 10^{-14}, a good compression is obtained naturally without any loss of accuracy.

Finally, it should be noted that, by choosing a more aggressive threshold ε, the BLR format can be used to efficiently produce loosely approximated factors that could be used as effective preconditioners for iterative solvers [104].

8. Design and optimization of induction heating applications

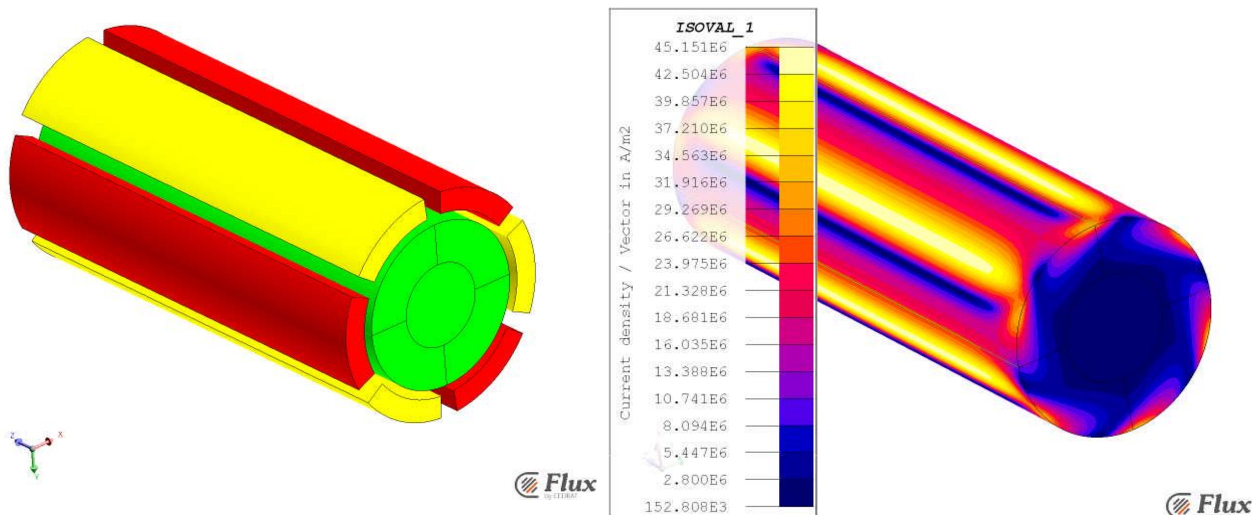
8.1 Design of a Permanent Magnet Heater

The mass heating of billets before hot metal forming plays a major role among industrial induction heating applications as regarding the number of installations, unit rated power of heaters and energy consumption. The efficiency of the classical induction process, i.e. the ratio between power transferred to the workpiece and the power supplied to the inductor, is in the range of 50-60% for aluminum or copper billets. For these materials, a DC induction heating concept has been proposed to improve the process efficiency. In this approach the billet is forced to rotate inside a transverse DC magnetic field by means of an electric motor. Due to the magnetic flux variation an induced current distribution reacts to the driving torque during the rotation and generates thermal power within the billet.

Recently, Laboratory for Electroheat Processes (LEP) proposed an innovative approach based upon a rotating system of permanent magnets [112]. This solution appears to be very promising as it allows for a high efficiency without using expensive superconductive systems to generate the magnetic field.

It should be pointed that a virtual prototyping methodology has been effectively used to support the design phase of the project. Initially, several versions of the machine have been proposed, thus a large number of simulations have been performed with the aid of advanced finite element tools [39] in order to evaluate benefits and drawbacks for each solution, in terms of feasibility, costs and effectiveness.

Parallel computing capabilities played an important role. In fact, by exploiting MUMPS and PARDISO parallel solvers, solution time could be remarkably reduced from some days to a few hours, thus opening the way to the optimization process. Then, in the context of an optimization process, a numerical model of the heater has been built (Figure 66(a)), and a set of magneto-thermal simulations have been carried out (Figure 66(b)) in order to identify the correct design parameters, i.e. geometrical dimensions, position, number and material of magnets, dimensions of stator core and rated power of the induction motor. The optimization goals are cost reduction, the best exploitation of magnetic materials, and the best performance of the machine.

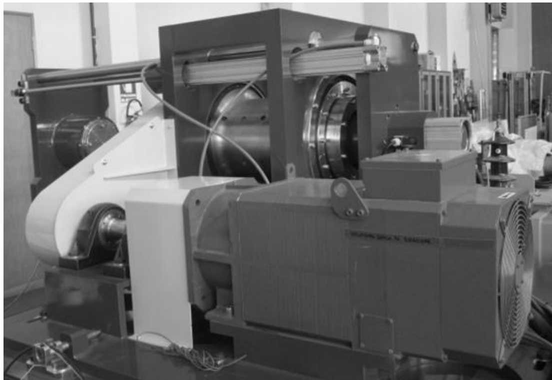


(a) Geometry of the PMH model. Steel cores, supporting and rotating with magnets, are not reported for sake of clearness.

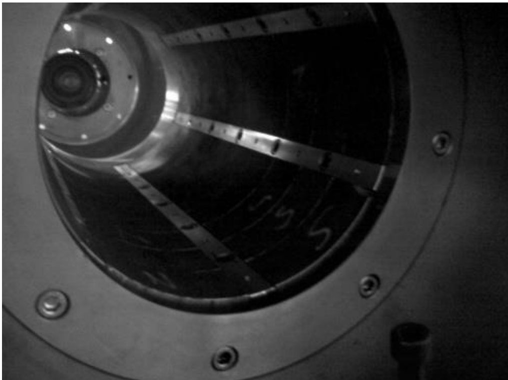
(b) Eddy current distribution induced in the billet during the rotation of magnets.

Figure 66. *Finite element model of PMH, used for performance evaluation and optimizations before the construction of the prototype. The nonlinear 3D transient electromagnetic problem is coupled to a thermal and a mechanical problems. The solution of each time step requires ten minutes on parallel hardware.*

Once the industrial scale prototype has been realized (Figure 67), electrical and thermal experimental measurements have been performed in order to verify the simulation results. A 200 mm diameter, 500 mm long aluminum billet (42 kg) can be heated uniformly with unmatched heating efficiencies, process integration and product quality. The motor drive has a rated power of 52 kW at 2500 rpm, and the magnetic field is produced by SmCo rare earth permanent magnets.



(a) The prototype installed in the Laboratory of Electro heat of Padua University.



(b) Arrangement of the permanent magnets inside the steel rotor.

Figure 67. *Industrial scale laboratory prototype.*

In Figure 68, a layout of the measurement system is shown. Electrical measurements have been performed by means of a three phase power analyzer (Chauvin-Arnoux, mod. C.A. 8335B

Qualistar) and four current probes (Chauvin-Arnoux, mod. MN93BK). Thermal measurements at the end of the heating process have been performed by means of an infrared thermo camera (AVIO, mod. TVS2000). The billet is blackened with a high temperature resistant paint to control the emissivity coefficient. Thermocouples have been used to monitor the billet surface temperature after the heating process. In particular, K-type thermocouples have been connected to a data logger unit.

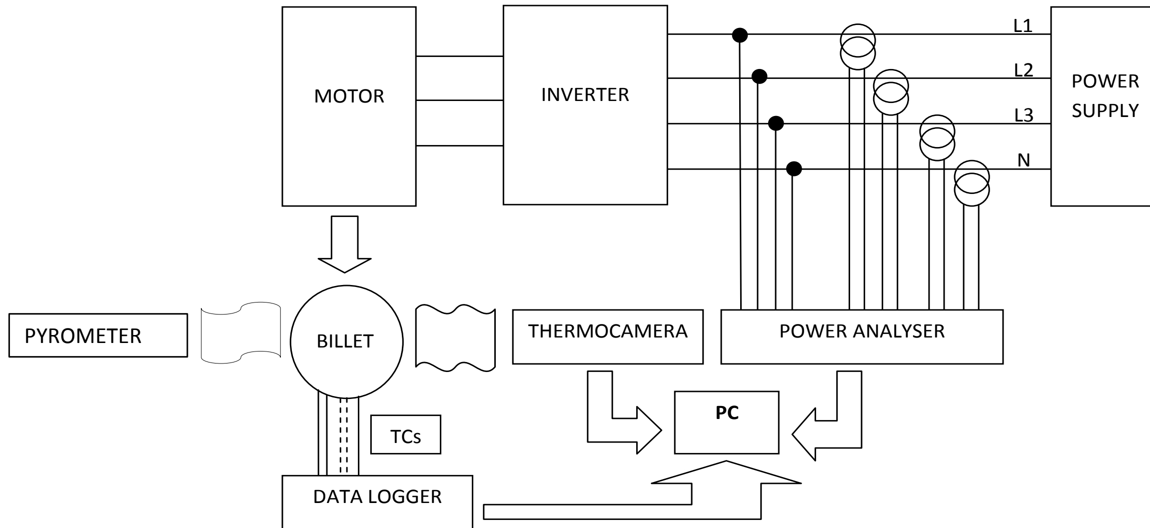


Figure 68. *Layout of thermal and electrical measuring system.*

Experimental results are provided in Figure 69, for different rotating speed, ranging from 500-750-900, up to 1000 rpm, at which the maximum induced power of 57 kW is transferred to the billet (the induction motor can be overloaded for testing purpose).

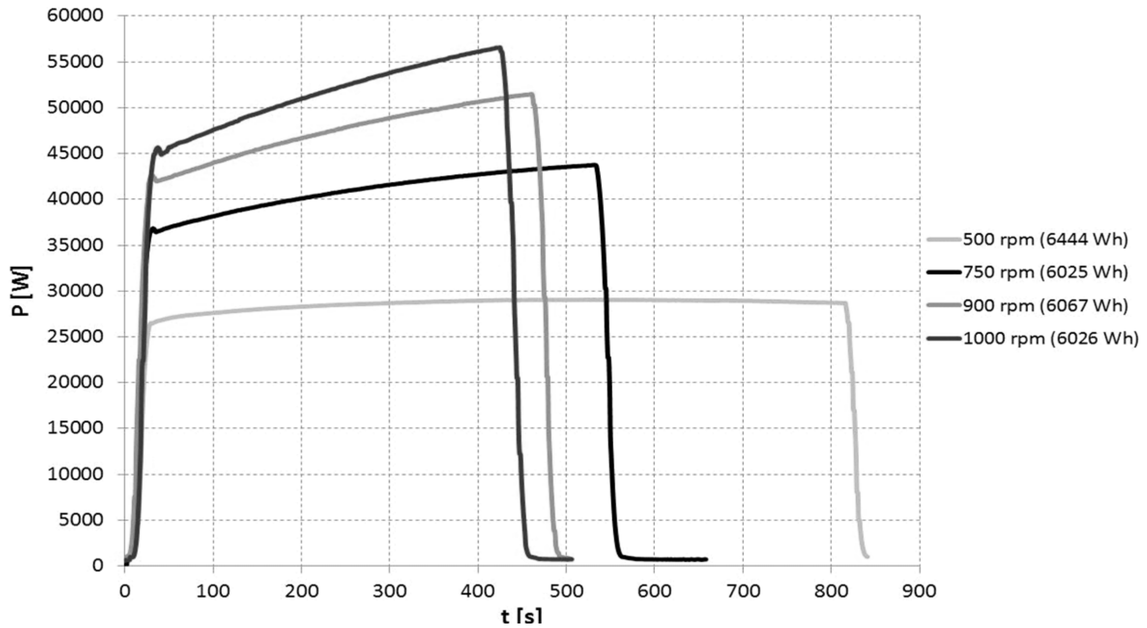


Figure 69. *Absorbed Active Power for different rotating speed. In each test, roughly the same energy is provided to the billet. Note that additional losses are also included, even if they account for a small contribute.*

As shown in Figure 70, a comparison shows good agreement between experimental and simulation results. An almost constant difference between computed and measured power is related to additional power losses affecting the prototype, i.e. losses due to inverter, induction motor and transmission, that are not taken into account in simulations.

Figure 69 shows measurements of power absorption from the supply network, for different rotor velocities. The corresponding temperature transients, reported in Figure 71, have been measured for insulated and not insulated clampers, holding the billet. In fact, clampers (made of steel) are not preheated at the beginning of each test (as they would be in a continuous operation of the heater). In a first configuration, clampers are in contact with the billet during the heating phase and heat is transferred from the billet to the clampers by thermal conduction, leading to a non-realistic estimation of additional losses. Thus, in a second configuration, clampers are thermally insulated by a thin layer of fiberglass. The second configuration, that enables for higher temperature and efficiencies, gives a correct estimation of the behavior of the machine in a continuous operation.

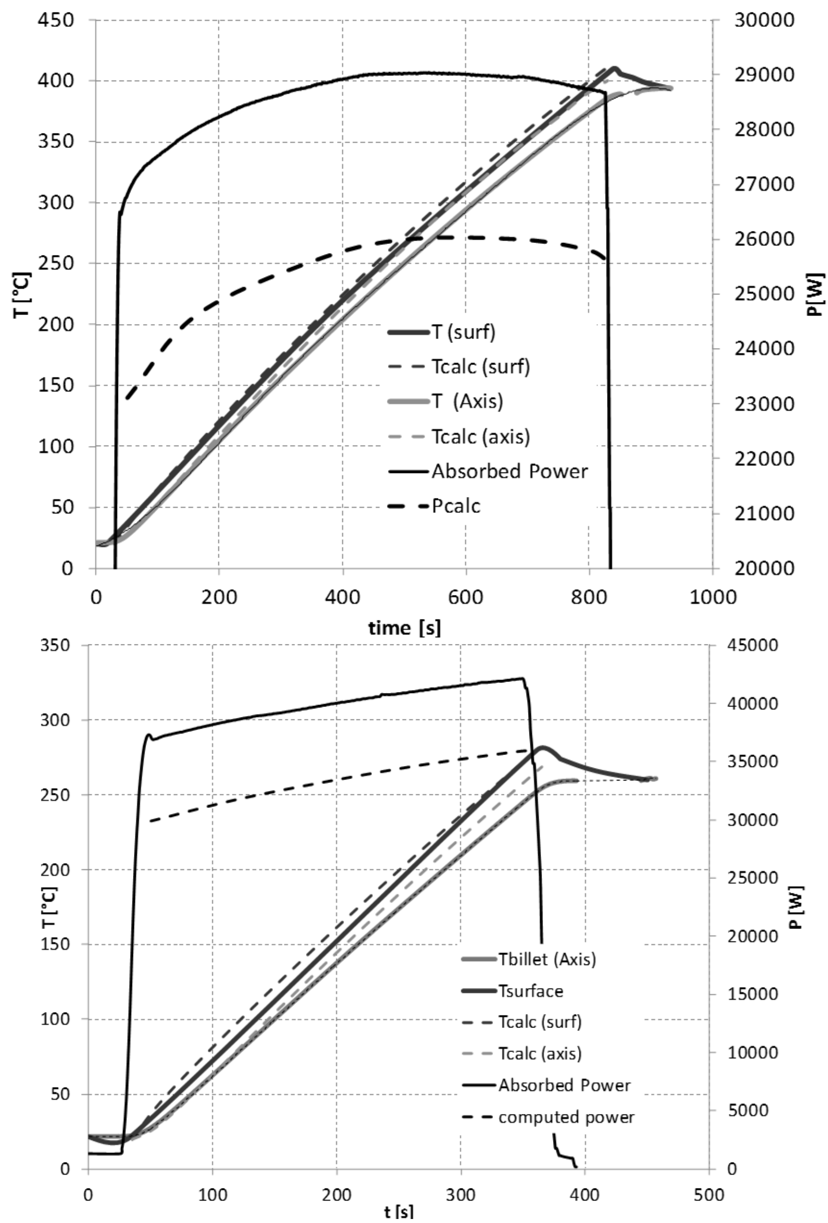


Figure 70. *Experimental and numerical results for two different rotating speeds: 500 rpm (upper) and 750 rpm (bottom). Measured values are represented with continuous lines. Computed quantities are represented with dashed lines.*

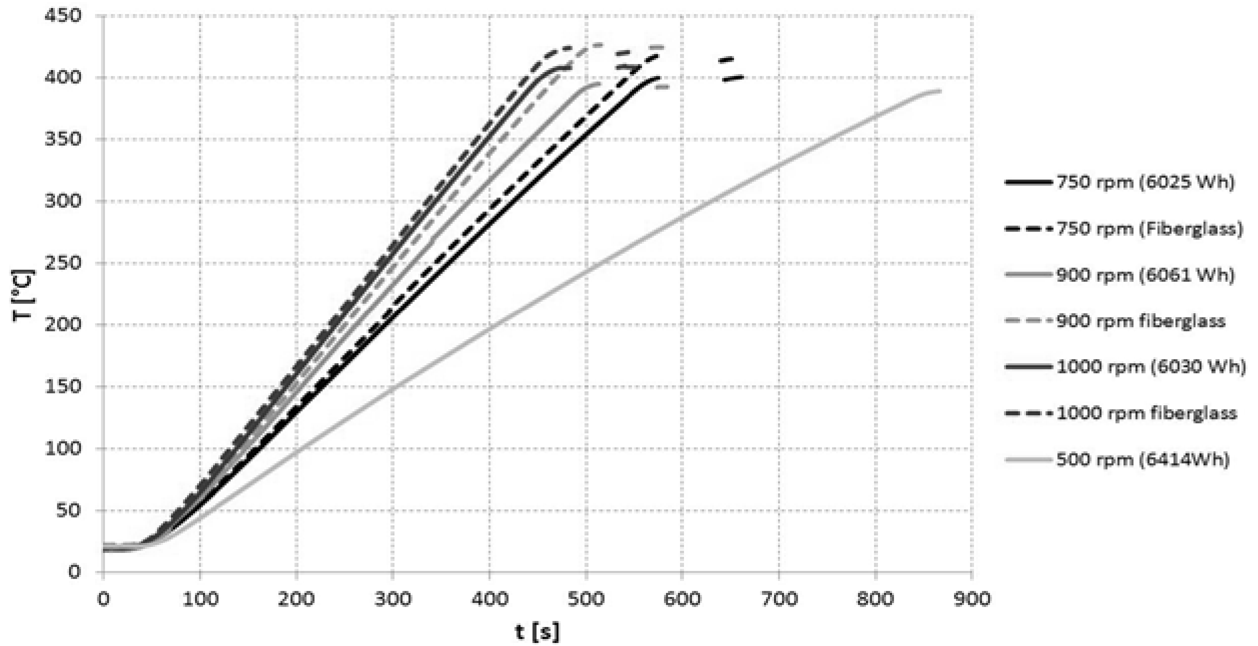


Figure 71. *Temperature evolution inside the billet, when clampers are not insulated, are indicated by continuous lines. Temperature evolution inside the billet, when clampers are insulated by a fiberglass layer, are indicated by dashed lines.*

The amount of thermal energy transferred to the billet can be estimated from:

$$E_{transf} = \frac{cM(T_{avg} - T_{room})}{3.6 \times 10^6}$$

where E_{transf} is expressed in kWh, $T_{avg}$is the average temperature after a thermal equalization phase (45 seconds), T_{room} is the ambient temperature, $c = 900\ J/kg\ K$ is aluminum specific heat and $M = 42.5\ kg$ is the mass of the billet.

Then, global efficiency of the prototype is computed by an energy ratio, according to:

$$\eta_{global} = \frac{E_{transf}}{E_{electric}} \cdot 100$$

where the absorbed electrical energy $E_{electric}$ is measured accordingly to the scheme presented in Figure 68.

An estimation of efficiency is reported in Figure 72, with and without the fiberglass thermal insulating layer between the clampers and the billet. An higher efficiency is expected to be achieved when the heater is operated continuously. In this condition, the clamping system reaches its regime temperature and the heat transferred by thermal conduction from the billet to clampers is lowered. Therefore, reported global efficiency is a conservative estimate.

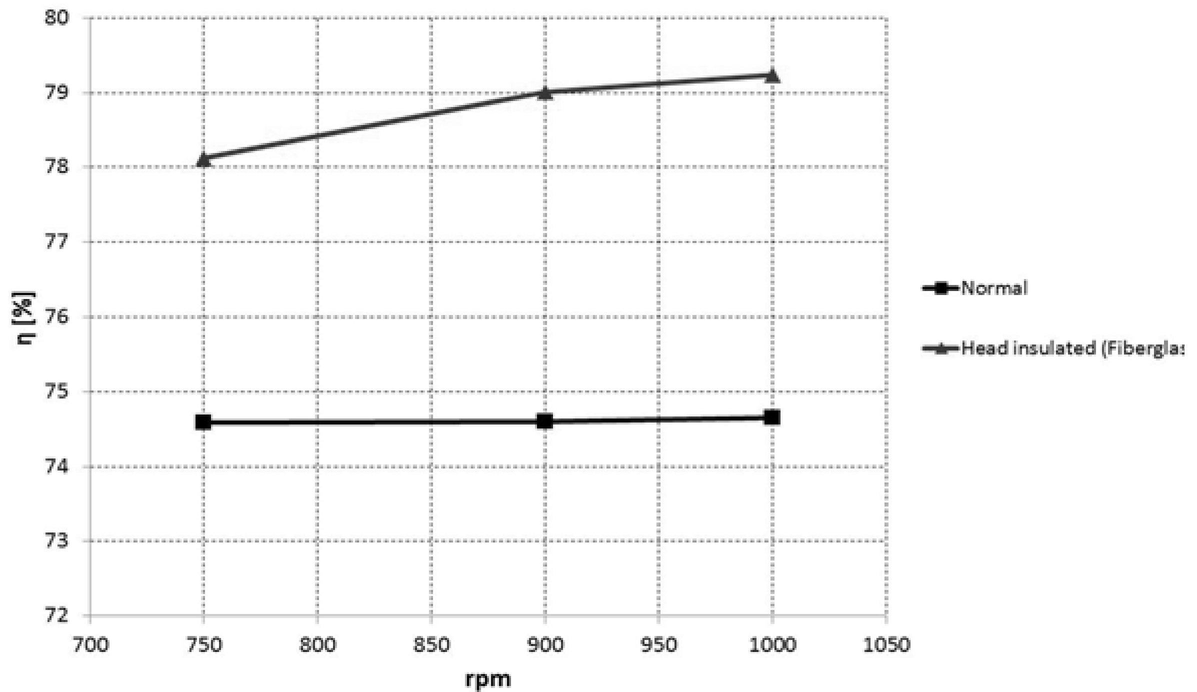
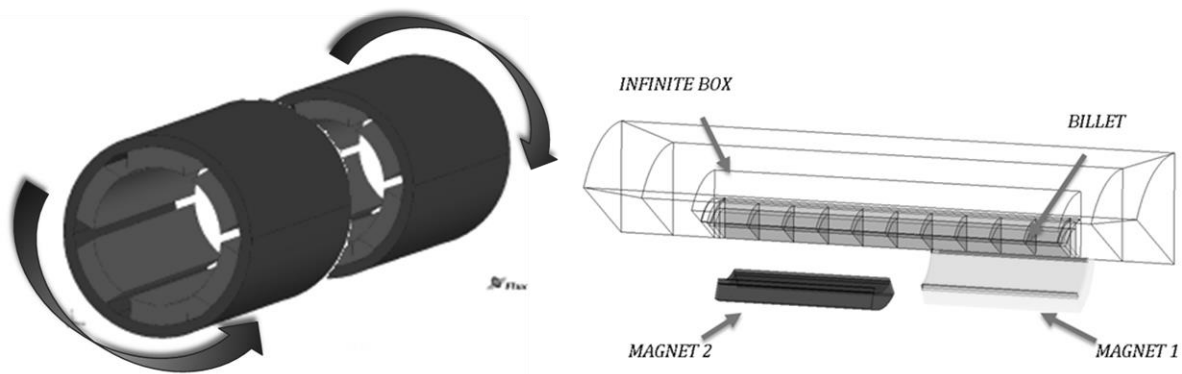


Figure 72. Global system efficiency for different rotational speed, both with and without thermal insulation towards clampers.

In order to improve the performances of PMH system and to obtain a tapered temperature profile at end of the heating process, several approaches have been evaluated. Among all, a 'multiple rotors' solution, in which different rotors can rotate independently at different speeds, can lead to a controlled non-uniform temperature profile along the billet.

A feasibility study has been carried out with a dual rotor solution. In the proposed solution the rotors can turn in opposite direction, as reported in Figure 73(a). Opposite directions of rotation allow for a remarkable reduction of the total electromagnetic torque acting on the billet, up to a cancelation of the net torque transferred to the clampers when the rotational velocity is the same.

The studies are carried out by means of 3D transient finite element analysis. In order to reduce the model complexity and computational time, only one magnetic pole of the system is considered in the model, because of periodicity conditions. The geometry of finite element model is reported in Figure 73(b). Simulations are performed by a commercial code [39], in which rotational movement of regions and periodicities are managed by specific algorithms. Since the model considers only a slice of the full system, of an angular width $\alpha = 360/p$ depending on the number of magnetic poles p, this kind of model allows for an easy parameterization on the number of magnetic poles.



(a) *The concept of a dual rotor system, with opposite rotational velocity.*

(b) *Simulated model is reduced to only a α_{mod}-wide slice of the entire system, due to symmetries.*

Figure 73. *Model of a dual rotor PMH.*

In the model, the volumetric region of the aluminum billet is divided into ten small volumes, in order to compute the integral of the induced power density in each single volume. The integral values of the power induced in each volume give a rough but quick representation of the non-uniformity in induced power density distribution along the billet, and it can easily exploited in order to decouple thermal and electromagnetic problems, for faster evaluation of performance. An example of real induced current distribution is presented in Figure 74, a $p = 6$ system where both the rotors turn at 900 rpm, with opposite directions.

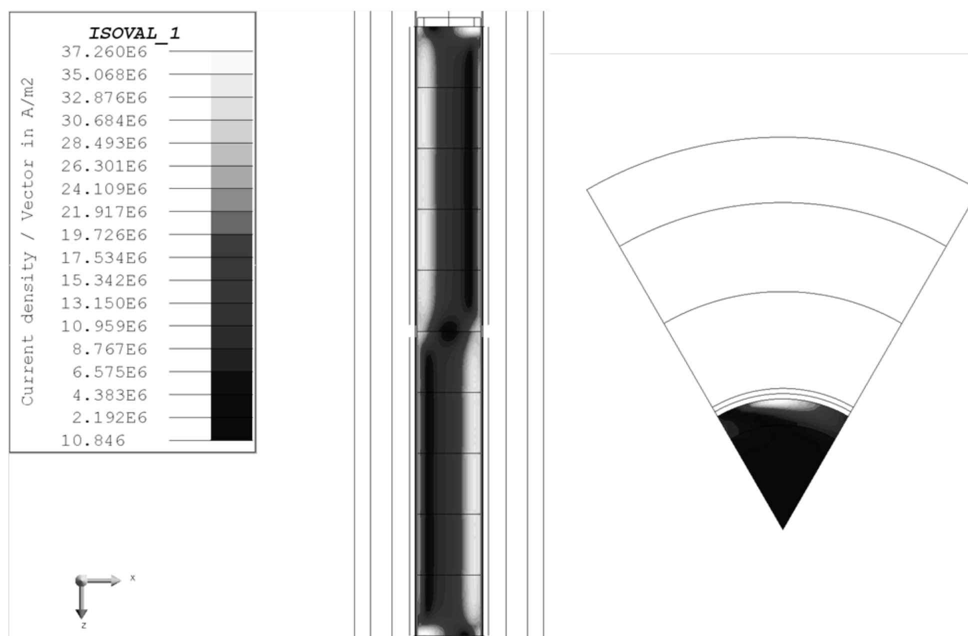


Figure 74. *On the left, 3D simulation results in terms of eddy current distribution induced in the billet by a $p = 6$ rotating system. On the right, the eddy current distribution on a planar cross section is shown.*

The power density distribution, represented by a discretized distribution, is shown in Figure 75, for different rotor speeds and different number of magnetic poles.

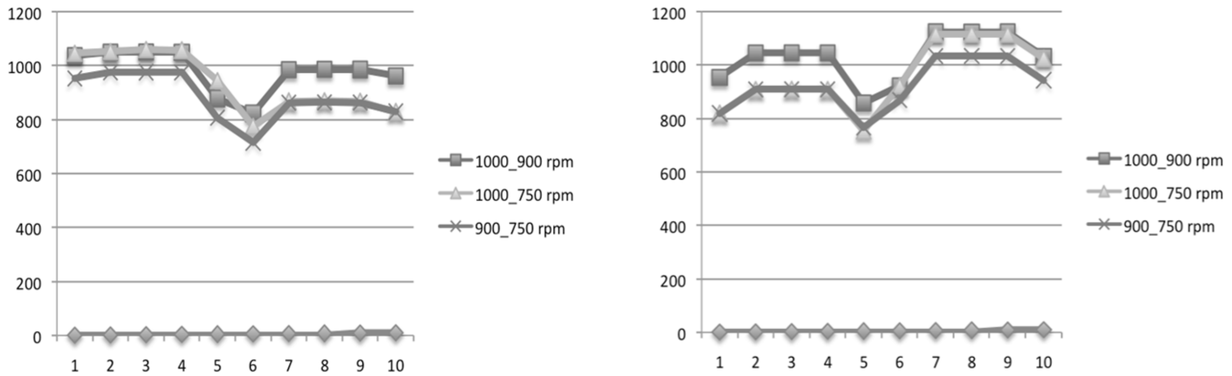


Figure 75. *Induced power for different rotational speeds of the rotors in the ten subdivisions. On the left, a* $p = 6$ *system is considered. On the right a* $p = 8$ *is considered.*

For practical reasons concerning the successive extrusion process, a temperature difference of about 60-80 °C between the front and rear sides of the billet is requested. Numerical results, obtained by the solution of the thermal problem by setting the power density distribution (previously calculated for different rotor speeds) as heating source, are summarized in Table 12. Maximum and minimum temperature in the billet are reported for three different combinations of rotor speeds, both at end of the heating phase and after 30 seconds of thermal equalization.

Table 12. *Summary of calculated temperature in a dual rotor system, with different rotational speeds of the two rotors [case study: 6 magnetic poles].*

Rotational speed [rpm]	**Process time**	**Temperature at the end of the process**		**Temperature after 30 seconds of thermal equalization**	
ROTOR1_ROTOR2	[s]	T_{MIN} [°C]	T_{MAX} [°C]	T_{MIN} [°C]	T_{MAX} [°C]
1000_900	440	429	482	437	472
1000_750	470	403	504	409	494
900_750	500	426	497	432	487

In this cases, the net electromagnetic torque acting on the clampers is not fully negligible when different rotors are set to different speed values. However, the resulting net torque is significantly reduced, leading to a simplification in the design of the clampers.

In conclusion, the effectiveness of the “multiple rotors” approach is confirmed by numerical simulation results and it will exploited in the second generation prototype. By varying the speed

of rotors independently, different distributions of the power density can be induced in the billet, thus conical temperature profiles can be obtained.

8.2 Optimization of temperature profiles in heated parts

In many heating applications, an ideal temperature distribution in the workpiece is usually desired. Induction heating is applied to various thermal processes, because it allows for controlled heat generation inside the workpiece, thus allowing for higher heating efficiency and good temperature control, even in the case of very fast heating treatments. Currently, induction heating of complex shaped workpieces is still a challenge, usually left to coil designer experience. However, in order to understand and to optimize the overall heating process, in particular for surface heat treatments of metals, a fine tuning of process parameters cannot be done empirically and requires advanced simulation tools.

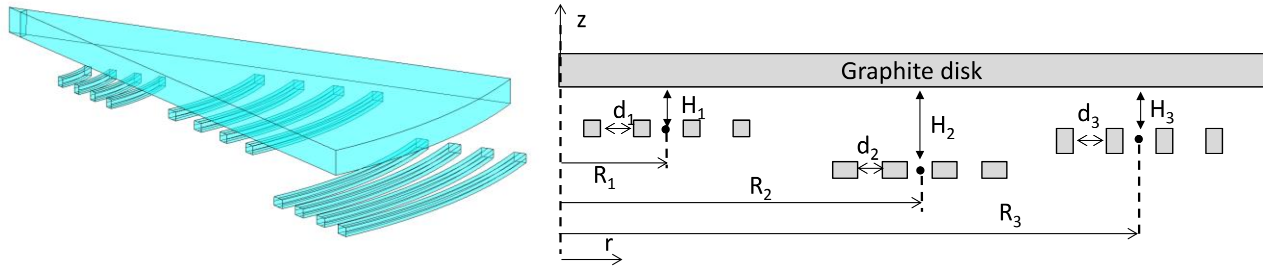
Then, an optimal design for such processes is not limited to the coil design itself, but it implies the solution of coupled electromagnetic and thermal problems by finite element simulations, and the use of modern optimal design procedures to identify advantages and drawbacks of a set of feasible solutions.

In cooperation with University of Pavia, a multiobjective optimal design procedure has been developed and tested on a simple induction heating benchmark. As a reference, a single objective algorithm is applied to the same benchmark in [114], in which the temperature uniformity is the optimization goal. In the new approach, an automated procedure of bi-objective optimization, based on NSGA-II algorithm [115], has been used to solve a multiphysics inverse problem. Both magnetic and thermal aspects are considered in the objective functions. In particular, the temperature uniformity in the workpiece is evaluated by:

- a classical min-max criterion
- a criterion of proximity

Then, a family of improved solutions, found using both criteria, is considered in a comparative way.

From the optimization process point of view, the direct problem consists of a pancake coil heating a graphite disk. The inductor is composed by 3 groups of 4 turns. All the turns are series connected and carry a current of 1000 A at a frequency of 1000 Hz. Figure 76(a) shows $1/12$ of the device, due to symmetries. Though the benchmark problem is axisymmetric, a 3D model is simulated for the sake of generality. In particular, the same procedure is valid for non-symmetric models, i.e. with magnetic flux concentrators. The electromagnetic (EM) problem is solved in time-harmonics conditions, in terms of electric vector potential and magnetic scalar potential formulation. The physical domain includes the inductor, the graphite disk and the surrounding air.



(a) Geometry of the model. Symmetries are taken into account, thus only a slice of the device is modeled.

(b) Geometrical parameters.

Figure 76. *The direct problem consists in a pancake coil heating a graphite disk.*

Eddy current distribution and power losses, both in each turn and in the disk, are computed in order to evaluate the device efficiency. Then, a steady-state thermal problem is solved by assuming the power density distribution in the disk as the heat source. The thermal domain is the disk, and appropriate heat exchange boundary conditions are set on the surface. A typical mesh is composed of $130k$ first order tetrahedral elements.

The design variables, reported in Figure 76(b) are:

- the mean radius R_k of each group of turns
- the radial distance d_k between turns in each group
- the axial distance h_k of each group from the disk.

Two optimization goals have been defined as:

- the maximum electrical efficiency
- the temperature uniformity on the disk.

Note that the second goal only deals with uniformity of temperature, but a target temperature is not specified. In order to achieve a good uniformity around a an average target temperature, another goal could be easily added. Another option could be to transform the described optimization problem in a constrained optimization problem.

The electrical efficiency, η, defined as the ratio of active power transferred to the disk to the one supplied to the inductor, should be maximized. Then, the maximization problem is converted to a minimization problem by:

$$f_1(g) = 1 - \eta(g)$$

where $g = (R_k, d_k, h_k)$, $k = 1,2,3$.

The uniformity function, defined on the temperature profile, should be maximized. The uniformity function can be evaluated based on a criterion that minimizes the difference between the maximum and the minimum of the temperature along a radial path γ, located 1 mm below the upper surface of the disk:

$$f_2(g) = T_{max}(g,\gamma) - T_{min}(g,\gamma)$$

where T_{max} and T_{min} are maximum and minimum temperature along the path γ.

An alternative approach relies on a "criterion of proximity" for the evaluation of temperature uniformity. The criterion is based upon a tolerance band, ΔT^*, around a reference temperature value. An example is shown in Figure 77, where the path γ is sampled by means of N_{max} points. For each point, the corresponding temperature T_i, $i = 1, N_{max}$ is evaluated and compared with the temperature T_j, $j = 1, N_{max}$ for $j \neq i$ (all the other sampling points). A T_j value is considered to satisfy the "criterion of proximity" if the condition $|T_j - T_i| < \Delta T^*/2$ holds. Then, for each T_i value, the number of T_j values that satisfy the "criterion of proximity", is calculated. Finally, the uniformity function is evaluated, for $j \neq i$, as:

$$f_2'(g) = \left[N_{max} - \sup_j N_j \left(|T_j(g,\gamma) - T_i(g,\gamma)| < \frac{\Delta T^*}{2}\right)\right]$$

Actually, the value for index $j = 1, N_{max}$ that minimizes the right-hand side is searched for.

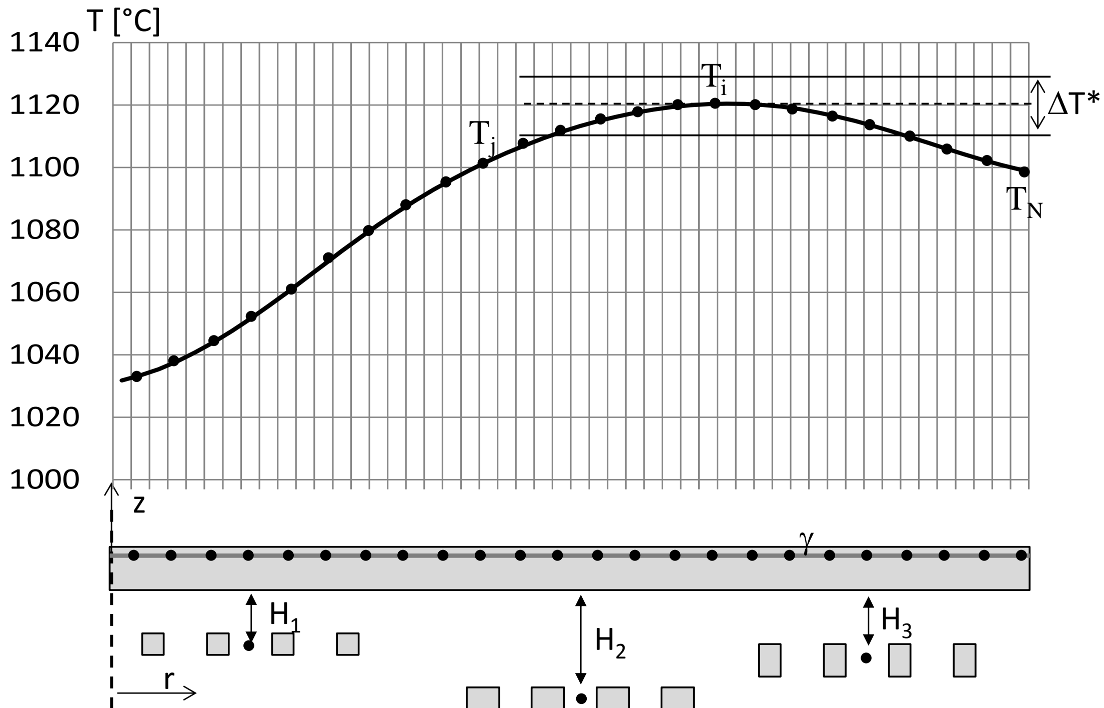


Figure 77. *Application of "criterion of proximity".*

```
count = 0
for i = 1: Nmax
        K = 0
        for j = 1: Nmax
                if |Ti − Tj| < ΔT* and j ≠ i
                        k = k + 1
        if k > count:
                count = k
return Nmax − count
```

Figure 78. *The pseudo-code for the evaluation of the uniformity function is reported.*

On these basis, both $f_1(g)$ and $f_2(g)$ (or $f_2{}'(g)$) objective functions have to be minimized with respect to the design parameters shown in Figure 71(b). The first objective refers to the electromagnetic problem and the second objective refers to the thermal problem, thus a multiphysics and multiobjective inverse problem is defined.

The optimization problem is solved and results obtained by using the two different uniformity criteria have been compared. Temperature is evaluated on $N_{max} = 51$ points, regularly distributed along path γ. The NSGA-II algorithm is set with a population of 20 initial individuals.

As regarding the min-max uniformity criterion, after 16 generations the NSGA-II algorithm lead to the well-defined Pareto front in Figure 79. An example of initial and final temperature profiles for some non-dominated solutions are reported in Figures 80 and 81.

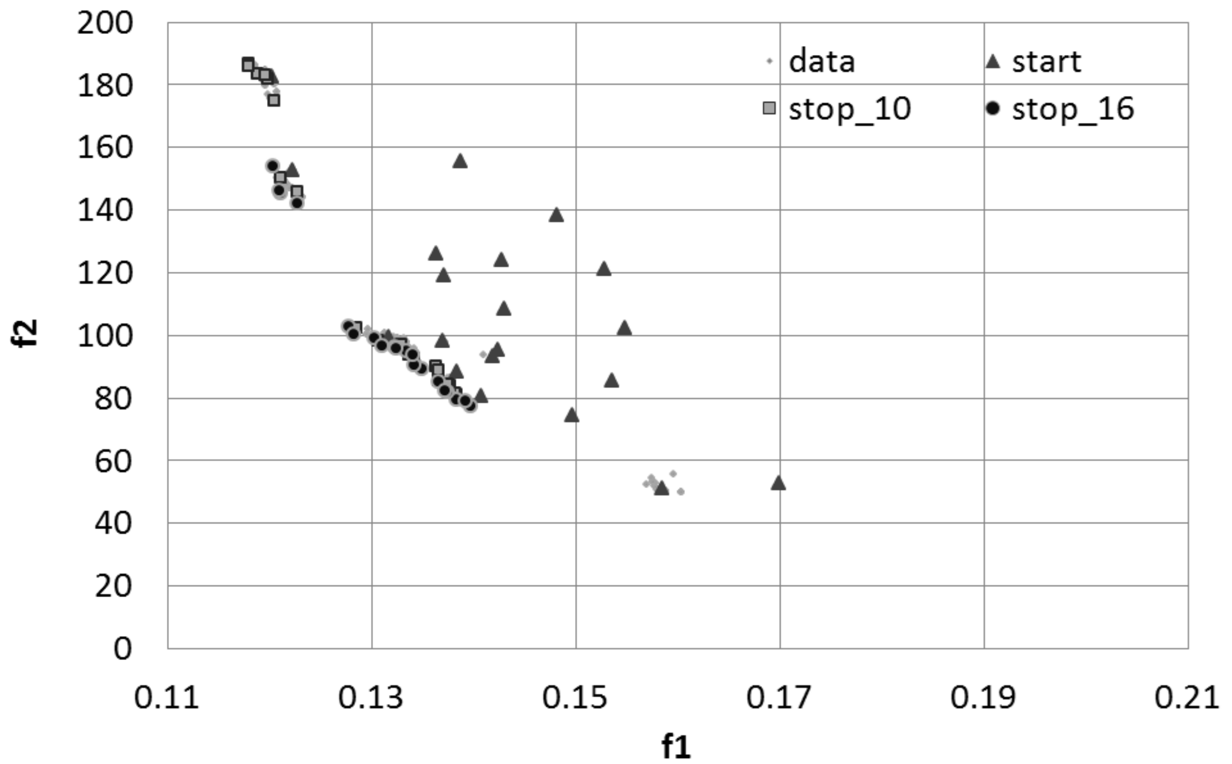
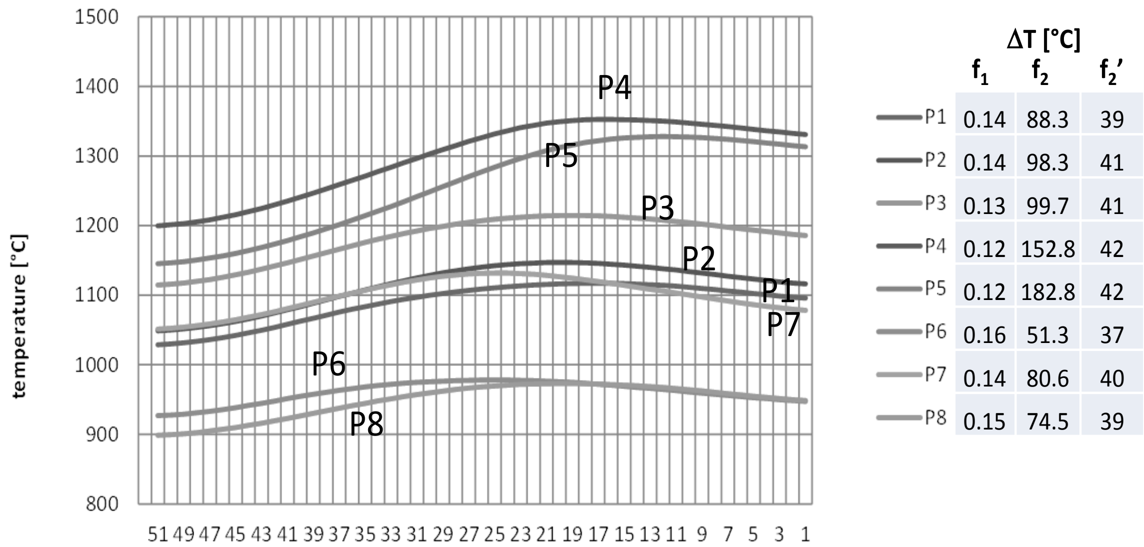


Figure 79. *Pareto front of optimization problem* $f_1(g)$ *and* $f_2(g)$.



	f_1	f_2	f_2'
	ΔT [°C]		
P1	0.14	88.3	39
P2	0.14	98.3	41
P3	0.13	99.7	41
P4	0.12	152.8	42
P5	0.12	182.8	42
P6	0.16	51.3	37
P7	0.14	80.6	40
P8	0.15	74.5	39

Figure 80. *Temperature profiles from the simulations of individuals of the first generation.*

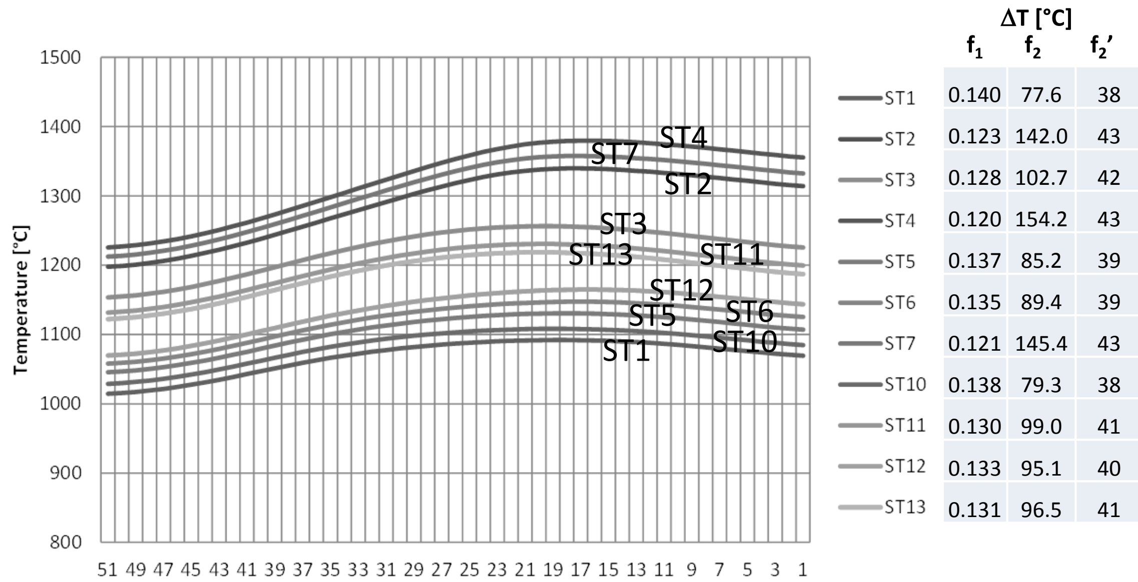


Figure 81. *Temperature profiles from the simulations of individuals of the last generation.*

Finally Figure 82 reports two coil configurations characterized by improved performance. In the first configuration, coil positions leads to an improved temperature uniformity according to $f_2(g)$; in turn, the second configuration leads to an improved electrical efficiency according $f_1(g)$.

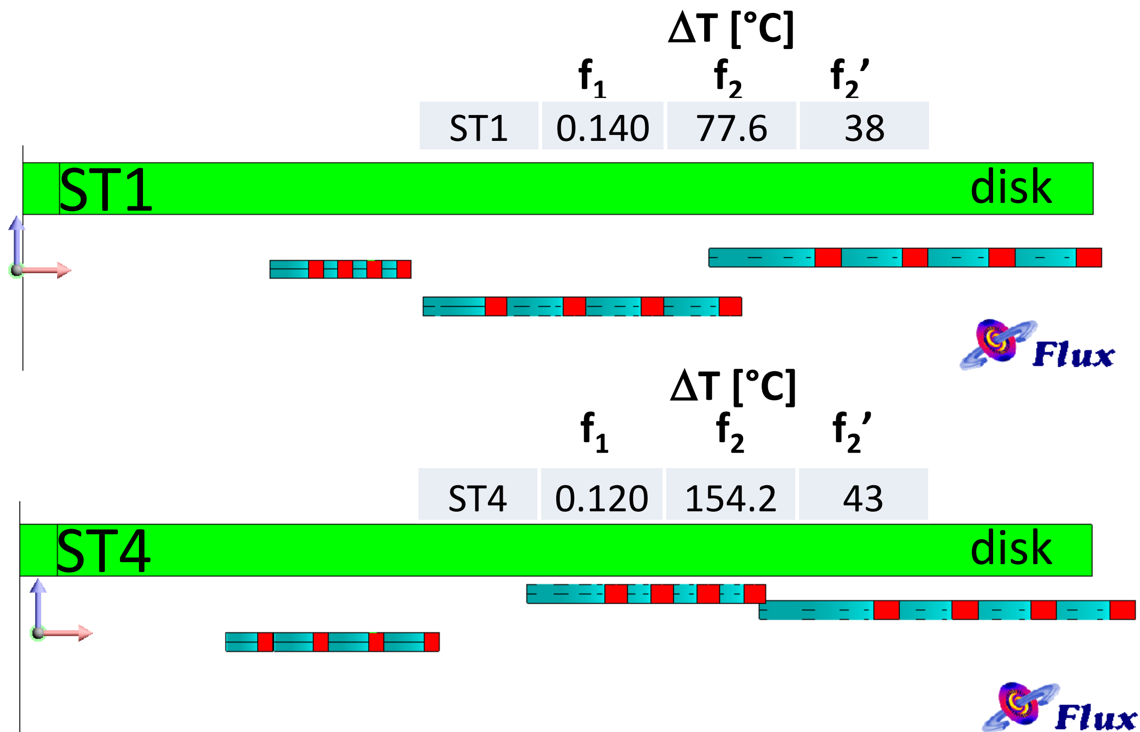


Figure 82. *Coil configurations for two of the Pareto front solutions.*

As regarding the second uniformity criterion, the Pareto front after 16 iterations is shown in Figure 83. An example of temperature profiles for some non-dominated solutions are reported in Figures 84 and 85.

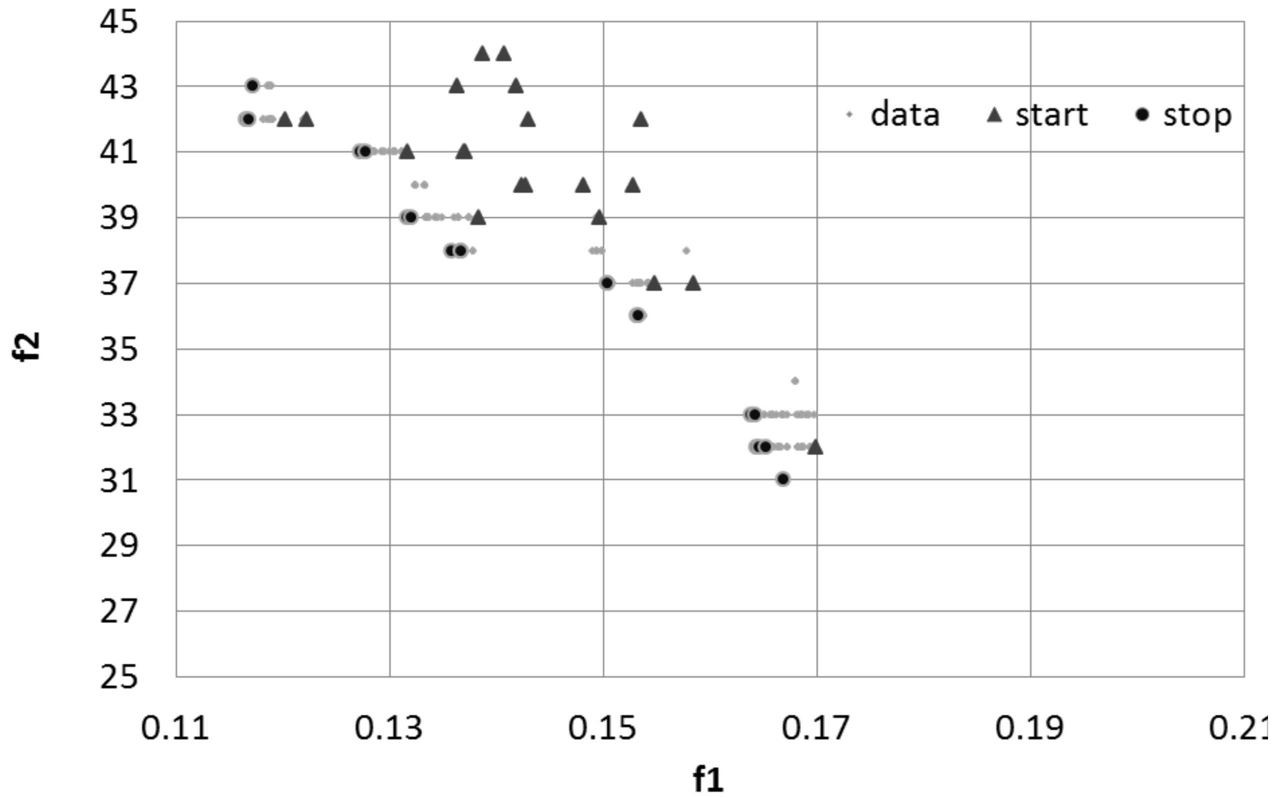


Figure 83. *Evolution of the Pareto front for optimization problem* $f_1(g)$ *and* $f_2{}'(g)$.

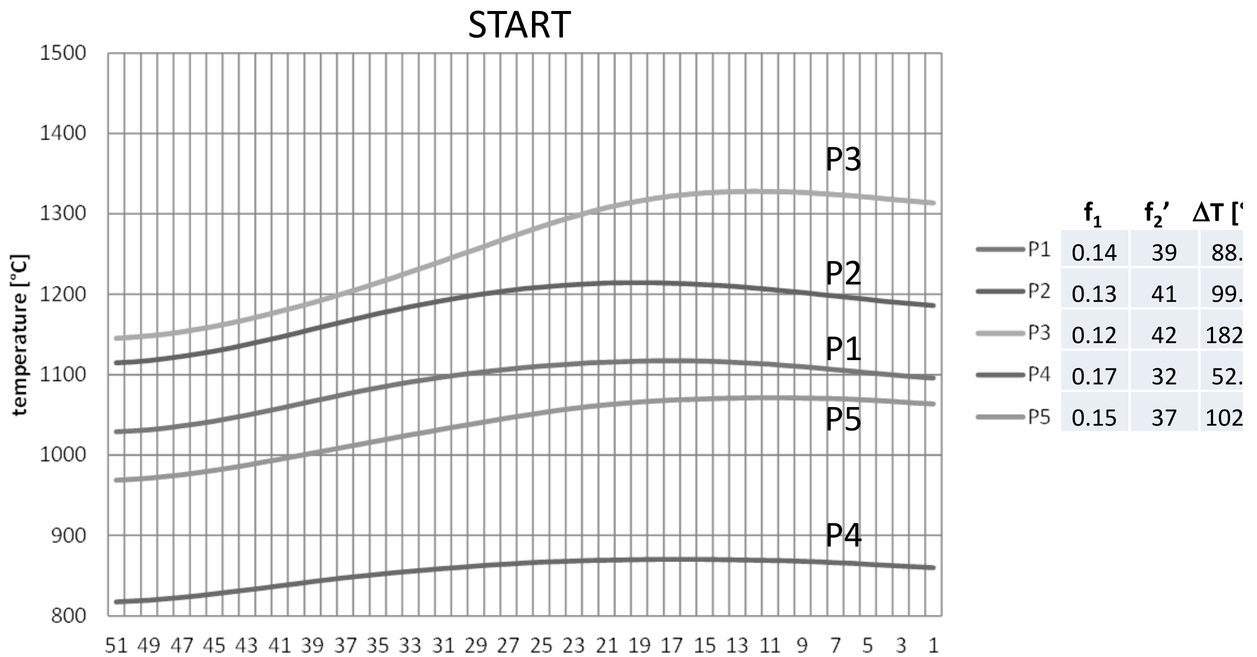


Figure 84. *Temperature profiles from the simulations of individuals of the first generation.*

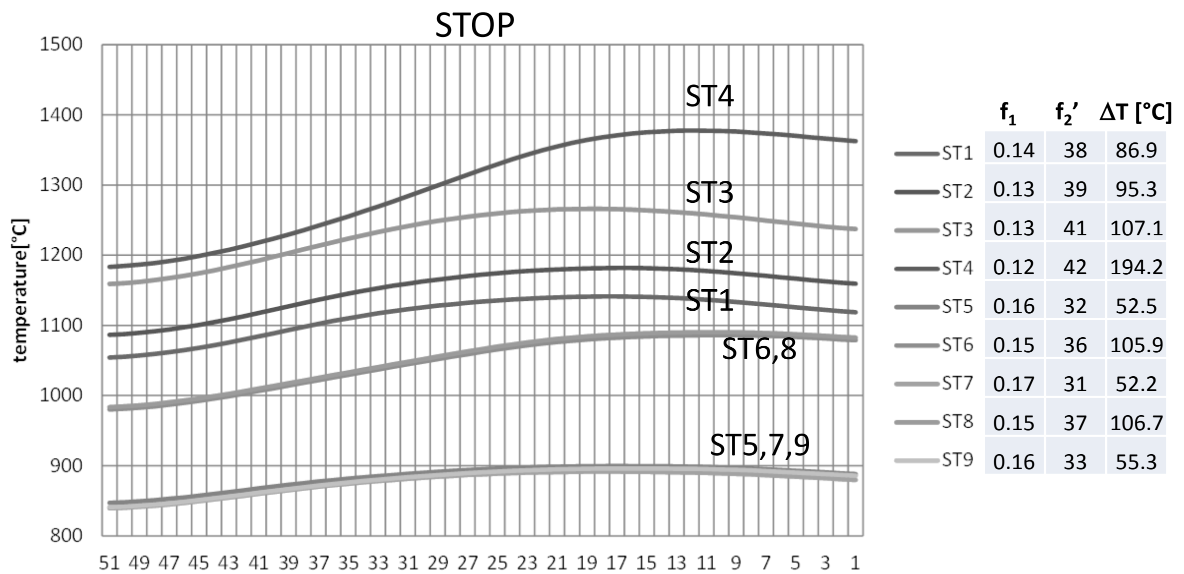


Figure 85. *Temperature profiles from the simulations of individuals of the last generation.*

In Figure 85, it can be observed that it appears that the best temperature gap is lower than in the previous case (52.2 °C vs 77.6°C).

Figure 86 shows two coil configurations corresponding to two solutions on the Pareto front. In the first configuration ST4, the efficiency is high but temperature uniformity is not good. In the second configuration ST7, the efficiency is lower but uniformity is good.

In Figure 87 results obtained from optimization by the two uniformity criteria are shown. The second uniformity criterion leads to more uniform temperature profiles.

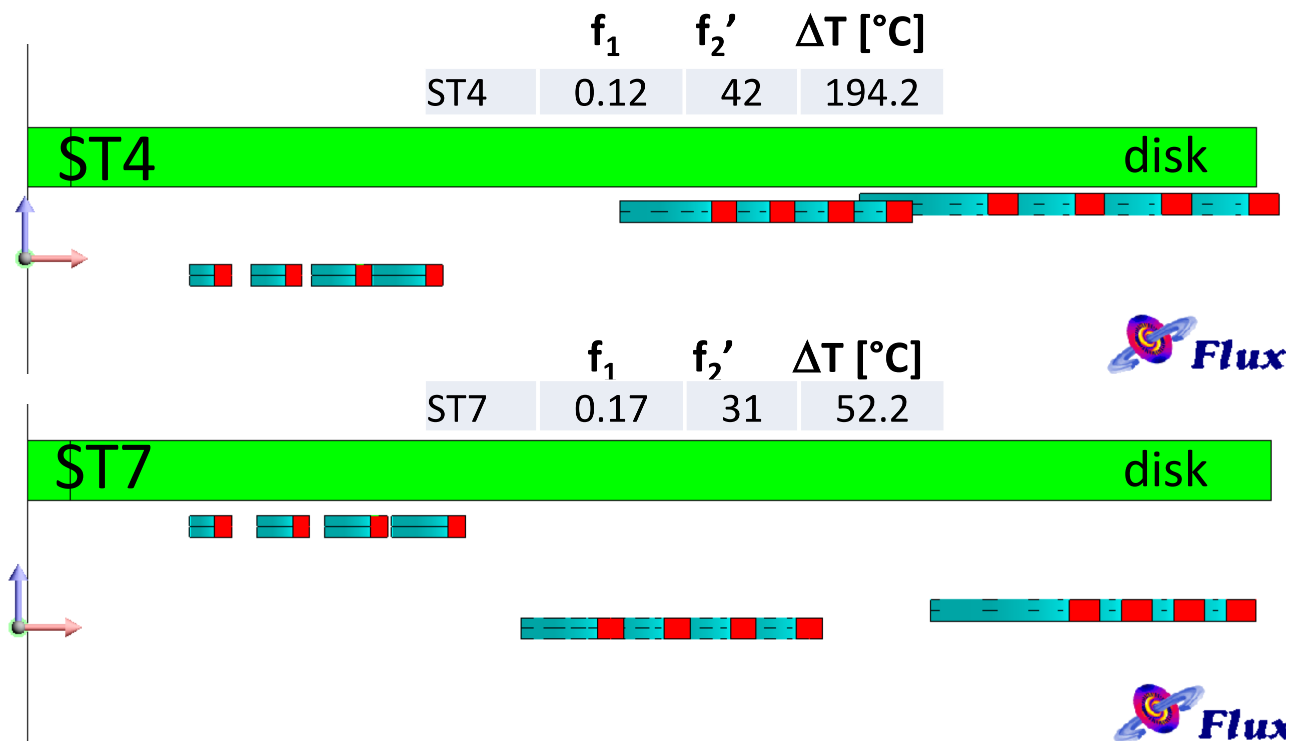


Figure 86. *Coil configurations for two solutions on the Pareto front. On one hand, in ST4 the efficiency is high but uniformity is not good. On the other hand, in ST7 the efficiency is low but uniformity is good.*

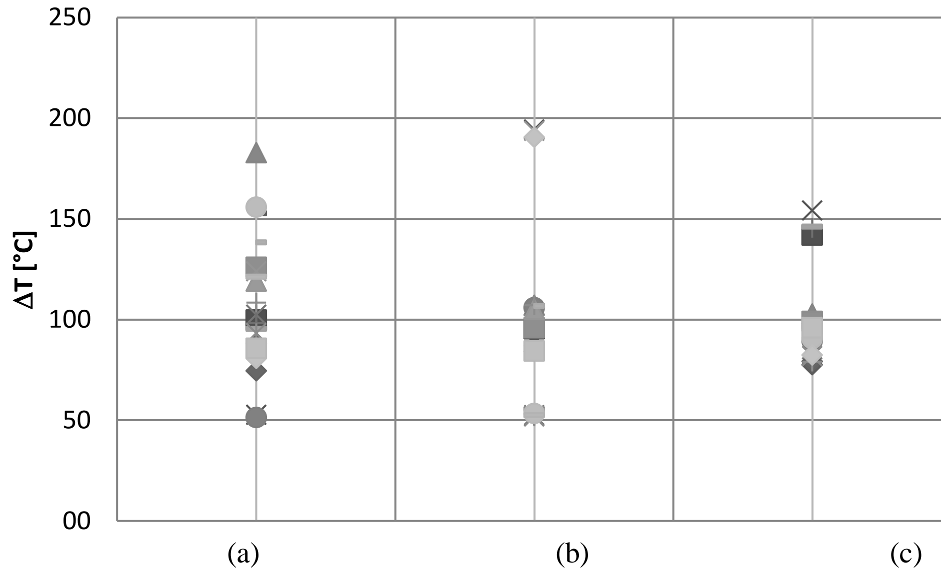


Figure 87. *Difference between the maximum and the minimum temperature along the path, for first generation (a) and last generation individuals. The proposed approach (b) lead to more uniform temperature profiles than in a min-max approach (c).*

Table 13. *Value of objective functions for the first and the last sets of individuals. The rank of the best 20 individuals is reported.*

First generation			Last generation – min-max objective function				Last generation – proximity objective function			
f_1	f_2[°C]	f_2'	f_1	f_2 [°C]	f_2'	rank	f_1	f_2 [°C]	f_2'	rank
0.138	88.3	39	0.140	77.6	38	0	0.136	86.9	38	0
0.132	99.7	41	0.123	142.0	43	0	0.132	95.3	39	0
0.120	182.8	42	0.128	102.7	42	0	0.127	107.1	41	0
0.170	52.7	32	0.120	154.2	43	0	0.117	194.2	42	0
0.155	102.3	37	0.137	85.2	39	0	0.164	52.5	32	0
0.159	51.3	37	0.135	89.4	39	0	0.153	105.9	36	1
0.137	98.3	41	0.121	145.4	43	0	0.167	52.2	31	1
0.122	152.8	42	0.128	100.3	41	0	0.150	106.7	37	1
0.142	95.4	40	0.121	146.0	43	0	0.164	55.3	33	1
0.150	74.5	39	0.138	79.3	38	0	0.137	84.2	38	1
0.136	125.9	43	0.130	99.0	41	0	0.132	95.9	39	2
0.137	119.0	41	0.133	95.1	40	0	0.128	105.1	41	2
0.143	123.9	40	0.131	96.5	41	0	0.117	193.6	42	2
0.142	93.2	43	0.137	81.8	38	0	0.165	51.8	32	3
0.139	155.8	44	0.134	90.6	38	0	0.164	53.4	33	3
0.143	108.4	42	0.139	79.0	38	0	0.153	107.1	36	3
0.148	138.2	40	0.134	93.7	39	0	0.150	106.7	37	3
0.153	121.2	40	0.132	96.3	40	0	0.165	52.4	32	4
0.141	80.6	44	0.137	82.4	39	0	0.117	190.5	43	4
0.154	85.7	42	0.132	95.9	40	0	0.137	84.4	38	5

Further tests should be performed on more complex models, in order to assess the generality of the described optimization procedure. Then, the method could be used in the design of devices in which the temperature distribution strongly affects the quality of a technological process, i.e. induction hardening of gears and silicon epitaxial growth.

In conclusion, parallel computing can be effectively exploited in the optimization process, as solutions of the same generation are independent one another, and can be computed simultaneously. Furthermore, parallel solvers can be exploited by finite element tools for each solution.

8.3 Multiphysics modeling of quasi-resonant induction cooktops

Induction cooktops are gaining an ever increasing share in home cooking market in comparison with gas, resistance or halogen cooktops. The main reasons are high energy efficiency, low heating times, accurate control of the heating power and uniformity, the increased safety, the easy installation and maintenance. Induction cookers are faster and more energy efficient than traditional hobs, as the heat is delivered directly inside the pot.

Manufacturers are looking for more performing and less expensive solutions, by developing new power electronics topology and control techniques, by optimizing the design of inductors, by exploiting innovative materials and smart software for user interface. Typically an induction hob consists in:

- a ceramic glass
- a set of inductors that generate variable magnetic field to heat the pot by induced eddy currents
- an electronic board with a frequency converter
- an user interface for control of heating power levels
- a mechanical structure.

The number of installed inductors can vary from 3 to 5 on different configurations, depending on diameters and shapes of cookware and available power supply. Inductors are placed on an aluminum plate, which acts as mechanical structure and as a shield for the electronic board.

A typical the inductor, reported in Figure 88, consists in:

- a pancake coil with one to three concentric spiral windings, in which stranded, litz copper wire conductors are used. The diameter of each single wire is in the range of 0.2-0.4 mm, depending on the maximum operating frequency. The number of wires in a conductor depends on the maximum inductor current and, thus, on the rated power of the inductor
- a number ferrite cores, as magnetic flux concentrators, arranged radially under the inductor in order to increase the efficiency and to avoid magnetic flux leakage

- a temperature sensor placed in the center of the coil, to protect the inductor insulation from overheating
- a mechanical structure on which all these components are assembled.

(a) Two concentric windings of litz wire

(b) ferrite magnetic flux concentrators and aluminum layer

Figure 88. *Typical arrangement of an inductor for induction hobs.*

In the usual configuration of resonant induction cooktops, a frequency converter, operating at a frequency between 20 and 50 kHz, must resonate with the inductor in order to achieve an optimal efficiency over the broadest possible operation range, by transferring power of about 1-3 kW to the pot. Then, the inductor must be designed in order to realize the Zero Voltage Switching (ZVS) condition for the frequency converter, over the whole range of operating frequency and power levels. These requirements are much more important in the quasi resonant induction cooktops, for limitations in handling the soft switch mode of solid state electronic components.

Among the two typical topologies presented in Figures 89 and 90, the cheapest is the quasi-resonant (QR) type because of a lower number of electronic switches, i.e. IGBT, and resonant capacitors is required. In fact, QR converters are a tradeoff solution between cost and efficiency [116,117]. However, despite of its topological simplicity, the design of quasi-resonant converter and related inductor is quite challenging.

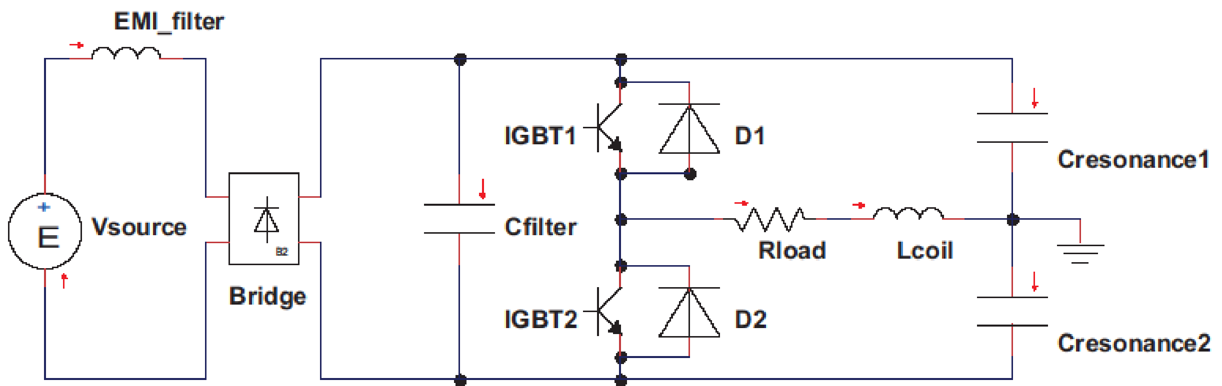


Figure 89. *The half-bridge topology is commonly used for resonant induction hobs.*

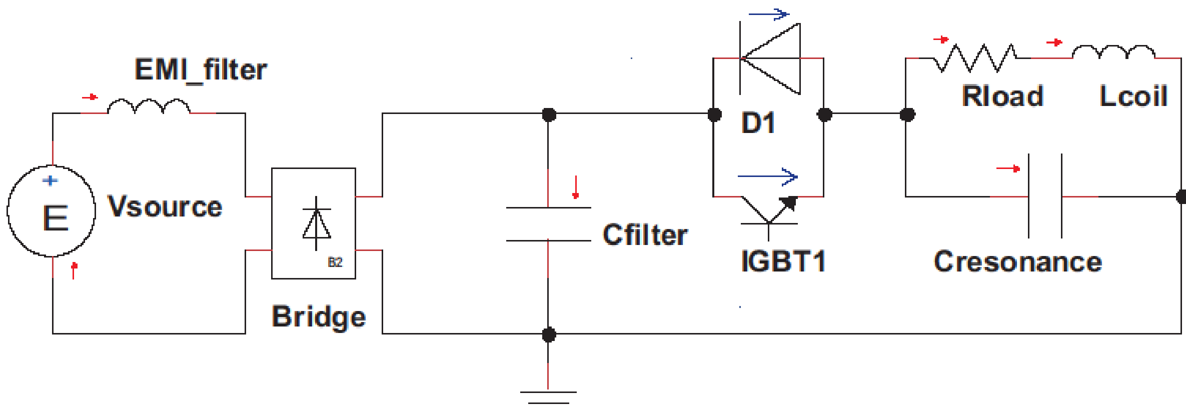


Figure 90. *Quasi-resonant topology is cheaper, as a lower number of electronic switches, i.e. IGBT, and resonant capacitors is required.*

The amount of the power deliverable by a QR converter can be controlled in a continuous way between a minimum and a maximum value. The width of this range, defined as the ratio between the maximum achievable power and the minimum settable power, is influenced by converter characteristics, the coil design and the pot quality. The maximum power is limited by the maximum voltage tolerable for the switch. The minimum deliverable power is due to the ZVS condition at low power. This intrinsic limit for the continuous regulation at very low power levels, is usually overcome through a time partitioned heating.

Thus, the main variables to be controlled for a good operation of the QR converter are the output power and the loading conditions. For this reason a precise design of the inductor is needed, by taking into account the real behavior of the supply voltage and current, depending on the operation mode of the QR converter.

In order to design such devices, a procedure based on a 3D finite element electromagnetic model of the inductor, coupled with the frequency converter circuital model and a suitable control logic has been developed.

In the half-bridge resonant topology, the power delivered to the pot is controlled by sweeping the operating frequency close to resonance frequency. In order to simulate such systems, 2D or 3D time-harmonic electromagnetic problems can be effectively used. For each different set of design parameters, a set of simulations might be performed for relevant frequency range. Then, most important quantities, as the inductor current and the power delivered to the inductor and to load, could be easily evaluated.

One of the control technique in a QR converter could be to set the peak current value, while the frequency depends on the natural oscillation of RLC circuit. In this case, an electromagnetic transient analysis must be performed, in order to evaluate inductor current and delivered power. Furthermore, through a coupled simulations, the operation mode of the QR converter can be tested, as for a real experimental test setup.

Inductors are modeled by a 2D axisymmetric FEM model, shown in Figure 91, in which the main regions of the model are presented. Ferrite magnetic flux concentrators are modelled as a continuous ferrite ring with an equivalent magnetic permeability.

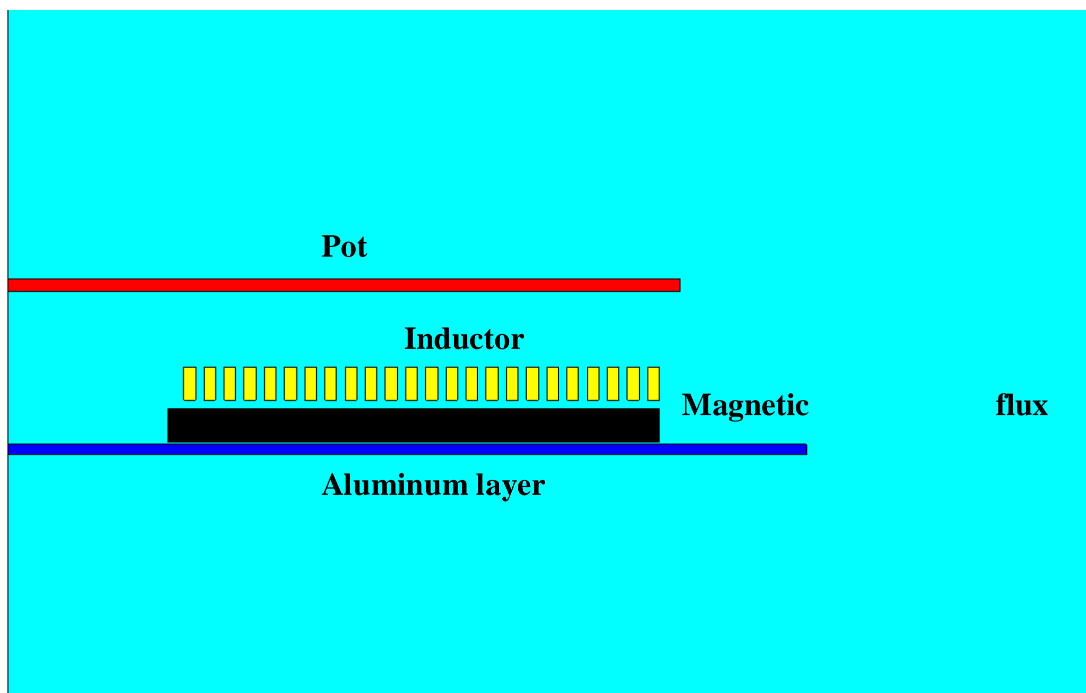


Figure 91. *2D axisymmetric FEM model of the device.*

The external diameter of the coil is 150 mm, intended for the small pot. In litz copper wired conductors, proximity and skin effects are negligible, thus coils are modeled as perfect coils. The pot, represented by a metallic disk, is typically made of non-linear magnetic steel (AISI 409 or AISI430). The mesh must be suitable for both the aluminum layer and the non-linear steel layer, in different magnetic saturation conditions. In this example, a 165000 nodes, second order element mesh is used, and the electromagnetic problem is solved in terms of vector magnetic potential and electric scalar potential formulation.

The 2D model solution returns some useful integral parameters of the device, as the equivalent resistance and inductance. However, magnetic saturation effects in the ferrite yokes must be modeled in a 3D FEM simulation (Figure 92). In order to reduce the solution time, symmetry conditions are posed and only $1/2\mathrm{n}$ of the device is modeled, where n the number of ferrites cores.

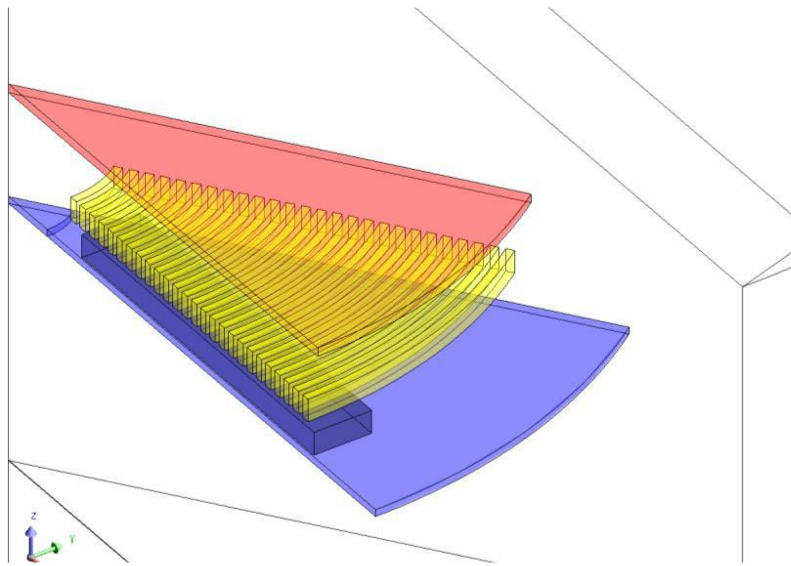


Figure 92. *3D FEM model of the device.*

Both 2D and 3D models must be coupled with the external circuit in Figure 93, where the voltage source supply is controlled by a quasi-resonant converter circuit simulation. The switch is controlled in a T_{on} and a T_{off} phases. In particular, a T_{on} signal is used by the control logic to set the power levels, in the circuit simulation, and sent to the electromagnetic simulation. The T_{off} signal, that determines the frequency, is not fixed and depends on the behavior of the transient electromagnetic solution. It must be computed by the logic, through a circuital equation which cannot be solved within the FEM model.

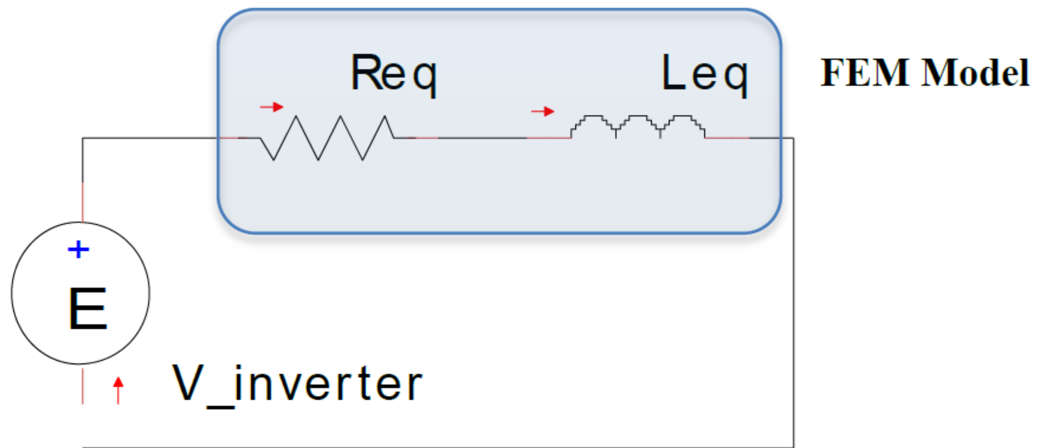


Figure 93. *The supply circuit coupled to 2D and 3D FEM model*

In order to simulate the behavior of the system, an automated evaluation of T_{off} and of the switching frequency is needed. Thus, a coupled multiphysics simulation has been performed by Portunus and Flux software. The electrical circuit described in Figure 88, supplying the inductor in the FEM model, must be controlled by the QR converter circuit simulation and an adequate control logic. In Figure 94, the multiphysics model is presented.

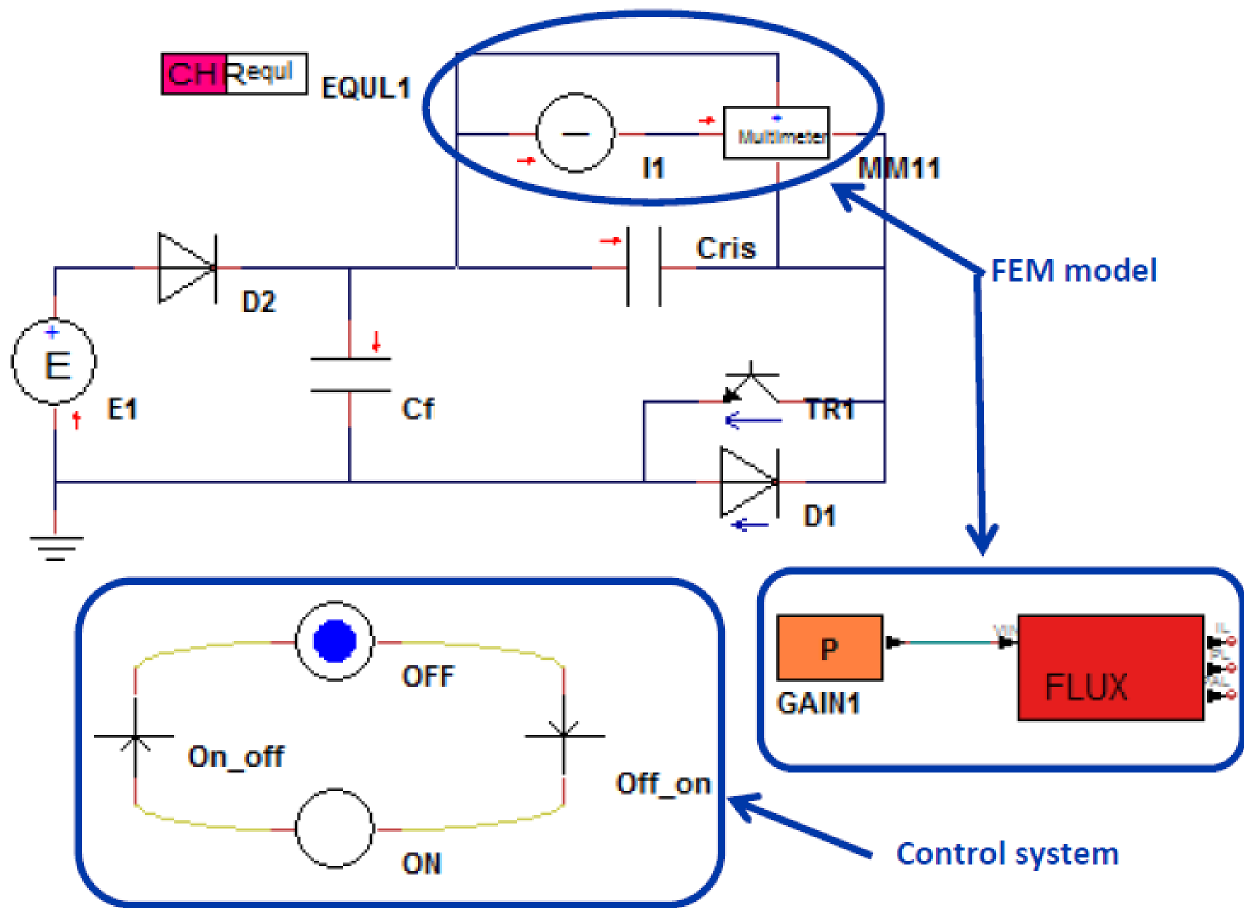


Figure 94. *Multiphysics modelization of a QR converter. A FEM model is coupled to a circuit model and a control logic model, and a transient simulation is performed.*

It should be pointed out that the FEM model is fully coupled with the circuit and control logic of QR converter. The electrical quantities calculated by the FEM model are used for the calculation of electrical parameters like voltage, current and power delivered by the converter at each time step of the transient analysis.

The multiphysics model relies on the following assumptions:

- the rectifier bridge and the EMI filter, towards the supply network, are neglected
- the solution is obtained by considering the rectified peak voltage
- the control logic is based on the analysis of the inductor current: T_{on} just before the positive current peak is reached, and T_{off} is determined by the time of zero crossing of current
- the maximum value for the peak current is determined by both the maximum current limit of the IGBT during the T_{on} phase, and the maximum voltage on the IGBT during T_{off} phase (most critical condition)
- the minimum value for the peak current is limited by the hard switching condition

The simulation results can be regarded as circuital results and FEM results. As regards the circuital quantities, an example of the main electrical quantities in the circuit and related control signal is reported in Figure 95.

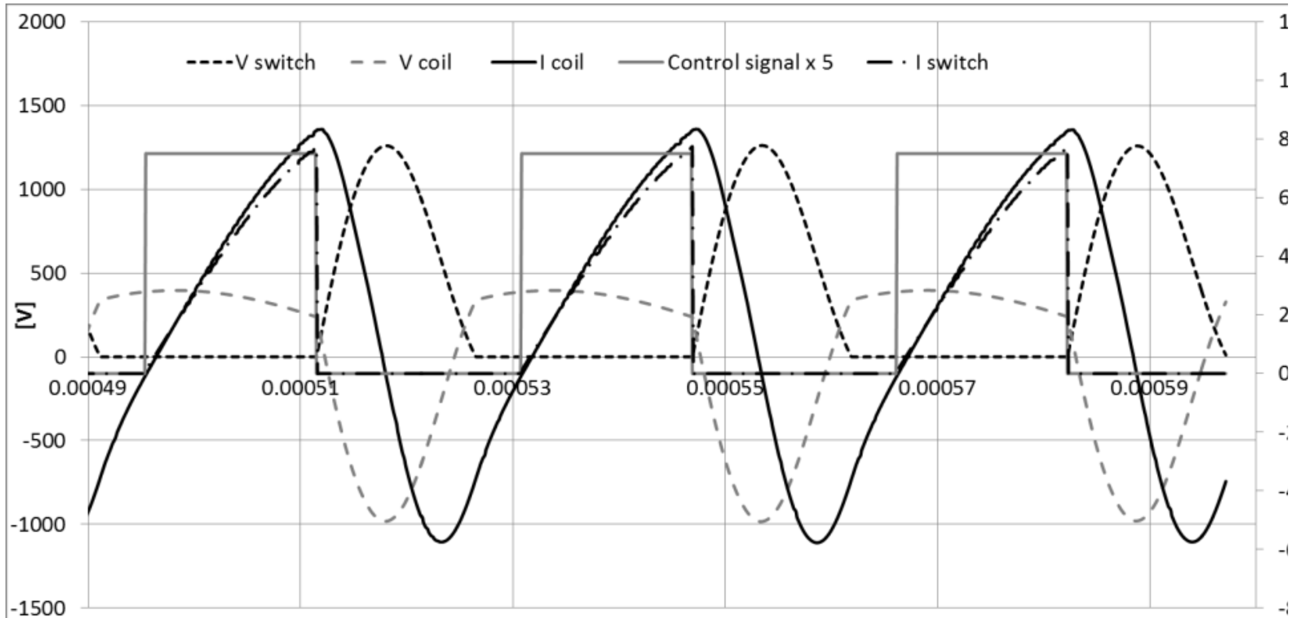


Figure 95. *Example of the main electrical quantities in the circuit and related control signal. Waveform of coil current, coil voltage, control signal for IGBT, IGBT current and IGBT voltage are shown.*

Other useful information can be derived from the 3D FEM simulation results, such as the power delivered to the inductor, the magnetic flux density in the ferrite cores and the power density distribution in the pot (Figure 96). Among all, the magnetic flux density distribution in the ferrite cores is extremely important for a suitable design of the device, and it can only be evaluated by a 3D model. In particular, eventual magnetic saturation effects in ferrite cores deteriorate the device performances, as heat losses in the ferrites and the aluminum plates arise. The power density distribution in the pot is another important information, as it can be correlated to the uniformity of heat generation in the pot (some foods are very sensitive to small temperature differences) and to the electrical efficiency of the device.

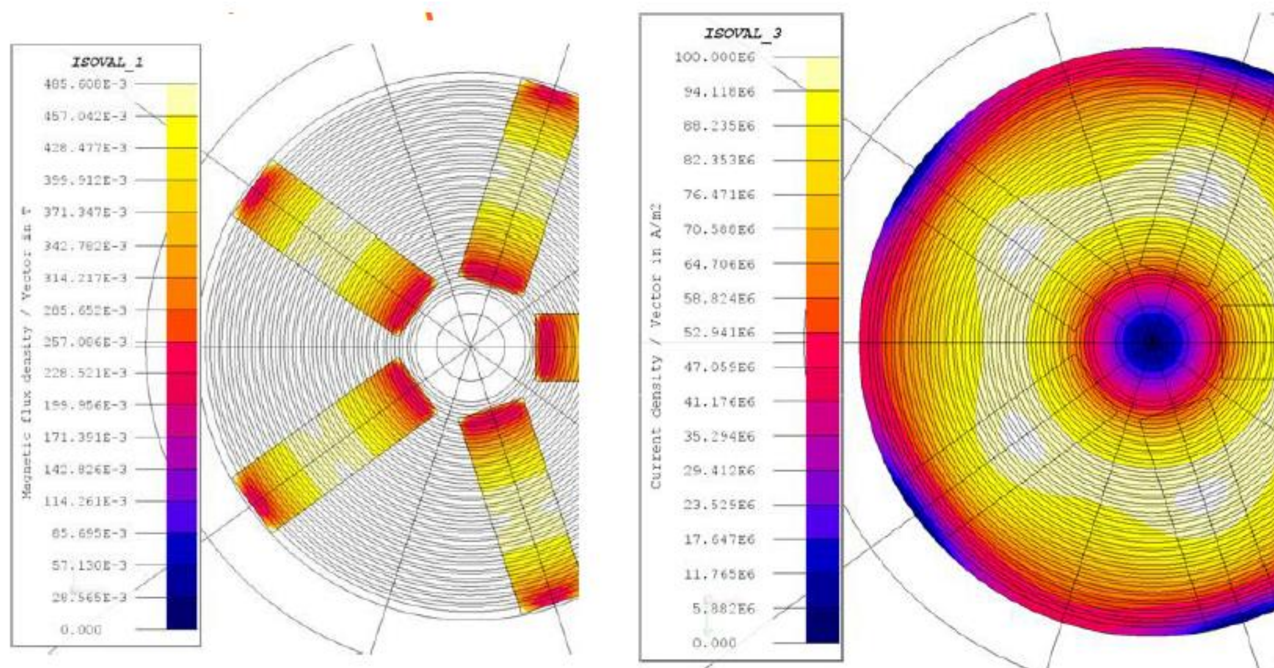


Figure 96. *Magnetic flux density in the ferrite cores and the power density distribution in the pot are among the important information from the 3D FEM model.*

Simulation results have been compared with experimental measurements on some prototypes. Geometrical and electrical parameters have been selected by meeting the requirements on the QR converter limitations, as the maximum continuous operation power, the minimum continuous operation power and the external diameter of the coil. The experimental tests on a series of prototypes demonstrated the suitability of the procedure, by giving errors in the range of 3-6 % in all the important metrics, for both the inductor and the QR converter components.

The solution of the described multiphysics problem is extremely time consuming because a high number of time steps is needed in order to obtain an accurate transient solution. Furthermore, at each time step, a non-linear electromagnetic problem must be computed with a good accuracy, by using very large meshes. The computing time for a transient multiphysics solution is in a range of one day for the 2D model, and 1 week for the 3D model, by an ICCG-type sequential solver. By exploiting different parallel solvers developed in the framework of this thesis, a remarkable reduction in the computing time has been achieved, and simulation time has decreased to only 18 hours for the 3D model based simulation, on an eight-core shared memory system.

In conclusion, all important parameters both for the converter and the inductor can be accurately calculated, thus allowing for an optimized design of the device.

Conclusions

In the modeling of induction heating applications, as well as for structural and fluid dynamics problems, the discretization of continuous problems by differential numerical methods leads to large sparse linear systems. Accordingly, the solution of such linear systems is the most limiting factor in the simulation of increasingly detailed and complex finite-element models.

As a conclusion of this thesis, a general review of the most remarkable results is outlined.

A significant reduction of the computing time needed for the simulation of induction heating processes has been achieved. Non-linear phenomena as magnetic saturation in magnetic flux concentrators, iron core losses and radiative thermal exchanges, eddy current losses and temperature distribution in heated parts can be taken into account in the solution of more detailed finite-element simulations.

In the first part of the present thesis, the full potential of simulation software has been exploited through the use of high performance parallel computing techniques. In particular, the simulation of induction heating processes could be accomplished within reasonable computing times, by implementing parallel direct solvers in the solution process.

The performance of the MUMPS package, an open source library for computation in distributed memory system, has been compared to other state-of-the-art libraries on a shared memory system. Results of this work have been presented in some prestigious conferences [104]. The contribution of the author is the development of an interface to a commercial finite-element tool, in order to obtain large sparse matrices from the simulation of induction heating processes. Then, different parallel direct solvers for the solution of large sparse linear systems have been implemented and benchmarked on relevant models developed through a commercial software.

Furthermore, outstanding improvements have been achieved in the context of a joint project developed by the MUMPS team in Bordeaux, Lyon and Toulouse, and supported by many French institutions, as CERFACS, CNRS, ENS Lyon, INPT(ENSEEIHT)-IRIT, INRIA and University of Bordeaux. Thanks to a fruitful collaboration with MUMPS team, such very recent functionalities developed in Lyon and Toulouse, could be tested on matrix benchmarks from induction heating models. The contribution of the author to this cooperation is the realization of different matrix benchmarks from typical induction heating process simulations. Experimental tests, performed by MUMPS team, show a very significant impact on the computing time and memory needed with respect to reference commercial codes.

In particular, a new multithreading approach has been introduced in MUMPS, in order to exploit tree parallelism in a shared memory system, as a multicore computer. In addition, BLAS optimized libraries allowed for an efficient exploitation of node parallelism for factorization

calculations. Then, a new low-rank approximation technique has been exploited in MUMPS, allowing for a remarkable reduction of both the number of operations and the amount of memory needed for the matrix factorization.

The application of such improvements to a distributed memory system is the natural prosecution of this work, as each node of a modern cluster consists in a shared memory subsystem, and MUMPS library was initially intended for computations on a distributed memory system.

In the second part of the thesis, the developed parallel solvers have been exploited in a set of finite-element simulations. In particular, successful examples of design methodology by virtual prototyping have been described. Complex multiphysics simulations have been performed by involving electromagnetic and thermal field problems coupled with circuital and mechanical problems. Experimental tests carried out on the realized prototypes confirmed the expected performance.

Multiobjective stochastic optimization algorithms have been successfully applied to multiphysics 3D model simulations in search of a set of different device configurations, characterized by improved efficiency and temperature uniformity on the heated parts. Also in this case, parallel computing plays a major role, as the number of simulations to be performed in optimization algorithms can be very high.

Various papers have been published on international journals [107,112,113]. The contribution of the author to these works is the production of complex multiphysics models and the development of new optimization strategies, as well as the acceleration of the simulation time through parallel computing with respect to the serial computing.

As computational power is rapidly increasing by the introduction of new hardware capabilities, future work will focus on the development of numerical methods able to benefit from the high number of cores available. Moreover, an accurate solution of multiphysics problems will lead to a deeper understanding of physical phenomena, allowing for better solutions to engineering problems.

Nomenclature

f		In a code, fraction of the execution time in which the instructions could be parallelized
S		Speed-up achievable for a parallelizable fraction f of a code
Np		Number of working threads in a parallel region
$\mathbf{A}$		Matrix of entries A_{ij}
$\mathbf{x}, \mathbf{y}$		Solution vector
$\mathbf{b}$		Right-hand side vector
P		Total energy associated to a system
v		Generic function subject to the prescribed boundary conditions
u		Particular function that minimizes the energy $P(v)$
V		Generic linear combination of test functions
U		Particular linear combination that minimizes the energy $P(V)$
n		Number of test functions, or dimension of a problem
H	[A/m]	Magnetic field
J	[A/m^2]	Current density
J_0	[A/m^2]	Impressed current density
E	[V/m]	Electric field
T	[A/m]	Electric vector potential
T_0	[A/m]	Impressed electric vector potential
A	[Vs/m]	Magnetic vector potential
A_r	[Vs/m]	Reduced magnetic vector potential
V	[V]	Electric scalar potential
ϕ	[A]	Magnetic scalar potential
Ψ	[A]	Total magnetic scalar potential

H_s	[A/m]	Impressed magnetic field
Ω_n		Non-conducting region
Ω_c		Conducting region
Γ_n		Boundary of non-conducting region
Γ_c		Boundary of conducting region
Γ_{nc}		Interface between Ω_n and Ω_c
σ	[S/m]	Electrical conductivity
μ	[H/m]	Magnetic permeability
NNZ, N_z		Number of non-zero entries in a matrix
$\mathbf{U}$,$\mathbf{L}$		Upper and lower triangular matrices arising from factorization phase
$\mathbf{S}$,$\mathbf{M}$		Preconditioning matrices
$\mathbf{R}$		Residual matrix
ε		Accuracy of low-rank approximation
k_ε		Rank at accuracy ε
$\mathbf{W}$,$\mathbf{Z}$		Matrices from low-rank approximation form
α	[°]	Angular width
p		Number of magnetic poles
R_k, d_k, h_k		Design variables for k-th individual
g		Individual characterized by a set of value for the design variables
$f_1(g)$, $f_2(g)$, $f_2{}'(g)$		Objective functions
γ		Path for the evaluation of the objective functions
T_{ON}, T_{OFF}		Control signal for Quasi-Resonant cooktop multiphysics model

Bibliography

1. Barney, B. *Introduction to Parallel Computing*, Tutorial from Lawrence Livermore National Laboratory, website: computing.llnl.gov/tutorials/parallel_comp, last visit on Dec. 2013.
2. Report from Grand Challenges Task Force, website: www.nsf.gov/cise/aci/taskforces/TaskForceReport_GrandChallenges.pdf, last visit on Dec. 2013.
3. Access Grid, website: www.accessgrid.org, last visit on Dec. 2013.
4. Amazon Web Services, website: aws.amazon.com, last visit on Dec. 2013.
5. Sørensen N. (2011). *Industry Benchmarks Performance*, Cisco Systems.
6. TOP 500, website: www.top500.org, last visit on Dec. 2013.
7. Hill, M.D.; Marty, M.R., *Amdahl's Law in the Multicore Era*, Computer , vol.41, no.7, pp.33,38, July 2008, doi: 10.1109/MC.2008.209 .
8. Vajda, A. (2011). *Programming Many-Core Chips*. Springer.
9. Hermanns, M. (2002). Tutorial: *Parallel Programming in Fortran 95 using OpenMP*.
10. Gerber, R. (2012). *Tutorial: Getting Started with OpenMP*, website: software.intel.com/en-us/articles/getting-started-with-openmp , last visit on Dec.2013.
11. Lawson, C.L., Hanson, R. J., Kincaid, D., and Krogh, F. T. (1979) *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5, pp. 308-323.
12. Dongarra, J.J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 14, pp. 1-17.
13. Intel MKL Math Kernel Library, website: software.intel.com/en-us/intel-mkl last visit on Dec. 2013.
14. Report from MPI Forum, website: www.mpi-forum.org last visit on Dec. 2013.
15. MPICH User's Guide, website: www.mpich.org last visit on Dec. 2013.
16. Krishna, J., (2010). *Implementing MPI on Windows: Comparison with Common Approaches on Unix*, in book "Recent Advances in the Message Passing Interface",Lecture Notes in Computer Science Volume 6305, pp 160-169, Argonne National Laboratory.
17. Torp, A. (2009). *Sparse linear algebra on a GPU with Applications to flow in porous Media*. Thesis at Norwegian University of Science and Technology.
18. Hennessy, J., and Patterson, D. (2003). *Computer architecture: a quantitative approach*. Morgan Kaufmann.
19. Bell, N. and Garland, M. (2008). *Efficient Sparse Matrix-Vector Multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004.
20. NVIDIA Tesla GPU Technical Specifications, website: www.nvidia.com/object/tesla_tech_specs.html last visit on Dec. 2013.

21. Intel Product Information, website: ark.intel.com last visit on Dec. 2013.
22. Altera Corporation, *From Multicore to Many-Core: Architectures and Lessons*, website: www.altera.com/technology/system-design/articles/2012/multicore-many-core.html last visit on Dec. 2013.
23. Falsafi, B. (2009). *Energy-Centric Computing & Computer Architecture*. Proceedings of the 2009 Workshop on New Directions in Computer Architecture, New York ,USA.
24. Chua, L. O., (1971). *Memristor-the Missing Circuit Element*. IEEE Transactions on Circuit Theory 18(5), pp. 507-519.
25. Kurian, G., Miller, J. E., Psota, J., Eastep, J., Liu, J., Michel, J., Kimerling, L. C., Agarwal, A. (2010). *ATAC: a 1000-core Cache Coherent Processor with On-Chip Optical Network*. Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 477-488.
26. Zimmerman, W. B., (2004). *Process Modelling and Simulation with Finite Element Methods*, World Scientific Publishing.
27. Zienkiewicz, O.C.; Taylor, R.L.; Zhu, J.Z. (2005). *The Finite Element Method: Its Basis and Fundamentals* (Sixth ed.). Butterworth-Heinemann.
28. Strang, G. (2008). *Algebra lineare*, pp.359-380, Apogeo.
29. Muhlbauer A. (2008). *History of Induction Heating and Melting*, pp.374-375, Vulkan-Verlag
30. Silvester, P., and Haslam, C.R.S., (1972). *Magnetotelluric: modelling by the finite element method*, Geophys. Prospecting, Vol 20, No 4, pp. 872-891.
31. Chari, M.V.K., (1973). *Finite element solution of the eddy current problem in magnetic structures*, IEEE Trans Power App. & Syst., Vol. PAS-92.
32. Foggia, A., Sabonnadiere, J. C., Silvester, P., (1975). *Finite element solution of saturated travelling magnetic field problems*, Power Apparatus and Systems, IEEE Transactions on, vol.94, no.3, pp.866-871 doi: 10.1109/T-PAS.1975.31917 .
33. McDonald, B. H, and Wexler, A. (1972). *Finite-element solution for unbounded field problems*, IEEE Trans. Microwave Theory Tech., Vol. 20, No.12, pp. 841-847.
34. Sundberg, Y., (1979). *Electric furnaces and induction stirrers*, ASEA, Metallurgical Industries Division, Internal Report.
35. Lavers, J.D., (1983). *Numerical solution methods for electroheat problems*, Magnetics, IEEE Transactions on, Vol. 19 , Issue 6, pp. 2566-2572.
36. Lavers, J.D., (1989). *Computational methods for the analysis of molten metal electromagnetic confinement*, ISIJ Int., Vol. 29.
37. Massé, P., Morel, B., Breville, T., (1985). *A finite element prediction correction scheme for magneto-thermal coupled problems during Curie transition*, IEEE Trans. Magnetics, Vol. 21.
38. Maten, E.J.W. ter, Melissen, J.M.B., (1992). *Simulation of inductive heating*, IEEE Trans. Magnetics, Vol. 28.
39. Cedrat Flux, website: www.cedrat.com, last visit on Dec. 2013.
40. Ansys Maxwell, website: www.ansys.com*, last visit on Dec. 2013.*

41. *Vector Fields Simulation Software, website:* operafea.com last visit on Dec.2013.
42. Quick Field, website: www.quickfield.com last visit on Dec. 2013.
43. Comsol Multiphysics, website: *www.comsol.com last visit on Dec. 2013.*
44. Carpenter, C.J., (1977). *Comparison of alternative formulations of 3-dimensional magnetic-field and eddy-current problems at power frequencies*, Proc. IEEE 124(11), pp. 1026-1034.
45. Rodger, D. and Eastham, J.F., (1983). *A formulation for low frequency eddy current solutions*, IEEE Trans. on Magnetics, 19, pp. 2443-2446.
46. Emson, C.R.I, and Simkin, J., (1983). *An optimal method for 3-D eddy currents*, IEEE Trans. on Magnetics, 19,pp. 2450-2452.
47. Leonard, P.J., and Rodger, D. (1988). *Finite element scheme for transient 3D eddy currents*, IEEE Trans. on Magnetics, 24 , pp. 90-93.
48. Nakata, T., Takahashi, N., Fujiwara, K., and Okada, Y. (1988). *Improvements of the T - Ω method for 3-D eddy current analysis*, IEEE Trans. on Magnetics, 24, pp. 94-97.
49. Birò, O. and Preis, K., (1989). *On the use of the magnetic vector potential in the finite element analysis of 3-D eddy currents*. IEEE Trans. Magnetics, 25, pp. 3145-3159.
50. Birò, O. and Preis, K. (1990). *Finite element analysis of 3-D eddy currents*, IEEE Trans. Magnetics, 26 pp. 418-423.
51. *Preis, K., Bardi, I.,* Birò*, O., Magele, C., Renhart, W., Richter, K.R., and Vrisk, G. (1991). Numerical analysis of 3D magnetostatic fields, IEEE Trans. Magnetics, 27 pp. 3798-3803.*
52. Birò*, O. (1993). Solution of TEAM benchmark problem #10 (Steel plates around a coil), ACES J. 8(2) pp. 203-215.*
53. Birò*, O. (1997). Edge element formulations of eddy current problems, Elsevier.*
54. *Kameari, A. (1990). Calculation of transient 3D eddy current using edge elements, IEEE Trans. Magnetics 26, pp. 466-469.*
55. *Bossavit, A. (1990). Solving Maxwell equations in a closed cavity and the question of 'spurious modes', IEEE Trans. Magnetics 26, pp. 702-705.*
56. *Cendes, Z.J. (1990). Vector finite elements for electromagnetic field computation, IEEE Trans. Magnetics 27, pp. 3958-3966.*
57. Albanese, R., and Rubinacci, G. (1990). *Magnetostatic field computations in terms of two-component vector potentials*, Int. J. Numer. Methods 29, pp. 515-532.
58. Kolbe, E., Reiss, W., (1963). *Eine Methode zur numerischen Bestimmung der Stromdichteverteilung*, Wiss.Z. Hochsch. Elektrot., Ilmenau, Bd. 9, no. 3.
59. Kogan, M.G., (1966). *Calculation of inductors for heating rotational bodies*, Moscow, VNIIEM.
60. Tozoni, O.V., (1967). *Calculation of the electromagnetic fields using computers*, Kiev, Ukraine.
61. Silvester, P., (1967). *AC resistance and reactance of isolated rectangular conductors*, IEEE Trans. Power App. & Sys., Vol. PAS-86.

62. Mayergoyz, I. D., Bedrosian, G., (1995). *On Calculation of 3-D Eddy Currents in Conducting and Magnetic Shells*, IEEE Trans. on Magnetics, Vol. 31 No. 3, pp. 1319-1324.
63. Mayergoyz, I.D., (1972). *Integral equations for the three-dimensional time variable magnetic fields*, Izvestia VUZ of USSR, No. 7.
64. Tozoni, O.V., Mayergoyz, I.D., (1974). *Calculation of 3D electromagnetic fields*, Kiev, Technika.
65. Tozoni, O.V., (1975). *The method of secondary sources in electrical engineering*, Energia, Moscow.
66. Nemkov, V., Demidovich, V., (1988). *Theory and calculation of induction heating devices*, Leningrad, Energoatomizdat.
67. Fawzi, T.H., Burke, P.E., (1974). *Use of surface integral equations for the analysis of TM-induction problems*, Proc. IEE, Vol. 121.
68. Hodgkins, W.R., Waddington, J.P., (1982). *The solution of 3D induction heating problems using an integral equation method*, IEEE Trans. Mag. MAG-18(2).
69. Lean, M.H., Bloomberg, D.S., (1984). *Nonlinear Boundary Element Method for two-dimensional fields*, J. of Applied Physics, Vol. 55, no. 6.
70. Yildir, Y.B., (1991). *Three dimensional analysis of magnetic fields using the Boundary Element Method*, EEIC/ICWA Conference, Boston, USA.
71. Ruehli, E., Antonini, G., Esch, J., Ekman, J., Mayo, A., Orlandi, A. (2003). *Non-Orthogonal PEEC Formulation for Time and Frequency Domain EM and Circuit Modeling*. IEEE Trans.on Electromagnetic Compatibility , 45(2), pp.167–176.
72. Saad, Y., (2003). Iterative Methods for Sparse Linear Systems 2nd edition, SIAM.
73. Matrix Market Format, website: math.nist.gov/MatrixMarket, last visit on Dec. 2013.
74. Gould, N., Hu, Y., Scott, J., (2005*). A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations*, CCLRC report, RAL-TR-2005-005.
75. Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Vorst, H.A. van der, (2001). Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, website: www.netlib.org, last visit on Dec. 2013.
76. Irons, B. M., (1970). *A frontal solution scheme for finite element analysis*. Int. J. Numer. Methods Eng. 2, pp. 5-32.
77. Duff, I. S., Erisman, A. M., Reis, J. K. (1989). *Direct Methods for Sparse Matrices*. Oxford Science Publications.
78. El Boukili, A., Madrane, A., Vaillancourt, R., (2004) *Multifrontal solution of sparse unsymmetric matrices arising from semiconductor equations*, CRM-3125.
79. Duff, I. S., Reid, J. K., (1983). *The Multifrontal Solution of Indefinite Sparse Symmetric Linear*, ACM Trans. on Mathematical Software (TOMS), v.9 n.3, pp.302-325.

80. Amestoy, P. R., Buttari, A., Duff, I. S., Guermouche, A., L'Excellent, J.-Y., Uçar, B., (2011). *The Multifrontal Method*, Encyclopedia of Parallel Computing, pp. 1209-1216. Springer.
81. Amestoy, P., *Factorisation de grandes matrices creuses non symétriques basée sur une méthode multifrontale dans un environnement multiprocesseur*, Doctoral dissertation, Université de Toulouse, 1990.
82. Amestoy, P., Ashcraft, C., Boiteau, O., Buttari, A., L'Excellent, J.-Y., Weisbecker, C., (2013). *Improving multifrontal methods by means of block low-rank representations*, SIAM.
83. Weisbecker, C., Amestoy, P., Buttari, A., (2013). An efficient solution of large sparse linear systems through low-rank matrix approximations. PhD thesis at INPT-ENSEEIHT-IRIT, Toulouse, France.
84. Weisbecker, C., (2013). *Block Low-Rank (BLR) approximations to improve multifrontal sparse solvers*. Presentation, Sparse Days 2013, CERFACS, Toulouse, France.
85. Guermouche, A., L'Excellent, J.-Y., Utard, G., (2003). *Impact of reordering on the memory of a multifrontal solver*, Parallel Computing 29, pp. 1191–1218.
86. Boman, E. G., and Wolf, M. M., (2007). *A nested dissection approach to sparse matrix partitioning for parallel computations*, Proc. in Applied Mathematics and Mechanics.
87. MUMPS 4.10.0 User's Guide, website: mumps.enseeiht.fr, last visit on Dec. 2013.
88. Amestoy, P. R., Duff, I. S., Koster, J., L'Excellent, J.-Y. (2001). *A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling.* SIMAX, 23(1), pp.15-41.
89. Amestoy, P.R. and Guermouche, A., L'Excellent, J.-Y., Pralet, S. (2006). *Hybrid scheduling for the parallel solution of linear systems.* Parallel Computing, 32(2), pp. 136-156.
90. Amestoy, P. R., Buttari, A., Duff, I. S., Guermouche, A.,L'Excellent, J.-Y., Uçar, B., (2011). *MUMPS*, Encyclopedia of Parallel Computing, pp. 1232-1238. Springer.
91. Amestoy, P., (2011). *MUMPS: a parallel sparse direct solver in Flux*. Presentation, 2011 Cedrat Flux Conference, Marseille.
92. Schulze, J., (2001). *Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods*, BIT 41 (4), pp. 800–841.
93. Karypis, G., Kumar, V., (1998). *METIS-a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices-Version 4.0*, University of Minnesota.
94. Pellegrini, F., (2001). SCOTCH 3.4 User's Guide, Technical Report RR 1264-01, LaBRI, Universitee Bordeaux I.
95. Cedrat Flux User's Guide, website: www.cedrat.com, last visit on Dec. 2013.
96. L'Excellent, J.-Y., Sid-Lakhdar, M., (2013). *Introduction of shared-memory parallelism in a distributed-memory multifrontal solver*, Research Report n.8227, Project-Team ROMA.

97. Sid-Lakhdar, M.W., L'Excellent, J.-Y., Vivien, F. (2013). *Exploitation of multicore architectures in the resolution of sparse linear systems by multofrontal methods*. PhD thesis at ENS-Lyon, France.
98. Tim Davis Matrix collection, website: www.cise.ufl.edu/research/sparse/matrices, last visit on Dec. 2013.
99. Grid-TLSE Matrix collection, website: gridtlse.enseeiht.fr:8080/websolve/index.jsp, last visit on Dec. 2013.
100. Operto, S., Virieux, J., Amestoy, P., L'Excellent, J.-Y., Giraud, L. and Ben Hadj Ali, H. (2007). *3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: a feasibility study*. Geophysics, 72(5), pp. 195-211.
101. Sourbier, F., Operto, S., Virieux, J., Amestoy, P., and L'Excellent, J.-Y. (2009). *FWT2D: a massively parallel program for frequency-domain full-waveform tomography of wide-aperture seismic data - part 2: numerical examples and scalability analysis*. Computer and Geosciences, 35(3), pp. 496-514.
102. Geist A. and Ng E. G. (1989). *Task scheduling for parallel sparse Cholesky factorization*. Int. J.Parallel Programming, 18 pp. 291-314.
103. PlaFRIM platform, Hiepacs project, Inria Bordeaux - Sud-Ouest, website: https://plafrim.bordeaux.inria.fr/doku.php, last visit on Dec. 2013.
104. Amestoy, P., Buttari, A., Joslin, G., L'Excellent, J.-Y., Sid-Lakhdar, M.W., Weisbecker, C., Forzan, M., Pozza, C., Perrin, R., and Pellissier, V. (2013). *Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation*, IEEE Trans. on Magnetics, Volume 50 , Issue 2, doi: 10.1109/TMAG.2013.2284024.
105. Dughiero, F., Forzan, M., Ciscato, D. and Giusto, F., (2011). *Multi-crystalline silicon ingots growth with an innovative induction heating directional solidification furnace*, Photovoltaic Specialists Conference (PVSC), 37th IEEE.
106. Lupi, S., Dughiero, F., and Forzan, M., (2006). *Modelling single- and double-frequency induction hardening of gear-wheels*, Proc. of the 5th Int. Symposium on Electromagnetic Processing of Materials, Sendai, Japan, pp. 473-8.
107. Dughiero, F., Forzan, M., Pozza, C., and Sieni, E. (2012). *A translational coupled electromagnetic and thermal innovative model for induction welding of tubes*, Magnetics, IEEE Trans. on, vol. 48, n.2.
108. Mifune, T., Iwashita, T., and Shimasaki, M. (2002). *A fast solver for FEM analyses using the parallelized algebraic multigrid method*. Magnetics, IEEE Transactions on, 38(2), pp. 369-372.
109. Bebendorf, M., (2008). *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*, (Lecture Notes in Computational Science and Engineering), Springer, 1 ed., 2008.
110. Borm, S., (2010). *Efficient Numerical Methods for Non-local Operators*. European Mathematical Society.

111. Chandrasekaran, S., Dewilde, P., Gu, M., and Somasunderam, N. (2010). *On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs*. SIAM, Journal on Matrix Analysis and Applications, 31(5), pp. 2261–2290.
112. Bertazzo, M., Bullo, M., Dughiero, F., Forzan, M., Zerbetto, M. (2013). *Experimental results of a 55 kW permanent magnet heater prototype*, HES-13 Heating by Electromagnetic Sources, pp. 377-384.
113. Di Barba, P., Forzan, M., Pozza, C., Sieni, E., (2012). *Optimal design of a pancake inductor for induction heating: a multiphysics and multiobjective approach*, Proc. CEFC-12, Oita, JP.
114. Di Barba, P., Dughiero, F., Lupi, S., Savini, A., (2003). *Optimal shape design of devices and systems for induction heating: methodologies and applications*, COMPEL, vol. 22, no. 1, pp. 111-122.
115. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., (2002). *A fast and elitist multiobjective genetic algorithm: NSGA-II*, IEEE Trans. on Evolutionary Computation, vol. 6(2), pp. 182–197.
116. Dughiero, F., Forzan, M., Pozza, C., Pastore, C., Zerbetto, M., and Barbati, M. (2013). *Coupled multiphysics circuital modelling of Quasi Resonant induction cooktops*, HES-13 Heating by Electromagnetic Sources, pp. 253-260.
117. Crisafulli, V., Pastore, C., (2012). *New control method to increase power regulation in a AC/AC quasi resonant converter for high efficiency induction cooker*, Proc. 3rd IEEE International Symposium on Power Electronics for Distributed Generation Systems - PEDG 2012, Aalborg, Denmark.
118. Schenk, O., Gartner, K., Fichtner, W., (2000). *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1), pp.158-176, 2000.
119. Schenk, O., Gartner, K., (2002). *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28, pp.187-197, 2002.
120. Duff, I. S., and Koster, J., (1999). *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices*. SIAM J. Matrix Analysis and Applications, 20(4), pp.889-901, 1999.
121. Li, X.S., Demmel, J.W., (1999). *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, Texas, March 22-34,1999.
122. Sonneveld, P., (1989). *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10, pp.36-52, 1989.