Parallel strategies for reduced number of processors applied for CFD simulations on workstations

Popescu, Cristian-Claudiu

2009

Popescu, C. -C. (2009). Parallel strategies for reduced number of processors applied for CFD simulations on workstations. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/42376>

<https://doi.org/10.32657/10356/42376>

Parallel strategies for reduced number of processors applied for CFD simulations on workstations

A final thesis submitted by
Popescu Cristian-Claudiu (MEng - G0703455K)

Supervisor Asst. Prof.
Dr. Roxana Vasilescu

**School of Mechanical and Aerospace Engineering
Nanyang Technological University
Singapore**

December 2009

Abstract

This work is aimed to exploit the computational capabilities of nowadays workstations by building algorithms parallelized for a limited number of processors. For an engineer that is working in aerodynamic design, the possibility of having efficient and whenever available computational resources is a considerable advantage. By using adaptive algorithms to numerical solution conform to workstation's hardware configuration, acceptable processing time and enough accurate solutions configuration can be obtained. The algorithms were tested on different hardware platforms and encouraging results were obtained.

Acknowledgement

I would like to address my thanks to my advisor and also my friend, Dr. Roxana Vasilescu, for her unbounded patience, for her inspired and valuable guidance who helped me throughout this master thesis.

I also would like to thank to my fellows from Nanyang Technological University: Chen Ling, Mitrea Catalin, Avram Andrei and Mihaela Carp for their supporting thoughts throughout this research project.

My last thoughts are going towards my family, to whom I would like to address my thanks for their acceptance to let me be here, far away from home.

Table of Contents

Chapter 1 - Introduction.....	1
1.1 Background.....	1
1.2 Motivation.....	4
1.3 Objective.....	5
Chapter 2 - Literature survey.....	6
2.1 Introduction.....	6
2.2 Parallel algorithms performance	6
2.3 Parallel algorithms balance	8
2.4 Parallel algorithms decomposition strategies.....	9
2.4.1 Domain Decomposition.....	10
2.4.1.1 Grid partitioning.....	10
2.4.1.2 Parallel algorithms decomposition strategy shift.....	13
2.4.2 Functional Decomposition	17
2.4.2.1 Matrix partitioning for direct inversion	18
2.5 Parallel algorithms environment	27
2.5.1 Distributed-Memory Programming Model.....	27
2.5.1.1 MPI (Message Passing Interface)	28
2.5.2 Shared-Memory Programming Model	28
2.5.2.1 Computational Environment: OpenMP	29
Chapter 3 - Parallelization Strategies	31
3.1 Introduction.....	31
3.2 Full-matrix Inversion Algorithm.....	32
3.2.1 Full-matrix Inversion Algorithm: OpenMP implementation	34
3.3 Full-matrix Block Inversion Algorithms	36
3.3.1 Four-blocks Inversion Algorithms	37
3.3.1.1 Dual-core Algorithms.....	37
3.3.1.2 Quad-core Algorithms.....	42
3.3.2 Full-matrix Block Inversion Algorithms: OpenMP implementation	44
3.4 Sparse-matrix Block Inversion Algorithms	45
3.4.1 Six-blocks Matrix Inversion Algorithm	45
3.4.2 General Block Inversion Algorithm	49
3.4.2.1 Nested Sparse-matrix Block Inversion Algorithm.....	54
3.4.2.2 Nested Sparse-matrix Block Inversion Algorithm: OpenMP implementation	57
Chapter 4 - Parallel Algorithms Validation Applications	60
4.1 Introduction.....	60
4.2 Matrix Inversion Tests	60
4.2.1 Dual-core algorithms	62
4.2.2 Quad-core Algorithm	65
4.2.3 Remarks.....	66
4.3 Laplace Problem	71
4.3.1 Discussion on the domain decomposition	72
4.3.2 Discussion about the computational times obtained	75
4.3.3 Remarks.....	76
4.4 Steady state application: Airfoil Simulation	77
4.4.1 Remarks.....	85
4.5 Inviscid Compressible Vortex Transport Application	85

4.5.1 Remarks.....	98
Chapter 5 - Conclusions and Future work.....	100
5.1 Conclusions.....	100
5.2 Future work.....	102
Chapter 6 - References.....	103

List of Variables

Notation	Description	Unit
$\cdot, \cdot, \cdot, \cdot$	Submatrices	
k	Exponents	
	Fielder vector	
x, b, v	Vectors	
L, U	Lower and upper triangular matrices	
Q	Orthogonal matrix	
R	Upper triangular matrix	
S_D	Schur complement	
I	Identity matrix	
A, B, C, D	Matrices	
a, b, c, d	Submatrices	
A^{-1}, M^{-1}	Matrix inverse	
$v_{i,j}$	Vertex of graph	
G	Graph	
$A_{i,j}, B_{i,j}, C_{i,j}, D_{i,j}$	Submatrices	
$O(n)$	Number of operations	
f	Frequency	Hz
E, F	Submatrices	
$L(G)$	Laplacian matrix of a graph	
P	Number of processors	
S	SpeedUp	
	The non-parallelizable part of a process	
$a_{i,j}, b_{i,j}, c_{i,j}, d_{i,j}$	Matrix elements	
p	Fraction of a program which is parallelizable	

List of Figures

Figure 1.1: Different disciplines embedded in CFD [01]	1
Figure 2.1: Examples of Speedups [10]	7
Figure 2.2: Example of domain decomposition (or data parallelism)	10
Figure 2.3: Unbalanced partitioning of the grid into four subdomains with an overlap region	11
Figure 2.4 : Balanced partitioning of the grid into four subdomains with an overlap region .	12
Figure 2.5: Example of unstructured mesh	12
Figure 2.6: Partitioning unstructured grid into four subdomains with an overlap region.....	13
Figure 2.7: Mesh for the rotor blades [41]	14
Figure 2.8: An example of optimization for grid partitioning [42]	14
Figure 2.9: Visualization of the vortex diffusion (courtesy of Dr. Vasilescu)	16
Figure 2.10: The shift of focus from domain decomposition to functional decomposition....	16
Figure 2.11: Example of functional decomposition (or functional parallelism)	17
Figure 2.12: Example of functional decomposition: Matrix partitioning for direct inversion (Hans Boltz, 1923)	17
Figure 2.13: Resulted matrix after applying the steps of Wang's algorithm for a tri-diagonal matrix of order 16 partitioned into 4 blocks of order 4	23
Figure 2.14: An example of mapping a band matrix into tri-diagonal block matrices.....	23
Figure 2.15: Different types of parallelism in assembly trees [75]	26
Figure 3.1: The computation of LU decomposition for a 4x4 matrix.....	33
Figure 3.2: The computation of LU decomposition for an NxN matrix	33
Figure 3.3: Logical scheme of LU decomposition [58]	35
Figure 3.4: Example of parallel DO loop implementation in the early developments of the present work	36
Figure 3.5: Example of matrix partitioning in four blocks	37
Figure 3.6: Parallel Data Diagram for Algorithm 1	39
Figure 3.7: Parallel Data Diagram for Algorithm 2	41
Figure 3.8 Parallel Data Diagram for Algorithm 3	43
Figure 3.9: An example of using the section construct for the Algorithm 2	45
Figure 3.10: 1-D structured grid and the associated implicit operator matrix for a 5-point stencil	46
Figure 3.11: Example of block penta-diagonal matrix	46

Figure 3.12: A general form for mapping a penta-diagonal matrix into block tri-diagonal matrix	47
Figure 3.13: Matrix inversion property	47
Figure 3.14: Parallel diagram of the six-block matrix inversion algorithm	48
Figure 3.15: Example of matrices covered by developed algorithms.....	49
Figure 3.16: Example of increasing the preserve of a sparse matrix	50
Figure 3.17: Example of calculating the semi-band length (lband) for a penta-diagonal matrix	50
Figure 3.18: Main stages of the general sparse matrix algorithm (Algorithm 4)	51
Figure 3.19: Parallel diagram of the general sparse algorithm (Algorithm 4)	52
Figure 3.20: 2-D structured grid and the associated implicit operator matrix for a 5-point stencil; the nonzero values are displayed as x	53
Figure 3.21: The block diagonal matrices contain many zero entries	54
Figure 3.22: Example of nested parallelization [11].....	55
Figure 3.23: Nested parallelism for the general sparse algorithm (Algorithm 5 nested).....	56
Figure 3.24: Example of parallel matrix multiplication for additional two processors used in the nested region	57
Figure 3.25: Example of enabling the nested parallelism	58
Figure 3.26: Example of nested parallelism implemented in the present work	58
Figure 3.27: Example of enabling the nested parallelism	58
Figure 4.1: CPU time differences on Workstations 1 & 2 for serial Gaussian elimination	62
Figure 4.2: CPU time differences on Workstations 1, 2 & 3 for the parallel code of Algorithm 1	63
Figure 4.3: CPU time differences on Workstations 1, 2 & 3 for the parallel code of Algorithm 2	64
Figure 4.4: CPU time differences on Workstations 1 & 2 for the parallel code of Algorithm 3	66
Figure 4.5: CPU time differences between the sequential runs of the parallel algorithms	67
Figure 4.6: CPU time differences between all parallel algorithms developed	68
Figure 4.7: Speedups of the Algorithm 1 & 2	69
Figure 4.8: CPU time differences between for Algorithm 2 & 3 in parallel execution	70
Figure 4.9: Speedups of the Algorithm 2 & 3	71
Figure 4.10: Double-block grid and single grid comparison for the first analytical function.	73

Figure 4.11: Double-block grid and single grid comparison for the second analytical function	74
Figure 4.12: Grids resolution errors.....	74
Figure 4.13: Computational times for single grid case and double-block grid case.....	75
Figure 4.14: Computational times for single grid case (general sparse algorithm) and double-block grid case	76
Figure 4.15: Mesh for NACA 0012, 243x80 cells.....	77
Figure 4.16: Pressure field for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$	78
Figure 4.17: Pressure coefficient for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$	79
Figure 4.18: Lift coefficient history for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$	80
Figure 4.19: Drag coefficient history for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$	80
Figure 4.20: Pressure field for NACA 0012, Mach=0.5 and $\alpha=3^\circ$	81
Figure 4.21: Pressure coefficient for NACA 0012, Mach=0.5 and $\alpha=3^\circ$	82
Figure 4.22: Lift coefficient history for NACA 0012, Mach=0.5 and $\alpha=3^\circ$	83
Figure 4.23: Drag coefficient history for NACA 0012, Mach=0.5 and $\alpha=3^\circ$	83
Figure 4.24: Lift coefficients as function of angle of attack for NACA 0012, Mach=0.3.....	84
Figure 4.25: Drag coefficients as function of angle of attack for NACA 0012, Mach=0.3....	84
Figure 4.26: Distance vortex plots for all algorithms including the iterative method and the LAPACK routine	87
Figure 4.27: Initial vortex location	89
Figure 4.28: Vortex transportation at 5 m for CFL=0.678.....	90
Figure 4.29: Vortex transportation at 5 m for CFL=1.356.....	92
Figure 4.30: Vortex transportation at 18 m for CFL=0.678.....	94
Figure 4.31: Vortex transportation at 18 m for CFL=1.356.....	96
Figure 4.32: Distance accuracy reported to CFL (5 meters)	97
Figure 4.33: Distance accuracy reported to CFL (18 meters)	98

List of Tables

Table 1: CPU times for serial matrix inversion based on Gaussian Elimination	61
Table 2: CPU time obtained for baseline Gaussian Elimination and for serial code of the Algorithm 1	62
Table 3: CPU time obtained on Workstation 1, 2 & 3 for parallel code of Algorithm 1	62
Table 4: CPU time differences for baseline Gaussian Elimination and for serial code of the Algorithm 2	63
Table 5: CPU time on Workstation 1, 2 & 3 for the parallel code of Algorithm 2	64
Table 6: CPU time differences for baseline Gaussian Elimination and for serial code of the Algorithm 3	65
Table 7: CPU time on Workstation 1 & 2 for the parallel code of Algorithm 3	65
Table 8: Full-matrix algorithms tested on the small mesh	88
Table 9: Sparse-matrix algorithms and iterative method tested on the small mesh	88
Table 10: Data input for vortex transport at 5 and 18 m	89
Table 11: Distance accuracy for 5 and 18 meters	97

“It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.”

John Von Neumann (ca. 1949)

Chapter 1 - Introduction

1.1 Background

The field of Computational Fluid Dynamics (CFD) has become certainly a domain that intersects not only the field of mathematics with fluid mechanics, but includes also a branch of computer science (Figure 1.1). Facing the continuous challenges of simulating the complex flow physics with high (or at least acceptable) level of accuracy, CFD field becomes increasingly sophisticated. The last decades, have witnessed a constant progress in the CFD research driven by the evolution of the computational resources. The new developments in the hardware industry and the diversification brought in the computer architectures led to a significant advance in parallel computing and multiprocessing.

Figure 1.1: Different disciplines embedded in CFD [01]

Nowadays, the parallel computing became the mostly used technique for reducing the computational cost of the large scale applications. As well, the parallel computing has developed into an essential technology for implementation of the CFD as a reliable tool for comprehensive investigations applied in aerodynamic and multidisciplinary design

optimization. However, the CFD development has still far to go, significant breakthroughs still being expected in order to bring CFD simulations to an acceptable level of accuracy and computational costs for daily engineering applications.

Critical problems in parallel CFD are generated by the complexity of the investigated configurations that often lead to a sophisticated meshing and flow solving algorithms. The main requirements in designing parallel algorithms aim to exploit efficiently the capabilities offered by the parallel computing machines and are related to data partition, communication across internal boundaries, workload balance and minimization of the overhead caused by both communication and computation. Parallel computing it has been used for many years, especially in the High Performance Computing (HPC) applications, but it became widely spread in the recent years due to development of the new multicores processors architectures, such as dual or quad core processors.

From software developer perspective, coding in parallel environment is more difficult than the usual sequential programming, because of the new class of potential software issues (such as task granularity [02], locality issues [03], etc). A significant challenge to improve the performance in parallel programming is the synchronization and communication between the different threads performed via Message Passing Interface (MPI) commands. While the MPI environment was developed mainly for distributed memory systems [04], in the case of parallel shared memory (SM) architectures, a new toolkit [05] emerged in October 1997 to leverage the parallel execution efficiency for this type of systems. This toolkit, known as Open MultiProcessing (OpenMP), is basically a set of compiler directives (known as pragmas) that implement programs for Shared Memory systems. OpenMP led to significant extensions of the programming languages like FORTRAN, C or C++ in order to account for shared memory parallelism. Working as an application program interface (API), OpenMP makes the parallel application programming much simpler by masking many of the details of thread communication and management behind an easier programming interface [06].

As the distributed memory computer cluster systems replaced large SMP (Symmetric MultiProcessing) systems in the 90', the end of OpenMP (Open Multi-Processing) era was soon anticipated [07]. Nowadays, with the emergence of the core nomenclature (2004) leading to a new age of shared memory computers, OpenMP programming is making a revival among HPC users. An interesting cartoon in the SIGACT (Special Interest Group on Algorithms and Computation Theory) *News Complexity Theory Column 09* showed a parallel computing lab on whose main door was written "*Closed due to bankruptcy*". By contrary, the recent years showed a shift in the parallel computing from the

advanced research labs towards usual engineering workplaces. The main reason of this shift is the increased availability of the multi processor workstations that now can be acquisitioned at affordable costs. It can be roughly say that HPC, which once was a luxury that few can afford in an engineering environment, through the availability of low cost computational resources becomes part of a daily life routine.

1.2 Motivation

For an engineer, the availability of fast and low cost computational resources such as desktops and workstations is a considerable advantage. The use of powerful servers or cluster networks is usually limited by the cost of equipments as well as the level of expertise required from users. In order to enhance the fidelity of computational simulations for engineering applications, a practical solution would be the transfer and adaptation of the advanced CFD research, which is usually carried on the supercomputers and clusters, towards the more accessible workstations. However, a direct portability between the different machines is not feasible, since this represents a reduction of the calculation capability that would be simply translated in significant reductions in the grid sizes and increment of the computational time. Therefore, in order to reach an advanced level of accuracy and fidelity of the simulations performed on the workstations, the development of the numerical algorithms and the derived codes has to be oriented to and benefit from the specificity of the parallel environment offered by these machines.

Following this line of thoughts, the motivation of the research presented in this thesis is fueled by the potential to achieve acceptable reductions in computational time while enhancing the simulations accuracy by developing and adapting the flow solver algorithms for specific computational equipment. Specifically, this work is carried on the workstations with dual- and quad- core processors. Since these hardware equipments utilize shared memory multiprocessing, the most suitable API choice for running parallel tasks is OpenMP [05].

The incentive to develop this thesis is provided by the range of applications to which the parallelized algorithms are addressed. The class of applications targeted here consists on improvements of vortex transport simulations. Typically, the simulations for vortex dominated flows are performed with high accuracy only using significantly large grid sized and consequently appealing to exhaustive computational resources. Therefore, the ability of performing these simulations on simple workstations can be a breakthrough and a final motivation of the present work.

1.3 Objective

Motivated by the use of low cost computational resources, the current thesis has as main objective the development of parallel algorithms suitable for daily engineering applications (simulations). In order to efficiently exploit the available workstations, a key feature of their development constitutes that the algorithms are oriented on the hardware configuration. This reveals an additional goal of this thesis which is adaptability of the programming code to hardware configuration. Practically, it means that the code is adapted to specific hardware format determined by the number of cores and processors. Adaptability is present as well at the level of numerical methods investigated which are chosen conform to the targeted applications.

Synthesizing, this thesis proposes to investigate a number of methods which can prove by conceptual approach the efficiency and the benefits of using algorithms tailored to applications specifications and hardware configurations.

Chapter 2 - Literature survey

2.1 Introduction

The overall goal of CFD is to numerically simulate the physical flows encountered in nature and to characterize its main properties. The estimations of the flow properties made through numerical methods aim to be as accurate as possible. Accurate simulations of the flows around a complete aircraft for design applications, simulations of the wake-vortex flows generated by a helicopter rotor system, new configurations of flying objects are just few critical examples of computational aerodynamics challenges. However, for any CFD algorithm there is a cost for accuracy which has to be matched by sufficient computational resources. Unfortunately, the development of computational resources is facing physical constraints which are preventing frequency scaling [08] (a technique used to ramp the processor frequency so as to achieve performance gains). In computer architecture development, the frequency scaling was the dominant force in processor performance increases from the 1980s till the end of 2004. Practically, the end of frequency scaling has caused a hardware industry focus shift to parallel computing in the form of multicores processors.

In computational fluid dynamics, parallel computing has been study for many years but interest in it has grown lately due to the complexity of modern problems which, in current multicores state, demands the application of the most effective parallel solution procedures. For this reason, the creation of high quality parallel algorithms could be an alternative which provide the hope of advancing further to reach the goals indicated above.

2.2 Parallel algorithms performance

Two of the most important measures for the parallel algorithms quality are the *speedup* and the *efficiency*. Ideally, a parallel algorithm will reach its full quality when three key points will be achieved: the code does not contain a sequential part, no communication between processors is needed and each processor has the same workload attributed. Of course, these are ideal points, impossible to be fulfilled but the focus of the parallel codes should be made in these directions.

The performance obtained by a parallel algorithm is estimated by the SpeedUp factor. At the first sight, the SpeedUp gain from the code parallelization should be linear;

meaning that doubling the number of processors should reduce two times the processing time. Nevertheless, the practice shows that most parallel algorithms do not have a linear SpeedUp and only few of them reach the optimal SpeedUp. Researchers in the parallel processing community use Amdahl's law [09] and Gustafson's law [10] for *a priori* estimations of the SpeedUps as indicators of the parallel program potential to reduce the execution time. Amdahl's law defines a restriction of performance (SpeedUp) for a parallelized program. The law is stating that the part of a code which cannot be parallelized will limit the overall speed available for parallelization as shown in the following equation:

$$S = \frac{1}{(1 - \beta) + \frac{\beta}{N}}; (2.1)$$

where β is the fraction of the program that is parallelizable, N is the number of processors available and S is the SpeedUp. As example, if the serial part of a program is 20% of the total processing time then it can get no more than five times SpeedUp, no matter how many processors are added. Similar examples can be deduced from the figure below.

Figure 2.1: Examples of Speedups [10]

Gustafson's law estimates the SpeedUp as:

$$S(N) = N - \alpha(N-1); (2.2)$$

where N is the number of processors, S is the SpeedUp, and α the non-parallelizable part of the process. Gustafson's law by its definition address a shortcoming of Amdahl's law by scaling the availability of computing power as the number of processors increases. Any code will normally consist of parallelizable portions and non-parallelizable portions, which are in fact sequential parts of the program and *a priori* estimations of the time reduction achieved via parallelization, can be determined by using these laws. Close related to the SpeedUp of parallel algorithm comes its efficiency. Basically the efficiency of a parallel algorithm is the SpeedUp reported at the number of processors.

The performance of a parallel code is influenced by its load balance, communication overhead, hardware architecture and software overhead. Each of the mentioned aspects that are affecting the code performance will be discussed by detailing the ones which are considered crucial for the current work development.

The performance is mostly affected by the speed of the slowest node. Therefore, an important concern is to make sure that each node is performing the same amount of work which basically means that the system is load balanced.

2.3 Parallel algorithms balance

Available literature about the designing of parallel algorithms discusses mainly two important and somehow antagonistic themes depending on context. The first issue is the need to partition the problem into tasks in such way as to create a balanced workload for all the available processors. The exact amount of work load for each processor depends on the computational problem and the parallel computer. As an example, if a computer has processors that are not equally fast, then the tasks sizes depend on the speeds of the different processors such so that the faster processors would deal with larger tasks than the slower processors. The second problem is the need to reduce the communication overhead which will lead to an increased overall runtime (especially when the number of processors is increased [11], [12], [13], [14]). The current work is carried out for reduced number of processors therefore the communication overhead will not constitute an important factor in the algorithms design.

Regarding the workload matter, it has to be mentioned that is not so important to obtain always a problem partition into an absolute minimum number of tasks, but to achieve fast solutions with quality required by the user. Load balancing is the determination of partitioning a problem into tasks such that to avoid the waiting time among processors. Load balance is typically achieved in two ways: *static load balancing* and *dynamic load balancing*.

a. *Static load balancing* – The load balancing is called static when the partitioning of the problem is used throughout the whole calculation. A balanced partition has to take into account both the numerical algorithm and the hardware configuration of the parallel machine. If the characteristics of one of these aspects change in the course of the calculations, an initially balanced subdivision can become unbalanced. Static load balancing is carried out before the computational phase of the problem.

b. *Dynamic load balancing* – The load balancing is called dynamic when the partitioning of the problem is changing during the calculation. The characteristics of the problem partitions can change due to different factors like e.g. the unavailability of a cluster processor. If an unbalanced task arises during the computations, one may decide to adapt the partition in order to obtain a balanced distribution of the tasks again. In that case it is dealt with dynamic load balancing that is considerably more complex than the static balancing. The most important reason is that a lot of CPU-time is consumed by the calculation of the new partition and the new initialization of the computation, which usually involves transfer of some tasks between processors. In general it is not known in advance for how long the new partition will remain balanced. Therefore one processor has to estimate the CPU-time needed to partition again the problem and reinitialize the computation. Another difficulty is that in order to reduce the costs of the rebalancing, another processor tries to find a new partition which resembles the old one. The application of some of the most efficient static balancing methods can change dramatically if the domain is changed only a little. However, the initial static distribution is usually built according to an optimization criterion which might not be satisfied again by the rebalanced partition. A detailed discussion on the performance of several algorithms in static and dynamic partitioning can be found in [15], [16] and [17].

The amount of workload assign to each processor is basically established by the decomposition stage. At this stage the problem is split into ‘pieces’ of work that are distributed in different tasks.

2.4 Parallel algorithms decomposition strategies

In the creation of a parallel program, this step is known as *partitioning* or *decomposition*. Basically, there are two methods for partitioning a problem into tasks to be executed in parallel:

- Domain Decomposition (Data Parallelism);
- Functional Decomposition (Functional Parallelism);

processors by satisfying two conditions. The first one is that the number of nodes assigned to each processor is approximately the same, leading to balanced work load among processors. The second requirement is that the number of contiguous elements assigned to distinct processors to be as low as possible, in order to avoid the communication overhead among processors.

Algorithm approaches is made differently if the meshes are structured or unstructured. In the followings it will be discussed the partition options for both structured and unstructured grid and some examples will be given for the overlapping subdomains. This type of subdomains (e.g. Chimera grids very often used in CFD simulations [27]) has been chosen, considering that these are more complex then the non-overlapping domains. Each subdomain created by partitioning is augmented by a layer of so-called *ghost nodes* that surrounds it. This layer is denoted as the *overlap region* but other names such as *halo* are also often used. The width of the overlap region is determined by the extent of the stencil operators involved.

Structured grids are generated in two- or three- dimensional geometries (with plane or curved surfaces). Structured grids have a regular topology where the neighborhood relations between all points are modeled within two- or three- dimensional array.

In the Figure 2.3 it is presented an imbalanced partitioning of a $(2k+1) \times (2k+1)$ structured grid. The subdomains created have overlapping regions.

Figure 2.3: Unbalanced partitioning of the grid into four subdomains with an overlap region

Interior nodes are represented by white circles; boundary nodes are symbolized by grey squares and ghost nodes as black circles/squares.

There is a difference in the size of the four subdomains which practically will result in some load imbalance, but this is almost inevitable for the grids typically employed in the complex geometric applications.

In the next figure, for a $(2k) \times (2k)$ grid, a balanced partitioning is presented in subdomains with an overlapping region.

Figure 2.4 : Balanced partitioning of the grid into four subdomains with an overlap region

Different than the structured grids, the unstructured grids are characterized by irregular connectivity (Figure 2.5).

Figure 2.5: Example of unstructured mesh

This raises the level of difficulty and increases the computational time required for solving the problem [33], [35], [36]. The algorithms for partitioning unstructured grid

calculations are as also very complex and high computational time consuming. However, the main advantage of defining the mesh nodes distributions according to the specificity of the problem makes the unstructured grids very popular for complex configurations [34]. Figure 2.6 presents an example of partitioning of an unstructured grid.

Figure 2.6: Partitioning unstructured grid into four subdomains with an overlap region

For a proper partitioning (e.g. figure above) grids are often modeled by a graph [28]-[29]. Then, the graph is divided into a given number of subgraphs with the goal of minimizing communication while the load is balanced.

Therefore, graph partitioning algorithms can be used to satisfy these conditions by:

1. Modeling the mesh with a graph;
2. By partitioning it into several subgraphs which represent mesh decomposition;

Even the commonly used programs such as METIS [30] or PARMETIS [31] perform alike the partitioning step for both types of grid, for unstructured grids the partitioning is more complex than for structured ones. Hence, any parallelization strategy is much more complicated than in structured case, where the neighborhood relationships are known at compile-time.

The grid partitioning algorithms are facing different issues which affect in fact the results of the simulation. Further important issues, that will make the change of focus for the current work in terms of decomposition strategy, will be approached.

2.4.1.2 Parallel algorithms decomposition strategy shift

The choice of mesh either structured or unstructured is related to the flow solver used. However, the general strategy for both cases is to decompose the (spatial) problem

domain into subdomains, for which certain portions of the computations can be performed by different processors simultaneously. The coupling of the subdomains is handled by the data exchange between adjacent subdomains via message passing. The message passing, updated at proper times during the numerical procedure, allow the flow solution inside each domain to be proceeded independently.

Like it was mentioned in the previous chapter, the geometrical complexity of the solution domain (see Figure 2.7) is an important aspect of the parallelization approach.

Figure 2.7: Mesh for the rotor blades [41]

If the domain is represented by a rectangle, it can be partitioned into a set of parallel portions. This gets of course to a scheme of one dimensional communication where basically the messages are forwarded to neighbors from left and right only. Illustrating a simple example, the Figure 2.8 presents the discussed case, where a structured mesh is split into three subdomains.

Figure 2.8: An example of optimization for grid partitioning [42]

The grid (left figure) is partitioned into three subgrids (right figure). Each subgrid is assigned to a subtask which will be assigned to a processor. Grid points in demarcation lines have their copies in its neighbor subtasks, represented by the circle grids. The darkest grids are shared by neighbor subtasks. The lightly dark grids are non-shared grids, and can be processed independently without communications with neighbor subtasks.

The solution domain parallelization for some complex geometry meshes needs a sophisticated communication among subdomains. In other words, the communication distance cannot be limited just to nearest neighbors, but any distance on the processor architecture topology is possible. Therefore this complex geometries grid requires a more complex communication pattern to assure a load balanced application. It also needs more complex configurations of message passing among neighboring blocks, which may reside on the same, on a neighboring, or on distant processor. The basic parallelization manner for these issues is the introduction of a new type of boundary condition [59]. Basically it is an inter-domain boundary condition which will be updated in the solution process through message passing by the neighboring domains.

Considering the examples described in the current and previous chapter it can be first concluded that the requirement of new types of boundaries conditions will lead to an increased communication between processors. Secondly it has been learned that the increased complexity of the grid partitioning algorithms, required especially for unstructured meshes, will lead to an increased overall computational time.

Apart from the increased time for grid generation and grid partitioning described above, this way of achieving parallel CFD leads also to a significant loss in the accuracy of simulations results for certain fields (e.g. rotor flow applications). As an example, the Figure 2.9 shows the significant noise and vortex diffusion introduced by the grid partitioning during the parallelization process.

Domain decomposition

*Discretization and linearization of
the gov. eq.*

Functional decomposition

2.4.2 Functional Decomposition

In the functional decomposition, the focus is on the executed computation and not on the data manipulated by the computation. Practically, the problem is split in accordance with the computation that must be performed; each resulted task then executes a part of the total work as it can be seen in the Figure 2.11.

Figure 2.11: Example of functional decomposition (or functional parallelism)

Functional decomposition is basically referring at breaking the problem into tasks which are distributed to processors for parallel computation. It will be assumed that the problem which needs to be decomposed into parallel tasks is the system of linear equations resulted after the discretization and linearization of the governing equations. Into numerical codes parallelization, ‘matrix partitioning’ is a usual method used for solving the linear systems of equations. Therefore by analogy, matrix partitioning for direct inversions represents in fact a functional decomposition (e.g. see Figure 2.12).

Figure 2.12: Example of functional decomposition: Matrix partitioning for direct inversion (Hans Boltz, 1923)

Linear algebra equations can be solved also by routines of software libraries like LAPACK [44] (Linear Algebra PACKage). In its parallel versions these libraries can cope efficiently with many linear systems but its implementations may raise some issues in certain contexts. An incursion in the literature [45] revealed that even though routines can be implemented quite efficiently for specific data layouts it is not practical to build numerical linear algebra on top of them. Though such specialized software like LAPACK often provides very efficient implementations, it is difficult to adopt this software to the specification of the problem [46] and also difficult to port it to new hardware and software releases on related architectures. Hence, although solution and algorithms for linear systems of equations is one of the oldest topics in numerical analysis there is still a need for developing new algorithms that are well-suited for nowadays high-performance computers [45].

According to the sparsity of the final matrix resulted after discretization and linearization of the governing equations, different parallel strategies are needed for an adequate solution. The algorithms developed in the present work aim at efficient parallel ways of partitioning the domain of a matrix.

What impact will have this approach of matrix domain partitioning on the parallel efficiency and simulation results dedicated now to a single large mesh? For specific cases the disappearance of the inter-domain boundaries will increase the parallel efficiency. A special argument valid for specific vortex dominated flows is that the simulation results are less exposed to the noise that affects the vorticity field.

Further, present work will be referred practically to the matrix partitioning methods. Specifically, parallel decomposition algorithms will be surveyed for linear algebraic systems resulted after applying the CFD commonly finite-volume discretization approach.

2.4.2.1 Matrix partitioning for direct inversion

The idea of matrix partitioning for direct inversion is simple and has a basic principle as foundation: The inversion of a set of small matrices is faster than the inversion of the single large matrix. Matrix partitioning is also known as matrix block-wise inversion, a technique that was reinvented several times and is due to Hans Bolz (1923), who developed it for the inversion of geodetic matrix, and Tadeusz Banachiewicz (1937), who generalized it and proved its correctness.

In research [48] a matrix A was considered that can be factorized using the Gauss algorithm without pivoting. For any decomposition in 2×2 blocks of the matrix, the lower and

upper triangular factors L and U of the Gauss factorization of the matrix have the following block structure:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \quad (2.3)$$

By identification, the following relations hold:

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ A_{21} &= L_{21}U_{11} \\ A_{12} &= L_{11}U_{12} \\ A_{22} &= L_{21}U_{12} + L_{22}U_{22} \end{aligned} \quad (2.4)$$

The first line from the above relations means that L_{11} and U_{11} are the Gauss factors of block A_{11} . The two following ones provide a means of computing the lower and upper diagonal blocks of L and U as:

$$\begin{aligned} L_{21} &= A_{21}U_{11}^{-1} \\ U_{12} &= L_{11}^{-1}A_{12} \end{aligned} \quad (2.5)$$

Using the equations (2.5) and the last line of relations (2.4) will get to:

$$L_{22}U_{22} = A_{22} - L_{21}U_{12} = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12} = A_{22} - A_{21}A_{11}^{-1}A_{12} \quad (2.6)$$

So L_{22} and U_{22} are the Gauss factors of matrix $A_{22} - A_{21}A_{11}^{-1}A_{12}$. In the followings it will be given a definition. Given a general 2x2 block matrix:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad (2.7)$$

which also can be factorized using the Gauss algorithm without pivoting, the matrix $A_{22} - A_{21}A_{11}^{-1}A_{12}$ is called the Schur complement matrix. The Schur complement matrix may be interpreted in several ways. Note n_1 and n_2 the dimensions of diagonal blocks A_{11} and A_{22} . Once the first n_1 steps of the Gauss algorithm have been performed, the lower diagonal that remains to be factorized is exactly the Schur complement matrix.

At this point the Gauss factorization may be rewritten:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & L_{22}U_{22} \end{pmatrix} \quad (2.8)$$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{pmatrix} \quad (2.9)$$

Basically means that once the first n_1 steps of the Gauss algorithm have been performed, the lower diagonal block that remains to be factorized is exactly the Schur complement matrix. Therefore the solution of the linear system allows the determination of the subvector x_2 by

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\Leftrightarrow \begin{cases} x_1 = A_{11}^{-1}(b_1 - A_{12}x_2) \\ (A_{22} - A_{21}A_{11}^{-1}A_{12})x_2 = b_2 - A_{21}A_{11}^{-1}b_1 \end{cases}$$

$$A^{-1} = \begin{pmatrix} \left(A_{11} - A_{12}A_{22}^{-1}A_{21} \right)^{-1} & \left(A_{21} - A_{22}A_{12}^{-1}A_{11} \right)^{-1} \\ \left(A_{12} - A_{11}A_{21}^{-1}A_{22} \right)^{-1} & \left(A_{22} - A_{21}A_{11}^{-1}A_{12} \right)^{-1} \end{pmatrix}$$

$$A^{-1} = \begin{pmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & S_A^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}S_A^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}S_A^{-1} \\ -S_A^{-1}A_{21}A_{11}^{-1} & S_A^{-1} \end{pmatrix}$$

$$A^{-1} = \begin{pmatrix} I & 0 \\ -A_{22}^{-1}A_{21} & I \end{pmatrix} \begin{pmatrix} S_D^{-1} & 0 \\ 0 & A_{22}^{-1} \end{pmatrix} \begin{pmatrix} I & -A_{12}A_{22}^{-1} \\ 0 & I \end{pmatrix}$$

$$= \begin{pmatrix} S_D^{-1} & -S_D^{-1}A_{12}A_{22}^{-1} \\ -A_{22}^{-1}A_{21}S_D^{-1} & A_{22}^{-1} + A_{22}^{-1}A_{21}S_D^{-1}A_{12}A_{22}^{-1} \end{pmatrix}$$

where S_D is the Schur complement of A_{22} in A . This formulation has the advantage of requiring only that either A_{11} or A_{22} and the corresponding Schur complement to be invertible. It also has the advantage of increased computational efficiency, by requiring only two inversions, six multiplications and three additive operations on submatrices. If $2 \leq \beta$ is the exponent such that $n \times n$ matrix multiplications requires $O(n^\beta)$ element operations, then this form of matrix inverse, applied recursively, gets to an $O(n^\beta)$ algorithm for matrix inversion. Parallelization conducted based on the Schur's complement technique described above has been studied also in the works [60], [61] and its benefits were used also implemented in the present work.

If instead of full matrices is dealt with banded matrices applications, the Schur's complement technique as described above is not efficient because it's not preserving the matrix sparsity. Banded systems occur very frequently, sometimes as a sub-problem within large computation. For example in CFD, methods for partial differential equations often solve many independent one-dimensional sub-problems, each of which has a banded set of equations. The simplicity of such systems and the fact that they arise frequently has given rise to many special methods.

The simplest and most common banded system is tri-diagonal, for example,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{43}x_3 + a_{44}x_4 + a_{45}x_5 &= b_4 \\ &\vdots \end{aligned}$$

If the variables $x_1, x_3, x_5 \dots$ are eliminated the resulting set of equations will be on the form

$$\begin{aligned} a_{22}^{(2)}x_2 + a_{24}^{(2)}x_4 &= b_2^{(2)} \\ a_{42}^{(2)}x_2 + a_{44}^{(2)}x_4 + a_{46}^{(2)}x_6 &= b_4^{(2)} \\ &\vdots \end{aligned}$$

Which is again a tri-diagonal system, but with half as many variables as before. This reduction can be applied repeatedly to yield a method called odd-even (or cyclic) reduction [72]. Although cyclic reduction requires twice as many floating-point operations as

straightforward elimination and creates fill-in (for example $a_{22}^{(2)}$, $a_{24}^{(2)}$, $a_{46}^{(2)}$ in the above equations), it is useful for two main reasons.

First, if it is rewritten the system ordering odd-numbered variables first, we have a system of the form

$$\begin{pmatrix} D_1 & B_1 \\ B_2 & D_2 \end{pmatrix} \begin{pmatrix} x_{odd} \\ x_{even} \end{pmatrix} = \begin{pmatrix} b_{odd} \\ b_{even} \end{pmatrix} \quad (2.13)$$

Where D_1 and D_2 are diagonal matrices, B_1 is lower bi-diagonal, and B_2 is upper bi-diagonal. It is apparent that, unlike the standard solution scheme which is inherently recursive, the cyclic reduction scheme vectorizes because within each block the equations are decoupled. During the first stage the vectors have length $n/2$ if the matrix has order n .

Second, if the matrix has the block tri-diagonal form

$$\begin{pmatrix} A & -B & & & \\ -B^T & A & -B & & \\ & -B^T & A & -B & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & \ddots & -B & A \end{pmatrix}$$

which often occurs when a partial differential equation on a rectangular region is discretized, the same operations are applied to the matrix (but not to the right-hand side) in each block row. The reduced sub-matrix has the form

$$\begin{pmatrix} A^{(2)} & -B^{(2)} & & & \\ -B^{(2)T} & A^{(2)} & -B^{(2)} & & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & \ddots & -B^{(2)} & A^{(2)} \end{pmatrix}$$

Where

$$A^{(2)} = A - 2B^T A^{-1} B$$

And

$$B^{(2)} = B A^{-1} B$$

Clearly $A^{(2)}$ and $B^{(2)}$ need to be evaluated only once and similar economy is possible during successive stages.

An alternative algorithm for a tri-diagonal set of equations was proposed by Wang in [73]. In his work he portioned the matrix after augmenting it into p blocks of k size. The algorithm was applied for a matrix of order 16. After applying a number of steps the tri-diagonal system is reduced at a diagonal system which is solved as a parallel or vector operation (see Figure 2.13).

$$\begin{bmatrix}
 x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x
 \end{bmatrix}
 \xrightarrow{(\dots)}
 \begin{bmatrix}
 x & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & x & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x
 \end{bmatrix}$$

Figure 2.13: Resulted matrix after applying the steps of Wang's algorithm for a tri-diagonal matrix of order 16 partitioned into 4 blocks of order 4

In the cases where the number of processors is lower in comparison with the matrix order, the scheme ensures that they are all fully used at every stage of the algorithm, and the total number of arithmetic operations is very similar to that required for odd-even reduction. If a number of small systems are to be solved simultaneously, they may be placed end-to-end to yield a large system for which adequate parallelism is available. As for the odd-even reduction, the algorithm can be unstable and it is hard to see how pivoting can be introduced if it is needed. An unsatisfactory feature of this algorithm is that the parallelism varies across the steps of the algorithm, giving k-fold instead of p-fold parallelism. This will not matter on vector hardware efficiently. However on a parallel computer, the communication or synchronization costs may be substantial.

In work [62] it is presented a parallel algorithm for computing the inverse for a specialized extension (extension in the maximum entropy sense [76]) of a banded matrix by using a modified Schur's complement technique. Any banded matrix can be considered in fact as a block tri-diagonal matrix; as example the matrix A from Figure 2.14 has a bandwidth of 5. By mapping the matrix into overlapping blocks of 2x2, it can be considered as a block tri-diagonal matrix.

$$A = \begin{bmatrix}
 \boxed{\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix}} & \boxed{\begin{matrix} a_{13} & 0 \\ a_{23} & a_{24} \end{matrix}} & \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} \\
 \boxed{\begin{matrix} a_{31} & a_{32} \\ 0 & a_{42} \end{matrix}} & \boxed{\begin{matrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{matrix}} & \begin{matrix} a_{35} & 0 \\ a_{45} & a_{46} \end{matrix} \\
 \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix} & \boxed{\begin{matrix} a_{53} & a_{54} \\ 0 & a_{64} \end{matrix}} & \begin{matrix} a_{55} & a_{56} \\ a_{65} & a_{66} \end{matrix}
 \end{bmatrix} = \begin{bmatrix}
 A_{11} & A_{12} & 0 \\
 A_{21} & A_{22} & A_{23} \\
 0 & A_{32} & A_{33}
 \end{bmatrix}$$

Figure 2.14: An example of mapping a band matrix into tri-diagonal block matrices

In this form any block tri-diagonal matrix can be expressed as:

$$A=L+D+U$$

Where D is the matrix that contains the main diagonal blocks of A, the matrix U is containing the blocks in the upper diagonal and matrix L is the matrix containing the blocks in the lower diagonal. If the blocks outside the three diagonals will be set to zero then the inverse of the block tri-diagonal matrix in general is full. However, in the case where the blocks outside the three diagonals are defined by extending the matrix as:

$$\tilde{A}=L+D+U+LD^{-1}L+UD^{-1}U+LD^{-1}LD^{-1}L+UD^{-1}UD^{-1}U+\dots$$

Then $\tilde{A}$ has block tri-diagonal inverse (has the same bandwidth as the matrix A and the matrix $\tilde{A}$ is basically the extension of A). To determine the inverse of $\tilde{A}$, the algorithm executes the inverse of each 2x2 resulted block by the next partial Schur operation. If this operation will be applied to the matrix from Figure 2.14 then it is obtained:

$$\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{22}^{-1} + A_{22}^{-1}A_{23}B^{-1}A_{32}A_{22}^{-1} & -A_{22}^{-1}A_{23}B^{-1} \\ -B^{-1}A_{32}A_{22}^{-1} & B^{-1} \end{bmatrix}$$

If there are others blocks of 2x2 in the banded matrix, all will be executed in the way illustrated above except the first one which will be executed in the following way:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11}^{-1} + A_{12}B^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}B^{-1} \\ -B^{-1}A_{21}A_{11}^{-1} & B^{-1} \end{bmatrix}$$

The final inverse of $\tilde{A}$ is executed by adding the block values in the overlapping 2x2 block sub-matrices. Since the block inverses can be executed in all the 2 by 2 blocks independently, these computations can be executed in parallel. For the second parallel part, the matrix inverses in the overlapping blocks are added to reveal the inverse of $\tilde{A}$. Thus the algorithm shows maximal parallelism and basically any band matrix inverse can be executed in the two parallel steps where each processor requires to execute the partial Schur operation (except one processor will execute the complete inverse by the Schur method in the first block) of 2x2 block with block size (b-1)/2x(b-1)/2, where b is the bandwidth of the banded matrix. Similar approaches of matrix partitioning have been implemented as well in the current work developed algorithms. The particular changes that were reflected in the algorithms were focused with their tailoring on the available hardware platforms. The inversion of each submatrix encountered in the developed algorithms was implemented the Gaussian elimination direct method, which is basically applied for full matrices. If the encountered block matrices are for example banded matrices then the main issue in the implementation of direct methods lies in preserving as much sparsity as possible in the matrix factors [63].

Many parallel algorithms for computing the direct solution of the sparse systems are relying on the frontal approach ([64], [65], [66], [67]).

Frontal methods have their origin in the solution of finite-element problems in structural mechanics, but they are not restricted to this application nor need the matrix to be symmetric and positive defined [75]. In a finite-element problem the matrix is a sum

$$A = \sum_l A^{[l]}$$

Where each $A^{[l]}$ has entries only in the principal submatrix corresponding to the variables in element l and represents the contributions from this element. It is normal to hold each $A^{[l]}$ in packed form as a small full matrix together with a list of the variables that are associated with element l , which identifies where the entries belong in A . The formation of the sum is called assembly and involves the elementary operation

$$a_{ij} = a_{ij} + a_{ij}^{[l]} \quad (2.14)$$

An entry is called fully summoned when all contributions of the above form have been summed.

It is evident that the basic operation of Gaussian elimination

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{ij}^{(k)} a_{kk}^{(k)-1} a_{kj}^{(k)} \quad (2.15)$$

May be performed before all the assemblies (2.14) are complete, provided only that the terms in the triple product (2.15) are fully summed (otherwise the wrong quantity will be subtracted). Each variable can be eliminated as soon as its row and column is fully summed, that is after its last occurrence in a matrix $A^{[l]}$ (a preliminary pass through the data may be needed to determine when this happens). If this is done, the elimination operations will be confined to the submatrix of rows and columns corresponding to variables that have not yet been assembled (these variables are called active variables). This permits all intermediate working to be performed in a full matrix whose size increases when a variable appears for the first time and decreases when one is eliminated. The pivotal order is determined from the order of the assembly. If the elements are ordered systematically from one end to the region of the other, the active variables form a front that moves along it. For this reason the full matrix which all arithmetic is performed is called the *frontal matrix* and the technique is called the *frontal method*. A major deficiency of the frontal solution scheme is the lack of scope for parallelism [75].

Further it is discussed a generalization of the frontal technique that is able to accommodate any ordering scheme based on the structure of the matrix. Because more than

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}$$

in parallel, and this is referred as tree parallelism. It is quite evident that generally, the parallelism of the tree can be exploited efficiently in the lower section of the tree than near the top node.

Despite the design considerations of the parallel algorithm, it has been learned that its performance is influenced or determined also by the machine architecture. The choice of parallel computing hardware often determines the choice of parallel programming environment.

2.5 Parallel algorithms environment

Parallel computing hardware can be classified by two main features: *control structures* and *memory interconnections*. For control structures perspective, one mostly used classification of computer architectures since 1966 is called Flynn's Taxonomy [52]. The four classes defined by Flynn are established considering the number of simultaneous instructions (or controls) and streams of data available in the architecture.

From the memory-processor interconnections perspective there are three predominant types of architectures [53]: *Distributed Memory*, *Shared Memory* and *Distributed-Shared Memory*.

Since there are several different classes of parallel computing hardware, it will be several distinct models which compose the parallel programming environment. Various realizations of both shared- and distributed-memory programming models have been defined and deployed. An ideal interface (API) for parallel programming should be expressive enough to permit the specification of many parallel algorithms and easy to use as so to lead to efficient programs. Moreover, the more transparent its implementation is, the easier the work of the programmer to understand how to obtain good performance. Unfortunately, there are trade-offs between these goals and parallel programming APIs are quite different in the features provided and in the manner and complexity of their implementation.

2.5.1 Distributed-Memory Programming Model

Generically referred as "the message passing interface" (MPI), the distributed-memory programming model assumes that the programs are executed by one or more processes, which has their own private address space [53]. Message-passing approaches to parallel programming must provide a means to initiate and control the active processes, along with operations for sending and receiving messages, and possibly for performing special

operations across data distributed among different processes. For this type of structures the most used paradigm for programming is MPI (Message Passing Interface).

2.5.1.1 MPI (Message Passing Interface)

The Message Passing Interface was created in 1990 by a team of researchers from the University of Illinois [04]. It provides a set of library routines for managing processes and exchanging messages between distributed memory systems where multiple processes execute independently and communicate data as needed by exchanging messages. MPI is widely used in high-end computing, where problems are so large that many computers are needed to compute them.

The creation of an MPI program can be a complicated task. The programmer must create the code that will be executed by each process, and this implies a burden work of reprogramming. The need to restructure the entire program does not allow for incremental parallelization as does OpenMP. It can be difficult to create a single program version that will run efficiently on many different systems, since the relative cost of communicating data and performing computations varies from one system to another and may suggest different approaches to extracting parallelism. It requires a lot of attention in order to avoid certain programming errors, particularly deadlock where two or more processes each wait in perpetuity for the other to send a message. From programming perspective, MPI represents in fact a set of library routines for programming languages like such as C/C++ or FORTRAN.

2.5.2 Shared-Memory Programming Model

As its name implies, shared-memory programming model assumes that the programs (or program blocks) will be executed on one or more processors that share a fraction or the entire available memory. Shared-memory programs are typically executed by multiple independent threads (execution states that are able to process an instruction stream). Within the shared memory programming framework the programmer job is to specify what should be executed in parallel. The programmer has available tools and methodologies that permit manipulation of the shared memory in an organized (directly-controlled) way, thereby preventing unwanted software bugs [53]. There is a diversity of a shared memory programming models and APIs, many of which are targeted towards different architectures, user groups, and applications domains. Very little standardization exists for shared memory

platforms and OpenMP [05] constitutes the most notable effort. OpenMP is basically a standard for enabling parallelization techniques on shared-memory (SM) multiprocessors platforms. Similar to MPI, specifications are available for the most used programming languages like C/C++ and FORTRAN.

The current work algorithms are developed for shared memory platforms (workstations) therefore the OpenMP paradigm will be discussed into more details.

2.5.2.1 Computational Environment: OpenMP

OpenMP [05] is a shared-memory application programming interface (API) whose features facilitate the shared-memory parallel programming. OpenMP was built to be suitable for implementation on a wide range of SM (Shared Memory) architectures platforms. The insertions of OpenMP features into a sequential program will allow many applications to take advantage from shared-memory parallel architectures – often with a minimal modification to the code [54]. OpenMP provides ‘notations’ that define and delimit the regions of an OpenMP program that should be executed in parallel, enabling in this way additional information on how this should be accomplished.

OpenMP directives allow the user to specify which instructions to be executed in parallel and how to distribute them among the available threads. An OpenMP directive is an instruction in a special format that is understood only by the OpenMP compiler. The API (Application Program Interface) does not have many different directives, but they are powerful enough to cover a variety of needs [05]. In the current work, there are implemented mostly just two directives, which proved to be enough for accomplishing the desired tasks.

The first step in creating an OpenMP program from a sequential one is to identify the parallelism it contains. Practically, this means seeking instructions, sequences of instructions, or even large portions of code that may be executed in a parallel way by different processors. Sometimes, this is an easy task, but many times the developer must reorganize portions of a code to obtain independent instruction sequences. It may even be necessary to replace an algorithm with an alternative one that accomplishes the same task but offers more exploitable parallelism. This can be a challenging problem. Fortunately, for this work the developed algorithms are created suitable for an easy OpenMP implementation.

The second step in generating an OpenMP program is to model, using OpenMP, the parallelism that has been recognized inside the numerical algorithm. A huge practical benefit of the OpenMP is that can be applied to incrementally create a parallel program from an existing sequential code. This powerful benefit is also a motivation for the current

implementations and interest for this approach. The developer can insert directives into a portion of the program and leave the rest in its sequential form. Once the resulting program version has been successfully compiled and tested, another portion of the code can be parallelized. Although creating an OpenMP program in this way can be easy, sometimes simply inserting directives is not enough. The resulting code may not deliver the expected results, and it may not be obvious always how to fix this issue [54].

An OpenMP directive has a specially formatted comment or “pragma” that generally applies to the executable line immediately following it in the program. A directive or OpenMP routine generally affects only those threads that encounter it. Specifically, the program may not branch into or out of blocks of code associated with directives.

Summarizing, OpenMP provides means for the user to:

- To create teams of threads for parallel execution;
- To specify how to share work among the members of a team;
- To declare both shared and private variables;
- To synchronize threads and enable them to perform certain operations exclusively, without interference by other threads.

In the following chapter the incursion into the current work methodology will be approached. Strategies adopted for the parallel algorithms will be presented along with the introduction of the most notable OpenMP directives which have been used in the present work.

Chapter 3 - Parallelization Strategies

3.1 Introduction

The present work is proposing to develop parallel algorithms suitable for solving the linear system of equations obtained after the discretization and linearization of the governing equations. Generally there are two main classes of methods for solving the linear systems of equations: *direct methods* and *iterative methods*.

Direct methods are based on the standard Gauss elimination technique and through a fixed number of computations it is obtained the exact arithmetic solution. These methods basically apply row operations to transform the initial system of equation into another one easier to be solved. Most direct methods are based on the elimination or on the matrix inversion so that the equations system is reduced to a state that is amenable to a solution by algebraic analysis. Compare to the direct methods, the iterative methods involve a succession of approximations for the solution. Therefore, these methods compute a solution after a succession of a variable number of steps. The number of steps depends on the predefined accuracy criteria that decides if the solution is convergent and reached the required accuracy [57]. Consequently, they allow a certain control of the accuracy of the final solution but the convergence is not necessary guaranteed. Compared to the direct methods, the iterative techniques offer higher flexibility in the manipulation of the matrix storage that ends up sometimes as a critical advantage from the memory requirements point of view. The iterative methods are generally built upon the classical Gauss-Seidel method but there is an intensive research in their development especially for large sparse matrices [56]. However, the convergence and the accumulative errors of these approximate methods can raise significant difficulties.

The main issue of the problem investigated in this research refers to the techniques of solving a large linear system by using block inversion algorithms. The algorithms that accomplish this utilize Gaussian elimination for computing the inverse of submatrices encountered in the parallel algorithms. In the followings sections, it will be described the developed algorithm for LU decomposition which constitutes the first step of the Gaussian elimination.

3.2 Full-matrix Inversion Algorithm

The direct method used in the present work for calculating the inverse of the matrix is using practically the classical two steps. First is the calculation of the LU decomposition. The second step is to apply the forward and backward substitution to the matrix resulted from performing the LU decomposition.

A first user-implemented level of parallelization developed in this research was to parallelize the steps of LU decomposition of a matrix. In the following figure it is showed a matrix A , whose LU decomposition can be written in following way:

$$\begin{matrix} & \text{L} & & \text{U} & & \text{A} \\ \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} & \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} & = & \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \end{matrix} \quad (3.1)$$

It has to be mentioned that the entries situated on the diagonal of matrix are $L_{ij}=1$ when $i=j$.

By performing the multiplication between the matrices LU it will be obtained:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{11}l_{21} & u_{12}l_{21} + u_{22} & u_{13}l_{21} + u_{23} & u_{14}l_{21} + u_{24} \\ u_{11}l_{31} & u_{12}l_{31} + u_{22}l_{32} & u_{13}l_{31} + u_{23}l_{32} + u_{33} & u_{14}l_{31} + u_{24}l_{32} + u_{34} \\ u_{11}l_{41} & u_{12}l_{41} + u_{22}l_{42} & u_{13}l_{41} + u_{23}l_{42} + u_{33}l_{43} & u_{14}l_{41} + u_{24}l_{42} + u_{34}l_{43} + u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad (3.2)$$

At the first glance, one can observe that in the resulting matrix exist inter-dependencies between columns and between rows. As an example, it is not possible to compute the elements of the second row without computing the elements of the first row. Similar example can be applied for the column case.

raised by the inter-dependencies between the rows of upper triangular matrix U . The available literature on parallel LU decomposition methods revealed similar results ([14], [18], [19]). Therefore, the described reasons conducted to a shift of focus from data parallelism to task parallelism in the matter of parallelizing strategies.

3.2.1 Full-matrix Inversion Algorithm: OpenMP implementation

The implementation of the LU parallelization was made basically by using OpenMP tool to parallelize the DO loop which computes the matrix L from the logical scheme presented in the Figure 3.3. The parallel potential of the logical scheme was discussed in the previous section.

Figure 3.3: Logical scheme of LU decomposition [58]

The OpenMP provides work-sharing directives which state how the computation in a structured code is to be distributed among the threads. Probably the most common work-sharing approach is to distribute the work in a DO loop among the threads in a team [54]. To accomplish this, the programmer inserts the appropriate directive immediately before each loop within a parallel region that is to be shared among threads (for example see Figure 3.4).

```

:
!$OMP PARALLEL PRIVATE(i,k,sum) LASTPRIVATE(j)
!$OMP DO
    DO i=j+1,m
        sum=localM(i,j)
        :
    ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
:

```

Figure 3.4: Example of parallel DO loop implementation in the early developments of the present work

When parallelizing a DO loop, the programmer has to consider the data dependencies that exist in the loop. Data inter-dependency is preventing the user from parallelizing a loop when a value that is written in one loop iteration is either written or read in iteration. The more complex is the loop, the harder it can be to decide whether such dependency exists. It should be kept in mind that the correct positioning of the OpenMP directives is the programmer/user's responsibility and therefore a fundamental understanding of the application is needed to decide whether there is or not a data dependency.

3.3 Full-matrix Block Inversion Algorithms

The main idea for building the matrix inversion such way to be parallelizable is to use the blocks-matrix decomposition. By using this technique, the matrix is firstly mapped into a block structure and than by using matrix inversion definition, different inversion algorithms are deducted.

An important fact to consider in the algorithms development is that their development started from scratch. The current work started from simple algorithms that enhance only fully populated matrices and advanced gradually to more complex algorithms that preserve the matrices sparsity.

3.3.1 Four-blocks Inversion Algorithms

The algorithms developed in the followings sub-sections are based on partitioning the matrix on four blocks (see Figure 3.5).

[illegible]

Figure 3.5: Example of matrix partitioning in four blocks

Basically, three main algorithms will be approached in the following sub-chapters; the first two are suitable for dual-core configurations while the third one is suitable for quad-core configurations.

3.3.1.1 Dual-core Algorithms

In this section, the proposed algorithms are addressed to double-core workstations. The construction of the first algorithms will be started by considering an $N \times N$ matrix A and its inverse A^{-1} . The matrix A will be split into four submatrices a, b, c, d of size $\frac{N}{2} \times \frac{N}{2}$ having elements $a_{ij}=A_{i, j}, b_{ij}=A_{i, j+N/2}, c_{ij}=A_{i+N/2, j}, d_{ij}=A_{i+N/2, j+N/2}$ with $1 \leq i, j \leq \frac{N}{2}$. Different notations but same splitting technique is used also for the inverse matrix A^{-1} .

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}; A^{-1} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \quad (3.3)$$

By multiplying matrix A with its inverse A^{-1} it is obtained the identity matrix I (property of inverse matrices).

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix} \quad (3.4)$$

After the multiplication of the matrices the following four relations are obtained:

$$\begin{cases} a\alpha + b\gamma = I \\ c\alpha + d\gamma = 0 \end{cases} \quad (3.5)$$

$$\begin{cases} c\beta + d\delta = I \\ a\beta + b\delta = 0 \end{cases} \quad (3.6)$$

The Algorithm 1 was obtained after performing the following operations to relation (3.5):

$$\begin{aligned} \begin{cases} a\alpha + b\gamma = I \\ c\alpha + d\gamma = 0 \end{cases} &\Leftrightarrow \begin{cases} bb^{-1}a\alpha + b\gamma = I \\ dd^{-1}c\alpha + d\gamma = 0 \end{cases} \Leftrightarrow \\ \begin{cases} b^{-1}a\alpha + \gamma = b^{-1} \\ b^{-1}a\alpha - d^{-1}c\alpha = b^{-1} \Rightarrow \alpha = -(b^{-1}a - d^{-1}c)^{-1}b^{-1} \\ \gamma = -d^{-1}c\alpha \end{cases} \end{aligned}$$

Similar operations were applied also to relation (3.6):

$$\begin{aligned} \begin{cases} c\beta + d\delta = I \\ a\beta + b\delta = 0 \end{cases} &\Leftrightarrow \begin{cases} dd^{-1}c\beta + d\delta = I \\ bb^{-1}a\beta + b\delta = 0 \end{cases} \Leftrightarrow \\ \begin{cases} d^{-1}c\beta + \delta = d^{-1} \\ d^{-1}c\beta - b^{-1}a\beta = d^{-1} \Rightarrow \beta = (d^{-1}c - b^{-1}a)^{-1}d^{-1} \\ \delta = -b^{-1}a\beta \end{cases} \end{aligned}$$

At this step the following notations will be made for an easy reference and observation of the method.

$$E = b^{-1}a \quad (3.7)$$

$$F = d^{-1}c \quad (3.8)$$

$$D = E - F \quad (3.9)$$

In Figure 3.6 is presented the diagram of the algorithm for two cores.

Figure 3.6: Parallel Data Diagram for Algorithm 1

The algorithm has five sequential phases during which two inversions and three multiplications are performed. The diagram of the algorithm shows an overall good load-balance but on the Sequential Phase 3 it is required a task which must be executed just by one of the threads; the other thread is idle until the computation of the inverse D^{-1} is finished.

The second algorithm proposed for two processors (cores) is fully load-balanced. This method is derived from the basic relations (3.5-3.6) similarly as the Algorithm 1. Using the same notations (3.7-3.9) and replacing $\gamma = -d^{-1}c\alpha$ in the first equation of the relations (3.5) it is obtained:

$$\begin{cases} a\alpha + b\gamma = I \Rightarrow a\alpha - bd^{-1}c\alpha = I \\ c\alpha + d\gamma = 0 \end{cases} \quad (3.10)$$

Consequently becomes equal to:

$$\alpha = (a - bF)^{-1} \quad (3.11)$$

From the relation (3.6) will be deducted by replacing from first equation with $-b^{-1}a\beta$ such that:

$$\begin{cases} c\beta + d\delta = I \Rightarrow c\beta - db^{-1}a\beta = I \\ a\beta + b\delta = 0 \end{cases} \quad (3.12)$$

$$\beta = (c - dE)^{-1} \quad (3.13)$$

An illustration of the distribution of the work load between the two threads is shown in Figure 3.7.

Figure 3.7: Parallel Data Diagram for Algorithm 2

The algorithm requires two blocks inversions and three blocks multiplications. The threads have assured totally independent work which guarantees a perfect load balancing.

3.3.1.2 Quad-core Algorithms

The two algorithms presented previously are addressed for computer systems with two processors (cores). The next algorithm is suitable for quad-cores processors being developed for four threads. Each thread solves one block matrix inversion defined by the Schur's complement.

Based on Schur matrix complement [49] each block matrix can be written as:

$$\begin{cases} \alpha = (a - bd^{-1}c)^{-1} \\ \beta = (c - db^{-1}a)^{-1} \\ \gamma = (b - ac^{-1}d)^{-1} \\ \delta = (d - ca^{-1}b)^{-1} \end{cases} \quad (3.14)$$

In the Figure 3.8 it is presented the parallel diagram for the Algorithm 3. The diagram contains four independent tasks during which are executed two inversions and two multiplications. Though, the total number of operations of the algorithm is much higher than for the previous presented algorithms.

All four threads of the Algorithm 3 are independent and each can be allocated to one core of a multi-processor computer platform. One drawback of this algorithm is that all the blocks matrices need to be invertible.

3.3.2 Full-matrix Block Inversion Algorithms: OpenMP implementation

The use of OpenMP library to parallelize directly the sequential codes, is sometimes a burden job which implies handling new classes of software bugs and thread synchronization problems [54]. So instead of using directly the current sequential codes, present work created ways of restructuring serial algorithms or developing new algorithms, in order to take full advantage of nowadays parallel shared memory computers. The implementations of current version of OpenMP are suitable for data parallelism than for task parallelism; therefore a direction that implements task parallelism should be approached with minimal use of OpenMP library. For implementing the full-matrix block inversion algorithms a reduced use of OpenMP was achieved by using just the basic constructs of the library like the *section* construct.

Depending on the type of the work performed by the algorithm blocks and the number of threads used, this construct might lead to a *load-balancing* problem. This occurs when threads have different amounts of work to do and thus take different amounts of time to complete. A result of load imbalance is that some threads may wait a long time at the next barrier in the program, which means that the hardware resources are not being exploited efficiently. The development of the new algorithms should consider this impediment and therefore have a balanced structure. This means that the amount of work for each thread should be distributed in equal amounts.

The *sections construct* is the easiest way to get different threads to carry out different work, since it permits to specify several different code regions, each of which will be executed by one of the threads. Each section must be structured block of code that is independent of the other sections [05]. At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed once. If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks. If there are fewer code blocks than the number of threads, the remaining threads will be idle during the computation. Note that the assignment of code blocks to threads is implementation-dependent.

An example of using the syntax of this construct implemented in the current work is showed in the figure below:

```

:
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
CALL invA(dA,dInv,m)
CALL matrixMul(dInv,cA,F,m)
CALL matrixMul(bA,F,dInv,m)
:
!$OMP SECTION
CALL invA(bA,bInv,m)
CALL matrixMul(bInv,aA,E,m)
CALL matrixMul(dA,E,bInv,m)
:
!$OMP END SECTIONS
!$OMP END PARALLEL
:

```

Figure 3.9: An example of using the section construct for the Algorithm 2

Although the *sections* construct is a general mechanism that can be used to get threads to perform different tasks independently, its most currently common use is probably to execute functions or subroutines calls in parallel.

3.4 Sparse-matrix Block Inversion Algorithms

The algorithms addressed so far are suitable for fully populated matrices. The algorithms develop in the next section are more complex because they consider the sparsity of the final matrix which is influenced by the grid structure.

3.4.1 Six-blocks Matrix Inversion Algorithm

The first developed sparse algorithm was created for penta-diagonal matrix systems. Penta-diagonal matrices are usually resulted from 5 stencil problems on 1-D structured grid (see Figure 3.10).

Figure 3.10: 1-D structured grid and the associated implicit operator matrix for a 5-point stencil

An example of penta-diagonal matrix is shown in the Figure 3.11.

$$\begin{bmatrix}
 X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & X & X & X & X & X
 \end{bmatrix}$$

Figure 3.11: Example of block penta-diagonal matrix

A general form to map the resulted matrix is presented in the Figure 3.12; where d , u and l are the diagonal, upper and lower diagonal matrices of the now called block tri-diagonal matrix.

$$\begin{bmatrix}
 x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x & x \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x
 \end{bmatrix}
 \rightarrow
 \begin{bmatrix}
 [d_1] & [u_1] & 0 & 0 \\
 [l_2] & [d_2] & [u_2] & 0 \\
 0 & [l_3] & [d_3] & [u_3] \\
 0 & 0 & [l_4] & [d_4]
 \end{bmatrix}$$

Figure 3.12: A general form for mapping a penta-diagonal matrix into block tri-diagonal matrix

After mapping the tri-diagonal matrix, the development of the sparse algorithm starts from the definition of the inverse matrices: $AA^{-1}=I$ (see figure below).

$$\begin{bmatrix}
 [d_1] & [u_1] & 0 & 0 \\
 [l_2] & [d_2] & [u_2] & 0 \\
 0 & [l_3] & [d_3] & [u_3] \\
 0 & 0 & [l_4] & [d_4]
 \end{bmatrix}
 \begin{bmatrix}
 [\alpha_1] & [\alpha_2] & [\alpha_3] & [\alpha_4] \\
 [\beta_1] & [\beta_2] & [\beta_3] & [\beta_4] \\
 [\gamma_1] & [\gamma_2] & [\gamma_3] & [\gamma_4] \\
 [\delta_1] & [\delta_2] & [\delta_3] & [\delta_4]
 \end{bmatrix}
 =
 \begin{bmatrix}
 [I] & 0 & 0 & 0 \\
 0 & [I] & 0 & 0 \\
 0 & 0 & [I] & 0 \\
 0 & 0 & 0 & [I]
 \end{bmatrix}$$

Figure 3.13: Matrix inversion property

After using similar techniques with the ones described for the previous three algorithms, the matrix inverse is computed in a parallel algorithm form. The resulted algorithm is presented in the following parallel diagram.

Figure 3.14: Parallel diagram of the six-block matrix inversion algorithm

The algorithm has 15 sequential phases during which are performed 3 matrix inversions and 12 matrix multiplications. The algorithm has two tasks which are fully balanced with the

specification that at the end of Sequential Phase 4 a synchronization barrier is needed to avoid possible race conditions.

3.4.2 General Block Inversion Algorithm

In retrospect, the current work methodology started from developing block matrix inversion algorithms applied for full matrices and ended so far with creating block matrix inversion algorithms for sparse matrices (see figure below).

$$\left[\begin{array}{cccccccccccccccc} x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \end{array} \right] \rightarrow \begin{bmatrix} [a] & [b] \\ [c] & [d] \end{bmatrix}$$

$$\left[\begin{array}{cccccccccccccccc} x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{x} & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{cccc} [d_1] & [u_1] & 0 & 0 \\ [l_2] & [d_2] & [u_2] & 0 \\ 0 & [l_3] & [d_3] & [u_3] \\ 0 & 0 & [l_4] & [d_4] \end{array} \right]$$

Figure 3.15: Example of matrices covered by developed algorithms

In the above figure, one can observe that the upper and lower block matrices still contain many zero entries. In other words, the sparsity of the matrix is still not fully preserved. One example of what can be done in order to preserve more the matrix sparsity is to increase the number of blocks (see Figure 3.16).

Figure 3.16: Example of increasing the preserve of a sparse matrix

By induction with the previous one, a similar algorithm can be developed for 8 blocks but at this point it is desirable to create a general sparse algorithm that can split the matrix in an optimum number of blocks requisite by the matrix sparsity.

By deduction, the number of blocks (mb) is given by the following formula:

$$mb = \frac{\text{number of rows'columns } (n)}{\text{dimension of the block } (r)}$$

As example in Figure 3.16 the left-matrix has 16 rows/columns (which represent n) and it is split in 2 dimensional blocks (which represent r). Therefore the number of blocks (mb) is 8 per row/column. An optimum number for the dimension of the block r is chosen by taking into consideration the semi-band length ($lband$) of the main matrix; the relation between these two terms is $lband \leq r$. The length of the semi-band is calculated for the upper or the lower side of the matrix by considering the distance between the diagonal and the furthestmost nonzero value found either on the row or column. The example in the figure below illustrates how the length of the band was calculated for the below matrix example:

$lband = 3-1$

		←	→			
		1	2	3	4	5
1	x	x	x	0	0	
2	x	x	x	x	0	
3	x	x	x	x	x	
4	0	x	x	x	x	
5	0	0	x	x	x	

Figure 3.17: Example of calculating the semi-band length ($lband$) for a penta-diagonal matrix

Therefore, the general sparse matrix algorithm should split the matrix in the minimum number of blocks in order to preserve its sparsity as much as possible. If the sparse algorithm

from the sub-chapter 3.4 is examined, it will be noticed that there are 3 main stages which lead to its development: *elimination*, *midinversion* and *substitution*.

The *elimination* stage simplifies the blocks matrix by eliminating rows and columns until a 2x2 matrix system is reached; the stage starts from the both matrix corners situated at the ends of the main diagonal and ends at its middle. Basically, the *elimination* stage is equivalent with a two-sided Gaussian elimination. Practically, the factorization resumes at starting simultaneously Gaussian elimination from top-down and from bottom-up of the matrix in cause. The next stage, *midinversion* basically solves the remaining system which is formed by a 2x2 matrix. The upper diagonal and the bottom diagonal are in the two ends of the main matrix. With the values obtained the *substitution* stage gets the inverted matrix by applying a backward substitution which starts from the middle-up and middle-bottom towards the ends of the matrix. The following figure shows generally the 3 main stages of the general sparse algorithm. This general algorithm will be referred as Algorithm 4.

Figure 3.18: Main stages of the general sparse matrix algorithm (Algorithm 4)

Operations performed on the matrix during each stage can be grouped in 2 independent tasks which can be computed in parallel. Each of the tasks is focusing on a half of the main matrix, starting with the described corners and ending on the middle of the main diagonal. The independent operations performed on each stage stipulate the source main of parallelism for the general sparse algorithm. In the Figure 3.19 is presented the main parallel diagram of the Algorithm 4.

Figure 3.19: Parallel diagram of the general sparse algorithm (Algorithm 4)

Where the mb constitutes the number of blocks in which the column/row is split and the nb represent half of its value: $nb = \frac{mb}{2}$.

Compared to the Algorithm 1, 2 and 3, the Algorithm 4 basically splits the matrix into a maximum acceptable number of blocks by taking into account the matrix band. The previous algorithms didn't take into account the matrix band, they just split the matrix into a predefined number of blocks (specifically four). Algorithm 4 adds more flexibility and computes automatically the number of necessary blocks for an optimal split.

So far, it was discussed about the case of a 5 stencil problem on 1-D structured grid and it was presented an ingenious way of creating potential sources of parallelism that can be exploited in problem algorithm.

The general sparse algorithm can be as well applied on a 2-D structured grid for discretized partial differential equations (see Figure 3.20), but its performance will not be fully achieved. To understand why, it is needed to examine the resulted matrix for a 2-D structured grid (see Figure 3.20).

Figure 3.20: 2-D structured grid and the associated implicit operator matrix for a 5-point stencil; the nonzero values are displayed as x

By applying the general sparse algorithm in a similar way like was done for the pentadiagonal matrix, one can notice that the block matrices will contain many zero entries which will cost additional computational resources (see Figure 3.21).

$$\left[\begin{array}{ccc|ccc|ccc} x & x & & x & & & \dots & & \\ x & x & x & & x & & & & \\ & x & x & x & & & & & \\ & & x & x & x & & & & \\ & & & x & x & x & & & \\ & & & & x & x & x & & \\ & & & & & x & x & & \\ x & & & x & x & & x & & \\ & x & & & x & x & & x & \\ & & x & & & x & x & & \\ & & & x & & & x & x & \\ & & & & x & x & x & & \\ & & & & & x & x & x & \\ & & & & & & x & x & x \\ & & & & & & & x & x \\ & & & & & & & & x \\ & & & & & & & & \ddots \end{array} \right] \rightarrow \begin{bmatrix} [d_1] & [u_1] & 0 \\ [l_2] & [d_2] & [u_2] \end{bmatrix}$$

Figure 3.21: The block diagonal matrices contain many zero entries

Therefore in order to consider the zero entries of the block matrices, the algorithm will be improved by enhancing its sparsity. What was done in fact was to define each block by considering its band length (*iband*). In this way, the allocation of the block matrices is made only for the non-zero entries. The 3 main stages of the algorithm were not modified; the blocks definition affected the multiplications and inversions algorithms which took into account the new blocks lengths. This new algorithm was called Algorithm 5; its scheme is not presented because it has similar structure with the Algorithm 4.

So far, the developed algorithms were mainly developed for two tasks; additional processors which are available on different hardware platforms will be used dynamically for nested parallelism. In the next section it will be discussed about the implementation of nested parallelism in the general sparse algorithm.

3.4.2.1 Nested Sparse-matrix Block Inversion Algorithm

So far, the developed algorithms were developed for reduced number of processors mainly 2 with the exception of the algorithm 3 which was developed for 4 processors. But this work, like it was discussed in the chapter 3, is proposing to make use of the additional processors which are available on different workstations. Therefore, the latest algorithms implemented in the current work have embedded modules of nested parallelization. Basically, nested parallelism will allow that additional processors available for different platforms to be used efficiently in the subroutines of the main program. An example of nested parallelism is described in the Figure 3.22.

Figure 3.22: Example of nested parallelization [11]

It can be observed that the program started the parallelization with a team of three threads and at the nested level each processor created its own team of three sub-threads. After the computation of the nested region, the program returns to its previous state.

A nested parallelization was implemented on every stage of the last version of general sparse algorithm. Considering this implementation the general sparse algorithm will be referred as Algorithm 5 nested. The nested parallelization will be active only on the platforms with number of processors larger than two. First a general scheme of the nested implementation will be presented and then by giving some examples, more details regarding the implementation of nested parallelism will be given.

In the Figure 3.23 it is presented the location of the nested parallel regions on the diagram of the general sparse algorithm.

Figure 3.23: Nested parallelism for the general sparse algorithm (Algorithm 5 nested)

Basically, the nested modules are implemented at the level of each stage on every parallel task. The number of nested threads is dependent on the number of additional processors available; in this way will be avoided the creation of virtual threads which will compromise the algorithm performance. The nested implementations affect only fractions of the algorithm's stages. Practically, the nested modules are concerned with the parallelization of the submatrices operations (multiplications and inversions).

The strategies chosen for the operations parallelization approach an explicit implementation. To exemplify the technique it will be used as example the block matrices multiplications (see Figure 3.24).

$$\begin{bmatrix} x & \cdots & \cdots & x \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ x & \cdots & \cdots & x \end{bmatrix} \times \begin{bmatrix} x & \cdots & \cdots & x \\ x & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ x & \cdots & \cdots & x \end{bmatrix} = \begin{bmatrix} x & \cdots & \cdots & x \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ x & \cdots & \cdots & x \end{bmatrix}$$

A **B** **C**

Figure 3.24: Example of parallel matrix multiplication for additional two processors used in the nested region

In fact, the procedure requires dividing the rows of matrix A at the number of additional processors available. The resulted groups of rows are then multiplied independently with the matrix B. The same technique is applied also for matrix-vector or vector matrix multiplications. For the block inversions was used the previous developed algorithms for full matrices. As example, if the number of processors is 2 then for the block inversion Algorithm 1 or 2 can be used. If the number of additional processors is 4 then the Algorithm 4 can be used. Usually the nowadays workstation systems come in configuration of 2, 4 and 8 processors. Similar algorithms development can be used in a recurrent parallelization if some workstations have more than 8 processors.

3.4.2.2 Nested Sparse-matrix Block Inversion Algorithm: OpenMP implementation

The OpenMP standard and especially its latest version (OpenMP v3.0 released on May 2008) provides several means which help the programmer to interact with the execution environment, either to obtain information from it or to influence the execution of a program. A research conducted onto the exploitation of nested parallelization capabilities of OpenMP environment cannot be made without the interaction of program execution environment. The OpenMP standard defines internal control variables which governs the behavior of a program at run time in important ways. These variables cannot be accessed or modified directly at the application level; however they can be queried and modified through OpenMP functions and environment variables. The *Nest-var* environment variables controls whether nested parallelism is enabled for future parallel regions.

This environment variable can be set either true or false prior to execution by giving the OMP_NESTED environment variable the corresponding value. As specified, a runtime

library routine enables the programmer to adjust the setting of *nest-var* at run time, possibly overriding the value of the environment variable. Its syntax is *omp_set_nested*, and it applies to the thread that executes it; in other words, if this thread encounters a parallel construct, then that region will become active as long as the implementation can support such nesting. As example, the corresponding FORTRAN syntax used in current work for enabling the nested parallelism is:

```
CALL OMP_SET_NESTED( .TRUE. )
```

Figure 3.25: Example of enabling the nested parallelism

The *omp_get_nested* routine, whose result is a logical value in FORTRAN, returns the current setting of the *nest-var* variable for the thread that calls it: *true* if nesting is enabled for that thread and otherwise *false*.

Like it was mentioned earlier, the latest version of OpenMP comes in handy to suit the current needs in order to implement an efficient nested parallelization. The programmer has an enhanced control over the nested regions by using the improved support of OpenMP paradigm. The newly library routines help the programmer to determine the depth of nesting. If for example it is wanted to know the depth of nesting for the case shown in the Figure 3.26

Figure 3.26: Example of nested parallelism implemented in the present work

Then the following FORTRAN syntax will be implemented:

```
CALL OMP_GET_LEVEL( level )
```

Figure 3.27: Example of enabling the nested parallelism

The value returned by this subroutine for the variable 'level' will be '1', meaning that the current level is the first level of nested parallelism. This function is especially helpful when

there are multiple levels of parallelism because it gives to programmer the scheme of overall nested implementation.

The described features plus many others make from the latest version of OpenMP a powerful tool that is easing the work of programmer and ensure a better control of the nested parallelism.

Once the algorithms have been devised, they have to prove their correctness for different possible problems. In the following chapter, all the present work parallel algorithms will be validated on different applications.

Chapter 4 - Parallel Algorithms Validation Applications

4.1 Introduction

The algorithms were implemented using the FORTRAN programming language on the *Intel FORTRAN Visual Compiler* development software platform. The hardware platforms used for tests are three workstations. The first one is named '*Workstation 1*' and has an Intel Core 2 Duo processor at 2.4GHz. The second has two Intel Xeon Quad Core processors at 3.4 GHz and is named '*Workstation 2*'. The third workstation denoted as '*Workstation 3*' has two Intel Xeon Core Duo processors at 3.2 GHz. The parallel computational times (*CPU times*) obtained by the algorithms are measured per core (thread) and basically are expressing the wall clock times in seconds.

In the following sub-sections, the algorithms will be validated on different applications like inverting full matrices of random values, solving Laplace problems, simulating airfoils profiles and transporting vortexes on different locations of the physical domain.

4.2 Matrix Inversion Tests

The first three algorithms developed in this work do not consider the sparsity of the matrix. Therefore, it was considered that the suitable validation test for these algorithms will constitute a randomly values generated full-rank square matrix which can be factorized by using the Gauss algorithm without pivoting. The matrix dimension varies with different size steps from 250, 500 or 1000 to a maximum dimension of 5000. The running tests represent conceptual evaluations of the parallel efficiency of the algorithms used for matrix inversion. Before the actual tests of the first three parallel algorithms, it was tested the serial version of the Gaussian elimination code on the Workstations 1 and 2. The Table 1 shows the results obtained.

<i>Square matrix dimension (n)</i>	<i>CPU time Workstation 1 T_1 [sec]</i>	<i>CPU time Workstation 2 T_2 [sec]</i>	<i>Time reduction per iteration $\frac{T_1 - T_2}{T_1} (\%)$</i>
500	0.7	0.5	26
1000	8.6	6.4	26
1500	38.1	34.5	9
2000	84.1	76.2	9
2500	196.6	181.9	8
3000	319.7	314.1	2
3500	588.9	544.1	8
4000	833.0	818.6	2
4500	1322.5	1280.2	3
5000	1701.0	1678.0	1

Table 1: CPU times for serial matrix inversion based on Gaussian Elimination

All the workstations tested showed very similar times varying with the matrix size. The trends illustrating the CPU time increase with the matrix size (Figure 4.1) show very close (almost identical) behavior for both workstations. The variations in terms of few seconds are not fully conclusive, since independent tasks related to the operating system can occur during the runs. Since the workstation 2 has a better processor, it is expected to be faster, but the gain in CPU less than 5% is marginal. The Gaussian elimination decomposition algorithm coded in serial will be used as the baseline test for benchmarking the parallel algorithms developed further. Consequently, the best CPU time obtained for Gaussian elimination decomposition algorithm will be considered the reference time for all the future comparisons.

Figure 4.1: CPU time differences on Workstations 1 & 2 for serial Gaussian elimination

4.2.1 Dual-core algorithms

First the Algorithm 1 was tested on Workstation 3 in the sequential mode with the parallel OpenMP directives disabled. The CPU times obtained are showed in the following table.

Square matrix dimension (n)	CPU time for baseline Gaussian Elimination [sec]	CPU time for serial code of the Algorithm 1 [sec]
1000	6.1	4.2
2000	68.9	48.4
3000	257.5	204.4
4000	683.1	499.8
5000	1359.0	1111.9

Table 2: CPU time obtained for baseline Gaussian Elimination and for serial code of the Algorithm 1

One can observe that even though the Algorithm 1 was executed in serial, it obtained better computational times than the base line algorithm. This result was expected since the number of operations for the Algorithms 1executed in sequential is less than the base line algorithm.

Second the Algorithm 1 was tested on all three available workstations with OpenMP directives enabled. The results obtained are showed in the table below.

Square matrix dimension (n)	Baseline algorithm time T_b [sec]	W.1 time T_1 [sec]	W.1 time reduction per iteration $\frac{T_b - T_1}{T_b}$ (%)	W.2 time T_2 [sec]	W.2 time reduction per iteration $\frac{T_b - T_2}{T_b}$ (%)	W.3 time T_3 [sec]	W.3 time reduction per iteration $\frac{T_b - T_3}{T_b}$ (%)
1000	6.1	3.5	42	2.5	59	2.4	61
2000	68.9	38.9	44	32.1	53	29.4	57
3000	257.5	153.8	40	132.8	48	128.0	50
4000	683.1	366.4	46	361.1	47	293.0	57
5000	1359.0	808.9	40	769.2	43	680.3	50

Table 3: CPU time obtained on Workstation 1, 2 & 3 for parallel code of Algorithm 1

The Algorithm 1 average performance was around 40% better than the base algorithm. The computational time differences obtained are relative to workstations computational power. In Figure 4.2 it is presented the graph with the computational time differences for the Algorithm 1 on Workstation 1, 2 and 3. It can be observed that the Workstation 3 represented by the green line has the best results with a significant difference between the other two workstations.

Figure 4.2: CPU time differences on Workstations 1, 2 & 3 for the parallel code of Algorithm 1

Similar to previous algorithm, the Algorithm 2 has been tested also on Workstation 3 in sequential mode with parallel OpenMP directives disabled. The results are shown in the Table 4.

Square matrix dimension (n)	CPU time for baseline Gaussian Elimination [sec]	CPU time for serial code of the Algorithm 2 [sec]
1000	6.1	4.8
2000	68.9	54.8
3000	259.3	235.7
4000	685.5	570.2
5000	1365.4	1280.5

Table 4: CPU time differences for baseline Gaussian Elimination and for serial code of the Algorithm 2

The computational times obtained by Algorithm 2 in sequential execution are close to the ones obtained by the base line algorithm. Compare to the Algorithm 1, the Algorithm 2 has more operations to compute in sequential execution and hence its increased CPU time.

The next table shows the results obtained after running the Algorithm 2 on all three available workstations with OpenMP directives enabled.

Square matrix dimension (n)	Baseline algorithm time $T_b[\text{sec}]$	W.1 time $T_1[\text{sec}]$	W.1 time reduction per iteration $\frac{T_b - T_1}{T_b} (\%)$	W.2 time $T_2[\text{sec}]$	W.2 time reduction per iteration $\frac{T_b - T_2}{T_b} (\%)$	W.3 time $T_3[\text{sec}]$	W.3 time reduction per iteration $\frac{T_b - T_3}{T_b} (\%)$
1000	6.1	3.6	41	2.5	59	2.4	61
2000	68.9	39.9	42	36.7	47	30.7	56
3000	259.3	156.1	40	144.1	44	131.0	49
4000	685.5	405.8	41	359.5	48	301.1	56
5000	1365.4	822.0	40	776.9	43	686.9	50

Table 5: CPU time on Workstation 1, 2 & 3 for the parallel code of Algorithm 2

The time differences for Algorithm 2 on all the available workstations are showed in the graph from Figure 4.3.

Figure 4.3: CPU time differences on Workstations 1, 2 & 3 for the parallel code of Algorithm 2

As it can be observed from the above graphic, Workstation 3 obtained best computational times due to its increased computational power.

4.2.2 Quad-core Algorithm

In the Table 6 are presented the results obtained for the Algorithm 3 only for the serial cases with no OpenMP directives enabled.

Square matrix dimension (n)	CPU time for baseline Gaussian Elimination [sec]	CPU time for serial code of the Algorithm 3 [sec]
1000	6.1	7.5
2000	68.9	88.8
3000	257.7	394.7
4000	683.2	942.0
5000	1365.4	1470.5

Table 6: CPU time differences for baseline Gaussian Elimination and for serial code of the Algorithm 3

In serial execution the Algorithm 3 has more operations than the base line algorithm and because of this fact its computational time is more increased.

In the Table 7 are presented the computational times obtained for all available workstations for Algorithm 3.

Square matrix dimension (n)	Baseline algorithm time T_b [sec]	W.1 time T_1 [sec]	W.1 time reduction per iteration $\frac{T_b - T_1}{T_b}$ (%)	W.2 time T_2 [sec]	W.2 time reduction per iteration $\frac{T_b - T_2}{T_b}$ (%)	W.3 time T_3 [sec]	W.3 time reduction per iteration $\frac{T_b - T_3}{T_b}$ (%)
1000	6.1	6.3	-3	2.1	66	2.3	62
2000	68.9	69.3	-1	30.7	55	30.7	55
3000	257.7	270.7	-5	128.3	50	124.2	52
4000	683.2	657.0	4	293.9	57	282.2	59
5000	1365.4	1010.8	26	671.3	51	663.5	51

Table 7: CPU time on Workstation 1 & 2 for the parallel code of Algorithm 3

Algorithm 3 was tested also on Workstation 1, even though the number of its processors is less than the algorithm requires. The time difference obtained for the Algorithm 3 on Workstation 1 (see Figure 4.4) is nearly the same with the baseline algorithm. The results obtained emphasis the importance of having the hardware configuration reflected in the development of the algorithms.

Figure 4.4: CPU time differences on Workstations 1 & 2 for the parallel code of Algorithm 3

An algorithm for eight cores or more can be constructed by applying Algorithm 2 for each matrix inversion encountered in Algorithm 3. Specifically, Algorithm 2 can be applied for each the matrix inversion encountered in the Sequential Phase 1 and 4 of the Algorithm 3.

4.2.3 Remarks

The graph from the Figure 4.5 shows the computational time differences among the sequential runs (OPENMP directives are disabled) of the presented algorithms obtained on all the available workstations.

Figure 4.5: CPU time differences between the sequential runs of the parallel algorithms

The best time for the sequential execution of the codes is obtained by the Algorithm 1. Being executed in serial computation the Algorithm 1 has the lowest number of operations (four inversions and six multiplications) compared to Algorithm 2 (four inversions and six multiplications) and Algorithm 3 (eight inversions and eight multiplications). Algorithm 1 has a better time comparing with basic sequential algorithm because of the reduced number of operations created by implementing the block matrix algorithm.

In the Figure 4.6, it is presented the graph with the differences between the best CPU times obtained for all developed parallel algorithms on the available workstations (OpenMP directives are enabled).

In the above figure one can observe that the results obtained are almost similar in performance. If an analytical analysis is made based on the number of operations required by each algorithm, it can be assumed that the Algorithm 3 which, has the lowest total number of operations per processor (see Figure 3.8), will perform much better in parallel execution than Algorithm 1 and Algorithm 2 which have the same amount of operations (Figure 3.6 and Figure 3.7). Algorithm 3 is expected to perform better considering also its four full-balanced tasks.

Therefore, first important thing to remark is that the Algorithm 3 has a close performance with the others two algorithms even though it has the fewest amounts of operations in parallel execution and a perfect load balancing on four tasks.

The second important aspect from the Figure 4.6 that may be appear surprisingly is that Algorithm 1, which has not a perfectly load balanced on threads, has obtained a slightly better CPU time than the Algorithm 2 which has a perfectly load-balancing; both cases have the same number of operations. Therefore, even if the Algorithm 2 is almost full-parallelized and it is reaching almost the maximum SpeedUp (Figure below) its efficiency is a little inferior to Algorithm 1.

Figure 4.7: Speedups of the Algorithm 1 & 2

Figure 4.8: CPU time differences between for Algorithm 2 & 3 in parallel execution

The third important aspect that can be deduced from Figure 4.8 is referring at the Algorithms 2 and 3 which have almost the same task structure. Even that algorithm 3 is made for four tasks its performance are very close to algorithm 2 which make use only of two tasks. Therefore, by doubling the number of independent tasks from two (Algorithm 2) to four (Algorithm 3) it not always accomplished by a similar increase in performance; both algorithms have a perfectly load balancing. Thence, the focus should be on algorithm structure and not on the tasks structure and theirs number. Similar conclusion can be deducted also from the Speedup comparison between the two algorithms (see Figure 4.9).

Figure 4.9: Speedups of the Algorithm 2 & 3

Algorithm 3 shows a worse overall efficiency than the Algorithm 2, which has also an increased sensitivity to the configuration of the workstations.

Even with the minimal use of OpenMP directives used in current implementations, the synchronizations and communications overheads between the processors affect the parallel efficiency. These issues pull down the performance of the parallel system in a similar way with the three aspects discussed above concerning the Figure 4.6.

Concluding, so far it has been learned that the algorithms design dependencies configurations should be flexible; not always an algorithm which can be fully parallelized or perfectly balanced has a better speedup than an algorithm which have some fraction of code that cannot be parallelized or some imbalanced tasks. Furthermore, it was noticed that the algorithms should be focused on a reduced number of processors; better speedup efficiency was obtained for two tasks than for four tasks, applied for the same algorithm.

Like it was discussed, in the following chapter, the algorithms will be validated on the Laplace problem.

4.3 Laplace Problem

For the Laplace problem it will be applied the algorithms for full matrix inversions (Algorithms 1 and 2) and the general sparse matrix algorithm (Algorithm 4). Apart from algorithms validation, the current simulation signification is as well to emphasize the

advantages of using single-grid approach, implemented in current work, instead of using the classical method of multi-block grid partitioning.

The algorithms were applied for a basic problem of potential flows like the integration of the Laplace equation ($\Delta\phi=0$). The case was solved on a 2D square domain with a uniform Cartesian mesh using a single block and respectively two blocks. This comparison will make observable the influence of the inter-boundary conditions imposed on the common frontier between the two blocks. Discretization for the Laplace problem was made by using central finite differences with 2nd order spatial accuracy for the uniform mesh. As for the boundary conditions, it has been used Dirichlet on the outer frontier and Neumann discretized with the 1st order spatially accurate scheme.

4.3.1 Discussion on the domain decomposition

The Laplace problem was tested for two analytical functions $\phi = xy$ and $\phi = x^2 - y^2$. In general when dealing with multi-block grid parallelization strategies the selection of proper inter-boundary conditions constitute a tricky issue, because it is unpredictable what will happen at the boundary region where the grid is artificially split by the user. In this example, for both analytical functions was imposed the same inter-boundary condition $\frac{\partial\phi}{\partial x} = 0$ which mathematically is ill-posed for the first case, while for the second case is self-consistent with the analytical solution. In the followings, the declared analytical functions are analyzed and compared for single and double-block grid case.

Figure 4.10: Double-block grid and single grid comparison for the first analytical function

In the double-block grid section from the above figure, it can be observed that the ill-posed inter-boundary conditions affected the entire solution field by introducing a non-penetrative (slip) wall effect.

For the second analytical function, the inter-boundary conditions were well posed and due to this, the results for the double-block grid were similar with the ones for the single-grid (see Figure 4.11).

Figure 4.11: Double-block grid and single grid comparison for the second analytical function

The slightly differences obtained between the two methods were because despite its correct inter-boundaries conditions, the double-block grid parallelization shows error depending on the grid resolution at the boundary as it can be observed in the graph below.

Figure 4.12: Grids resolution errors

an accentuated curve of decreasing, but the pressure coefficient leap occurs after the experimental case.

Figure 4.17: Pressure coefficient for NACA 0012, $Mach=0.8$ and $\alpha=1.25^\circ$

It can be observed in the Figure 4.18 that the lift coefficients stabilized after around 1000 cycles (the time unit for each cycle is 0.1). The lift coefficient obtained in present work was of $C_L=0.2768$ compared to $C_L=0.3654$ showed by Jameson in [69].

Figure 4.18: Lift coefficient history for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$

The Figure 4.19 shows the drag coefficients for Mach=0.8 and $\alpha=1.25^\circ$. One can observe that coefficients stabilized after around 1000 cycles.

Figure 4.19: Drag coefficient history for NACA 0012, Mach=0.8 and $\alpha=1.25^\circ$

The drag coefficient was similar with the one showed in the work [69] $C_D=0.0232$.

Figure 4.21: Pressure coefficient for NACA 0012, $Mach=0.5$ and $\alpha=3^\circ$

The maximum lift coefficient for the present work was $C_L=0.3689$ while the experimental data showed in [70] shows a maximum lift coefficient of $C_L=0.4296$. The history of lift coefficients obtained in present work is showed in the Figure 4.22.

Figure 4.22: Lift coefficient history for NACA 0012, Mach=0.5 and $\alpha=3^\circ$

The drag coefficient for the present work is $C_D=0.0080$; the results presented by Jameson for the same case showed a drag coefficient of $C_D=0.0003$. The whole history of drag coefficients can be seen in the Figure 4.23.

Figure 4.23: Drag coefficient history for NACA 0012, Mach=0.5 and $\alpha=3^\circ$

4.4.1 Remarks

The nested sparse general algorithm implementation for the simulation of the airfoil profile (NACA 0012) was validated and obtained comparable results with different experimental data. The results obtained in the present work for the case of Mach=0.5 and an angle of attack of 3° were comparable with the experimental data showed by Jameson. Very good results, were also obtained for the case of Mach=0.8 and an angle of attack of 1.25° , even though the grid implemented in present work was smaller 243×80 (19440 grid points) compare to 321×65 (20865 grid points).

The airfoil profile was tested also at a mach speed of 0.3 for different angles of attack. The present work results were compared with the experimental data obtained by Gregory. The history of lift coefficients was similar with the experimental data, while the history of drag coefficients showed a slightly steeper slope for the present implementation. This was because the present implementation for the airfoil profile ignores the flow viscosity.

Like it was discussed, in the next chapter, the algorithms will be validated on an inviscid compressible vortex transport application.

4.5 Inviscid Compressible Vortex Transport Application

In this section the algorithms will be validated on a vortex transport problem. Practically, the application is referring at transporting an inviscid compressible vortex in different locations of the physical domain. The results obtained are also compared in terms of computational times with the LAPACK routine (SGESV [44]) and in terms accuracy with an iterative algorithm (LU-SGS [55]).

The vortex transport application is implemented on two grids. First grid has a size of 63×20 and the second grid has a size of 153×50 . The smaller mesh is used for validating the full matrix algorithms which are demanding large amounts of computer memory and increased computational times; even on the smaller mesh size the computational time for the full algorithms is in terms of hours and days, which is highly inefficient. Consequently, a smaller application was set up on a 63×20 mesh size, which has to move the vortex 0.4 meters on the physical domain.

The input data for the application was:

- Mesh size: 63x20
- Mach: 0.2
- Alpha: 0 deg
- Time unit: 0.01
- Number of iterations: 200
- Interval of iterations: 1

Except the Algorithm 3 which was found inapplicable for this problem considering its increased requiring for invertible submatrices (see Sub-chapter 4.3.1.2) all the algorithms including the iterative method and the LAPACK routine were tested on the Workstation 3.

For each algorithm the vortex was plotted in three positions: Initial position which shows the starting location of the vortex, middle position which shows the vortex in the middle of its total distance (specifically at 0.2 meters) and the final position which shows the vortex in its final location (specifically at 0.4 meters). For exemplification see the plots from the Figure 4.26.

Figure 4.26: Distance vortex plots for all algorithms including the iterative method and the LAPACK routine

As it can be observed in the Figure 4.26, all the algorithms moved the vortex from its initial to its final position with good accuracy. Regarding the computational times obtained by the algorithms, the time reductions are showed in the Table 8 and Table 9.

Mesh size ($N \times M$)	Baseline alg. time $T_b[\text{sec}]$	Alg.1 time $T_1[\text{sec}]$	Alg.1 time reduction $\frac{T_b - T_1}{T_b}$ (%)	Alg.2 time $T_2[\text{sec}]$	Alg.2 time reduction $\frac{T_b - T_2}{T_b}$ (%)
63x20	348133.06	213979.9	39	113712.81	67

Table 8: Full-matrix algorithms tested on the small mesh

In the above table, one can observe that for long run simulations that are dealing with very large matrices the Algorithm 2 took over Algorithm 1. Therefore, flexible design considerations for the algorithms which were discussed in the section 4.2.3 should be made also according to the specificity of the problem.

Even though the Algorithm 1 and 2 have a better performance than the baseline algorithm (see Table 8), their computational times are still far beyond the human patience. There is a rule of thumb saying that any simulation which runs longer than 8 hours is worthless. The Table 9 contains the computational times for the sparse algorithms and the LAPACK routine which will be considered as a reference.

Mesh size ($N \times M$)	LAPACK time $T_L[\text{sec}]$	Alg.4 time $T_4[\text{sec}]$	Alg.4 time reduction $\frac{T_L - T_4}{T_L}$ (%)	Alg.5 time $T_5[\text{sec}]$	Alg.5 time reduction $\frac{T_L - T_5}{T_L}$ (%)	Alg.5 nest. time $T_{5n}[\text{sec}]$	Alg.5 nest. time reduction $\frac{T_L - T_{5n}}{T_L}$ (%)
63x20	192.41	18.21	90	11.8	93	9.39	95

Table 9: Sparse-matrix algorithms and iterative method tested on the small mesh

In the above table one can observe that among the present work algorithms, the Algorithm 5 nested obtained the best time reduction (95%) compared to LAPACK routine (considered the reference algorithm).

In the next paragraphs accuracy comparative tests will be made between the present work (Algorithm 5 nested) and the iterative method. Like it was discussed, the second grid involved in the present application is an uniform structured grid with the size of 153×50 . The initial vortex position is showed in the Figure 4.27.

Vortex Transport at 5m and 18m	
CFL=0.678	CFL=1.356
Mesh size: 153x50	Mesh size: 153x50
Mach: 0.2	Mach: 0.2
Alpha: 0 deg	Alpha: 0 deg
Time unit: 0.05	Time unit: 0.1
Number of iterations: 5000	Number of iterations: 2500
Interval of iterations: 100	Interval of iterations: 50

Figure 4.28: Vortex transportation at 5 m for $CFL=0.678$

As it can be observed in the Figure 4.28, the present work algorithm was more accurate than the iterative method. The ILU-SGS algorithm showed not only a delayed location of the vortex but also a drop in the vortex pressure field. Also one can observe that on the Y scale the iterative method has a tendency of misleading the vortex. The CFL of 0.678 used is the limits of stability for the iterative method. The next figure shows the same displacement of the vortex (5 m) but for a CFL of 1.356 which makes the iterative method even more instable (see Figure 4.29).

Figure 4.29: Vortex transportation at 5 m for $CFL=1.356$

Like it was discussed, the vortex will be transported also at the distance of 18 m. The simulation is carried on the same computational domain through the use of periodic boundary conditions. These types of boundaries will allow the vortex transport to be cyclic on the same computational domain but represented at different distances. The Figure 4.30 shows the displacement of the vortex at 18 m carried by both methods.

Figure 4.30: Vortex transportation at 18 m for $CFL=0.678$

Even there was a drop in pressure intensity for the current work algorithm (see Figure 4.30) its location is almost similar with the ideal case. The iterative case is also inferior in terms of pressure intensity, but its location is totally misled on both computational axes XY. Its location is delayed with more than 2 meters from the ideal location. For the CFL of 1.345 (see Figure 4.31), the iterative method delayed even more the vortex location. The vortex accurate position is crucial in terms of fluid-dynamics simulations and a delayed of almost 5 meters is impractical. The current work method showed also a slightly delay for the vortex location but compared with the iterative method its accuracy is acceptable.

Figure 4.31: Vortex transportation at 18 m for $CFL=1.356$

To emphasize the difference between the two methods in terms of accuracy, more vortex transportations have been made for different numbers of CFL. In the Table 11 are presented the results obtained for 5 and 18 meters.

CFL	Distance accuracy (%)			
	5 meters		18 meters	
	Present work	Iterative method	Present work	Iterative method
0.67	100	90	100	89
1.35	100	76	99	74
2.71	100	58	98	56
3.39	98	52	97	49
6.78	93	34	92	31

Table 11: Distance accuracy for 5 and 18 meters

One can observe in the above table that the iterative method has not reach 100% accuracy neither for 5 meters and nor for 18 meters. The Figure 4.32 is presenting the distance accuracy graphs of the methods for different CFL numbers ranging from 0.67 to 6.78.

Figure 4.32: Distance accuracy reported to CFL (5 meters)

Figure 4.33: Distance accuracy reported to CFL (18 meters)

It can be seen in the above figure that the iterative method has an almost linear decrease of accuracy. For both measured distances, for a CFL number of 6.78 the present work method has more than 90% accuracy while the iterative method has less than 35% accuracy. This large difference between the two methods is showing that for higher numbers of CFL the accuracy of the present algorithm has an acceptable level while the iterative method almost fails to represent the vortex.

4.5.1 Remarks

In the previous section the present work algorithms were validated and tested for a vortex transport application. The results obtained were compared in terms of computational times with a LAPACK routine and in terms of accuracy with an iterative method.

It has been showed that the full matrix algorithms (specifically Algorithm 1 and Algorithm 2) are not practical for large scale application because of their increase computational times and computer memory requirements. An important fact that was noticed is that for large scale applications the Algorithm 2 took over the Algorithm 1. Therefore, flexible design considerations for the algorithms which were discussed in the section 4.2.3 should be made also according to the specificity of the problem.

More suitable for these types of applications are the sparse algorithms. The sparse Algorithm 5 nested obtained 95% time reduction compared with LAPACK routine. The

results of the Algorithm 5 nested showed also a better vortex accurate transportation then the iterative method. The difference between the two methods is increasing with higher CFL numbers.

Chapter 5 - Conclusions and Future work

5.1 Conclusions

The motivation of the research conducted in this thesis was fueled by the potential to achieve acceptable computational time while enhancing the simulations accuracy by developing and adapting the flow solver parallel algorithms for specific computational equipment. The ability of performing these simulations on simple workstations was an incentive and a final objective of this work. The parallel algorithms developed in the current work showed encouraging results on different applications ranging from inverting full matrices of random values to more advanced tests for Laplace problem, airfoil simulation and vortex transport.

The developed parallel algorithms for the full matrix inversions tests achieved many times 60% more efficiently than the basic sequential code. It has been learned that the algorithms design dependencies configurations should be flexible and according to problem specificity; not always an algorithm which can be fully parallelized or perfectly balanced has a better efficiency than an algorithm which has some fraction of code that cannot be parallelized or some imbalanced tasks. It was also remarked that an increased number of independent tasks, is not always an advantageous solution due to the communication and synchronization overheads between the processors. Also an increased number of tasks require more submatrices to be invertible for a proper application of the Gaussian elimination. To have an increased number of invertible block matrices is not always possible. Therefore, the focus should be on extended strategies for a finite number of processors in order to keep a reduced number of tasks. The additional available processors were used successfully in parallel nested implementations for more advanced applications.

For the Laplace problem solved on multi-block grids, it has been showed that the ill-posed inter-boundary conditions affected the entire solution by introducing a non-penetrative (slip) wall effect. One alternative in order to avoid the uncertainty of using the correct mathematically inter-boundary conditions and the problems caused by this is to use a single grid domain like in the present work. Even though that the inter-boundaries conditions are well posed mathematically, the multi-block grid method introduces error which are depending on the grid resolution. It also has been showed that the full matrix algorithms require more computational times than the double-block grid method. To overcome this, a

sparse matrix algorithm was implemented which took into consideration the sparsity of the matrix.

The sparse algorithm was also implemented for the simulation of an airfoil profile (NACA 0012). The algorithm was successfully validated and obtained comparable results with different experimental used by Jameson and obtained by Gregory. The history of lift coefficients was similar with the experimental data, while the history of drag coefficients showed a slightly steeper slope for the present implementation. This was because the present implementation for the airfoil profile ignores the flow viscosity.

The developed algorithms were successfully validated and tested also for a vortex transport application. The results obtained were compared in terms of computational times with a LAPACK routine and in terms of accuracy with an iterative method. It has been showed that the full matrix algorithms are not practical for large scale application because of theirs increase computational times and computer memory requirements. More suitable for these types of applications are the sparse algorithms. The sparse nested algorithm obtained 95% time reduction compared with LAPACK routine and showed also a better vortex accurate transportation then the iterative method. The difference in terms of accuracy between the two methods was increasing with higher CFL numbers.

In the present work has been showed that applications of vortex flows which are carried usually on more advanced computers can be successfully implemented on nowadays workstations. The portability between advanced computers and usual workstations has been made by having adaptive algorithms to numerical solution conform to workstation's hardware configuration. The results showed enough accurate solutions and affordable computational times.

Concluding it can be said that the currents work objectives has been reached and future research work can be carried base on the results obtained.

5.2 Future work

Considering the vast experience accumulated during the developing of the current thesis, several future recommendations are suggested in the following five paragraphs:

1. A future work for the present thesis can be conducted for the optimization of the developed parallel algorithms in terms of shared memory allocation and utilization. Analysis memory utilization tests should be performed for all the algorithms and according to the results obtained, modifications should be made for improving the algorithms performance.
2. Most of the algorithms were developed for dual-core architectures and the additional processors which were available on different tested platforms have been used for nested implementations. More advanced algorithms that approach different parallelization strategies can be similarly developed with a basic structure but for quad-core configurations.
3. A more advanced nested parallelism for the present work algorithms should be developed for a better utilization of the additional processors. Current thesis was not conducted for an optimized nested parallelization and hence, additional improvements can be made by enhancing the research in this direction.
4. The range of applications tested for the present work algorithms should be extended also to 3D simulations. By using the advanced algorithms recommended in the previous sections advanced simulations can be performed with acceptable computational times.
5. Finally, the OS (Operating System) platform used for implementing and validating the algorithms was Microsoft Windows. Additional benchmarks tests should be performed also on other operating systems (e.g. Linux), for a better utilization and allocation of the hardware resources.

Chapter 6 - References

- [01] G. H. Yeoh – *Computational Fluid Dynamics: A Practical Approach*, Book published by Butterworth-Heinemann, 2008, Pages: 2-12
- [02] David J. Kuck, Constantine D. P. – “Special Issue on Languages, Compilers and Environments for Parallel Programming”, *Journal of Parallel and Distributed Computing Volume 5 Issue 5*, October 1988, Pages: 430-459
- [03] Lionel M. N., K. C. Tai – “Special Issue on Software Tools for Parallel Programming and Visualization”, *Journal of Parallel and Distributed Computing Volume 9 Issue 2*, November 1990, Pages:101-102
- [04] William Gropp, Ewing Lusk, Anthony Skjellum – *Using MPI: Portable Parallel Programming with the Message-passing Interface*, Book published by MIT Press, 1994, Pages:14-22
- [05] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Ramesh Menon, Jeff McDonald – *Parallel Programming in OpenMP*, Book published by Morgan Kaufmann, 2001, Pages: 97-123
- [06] Max Domeika – “Development and Optimization Technique for Multi-Core Processors”, *Embedded Systems Conference Volume 4 Issue 6*, December 2007, Pages: 42-63
- [07] Tan C. C. – “Multicore Parallel Computing with OpenMP”, *SUV/Academic Computing Volume 38 Issue 5*, September 2007, Pages: 571-572
- [08] Sarita V. Adve [...] – “Parallel Computing Research at Illinois – The UPRC Agenda”, Department of Computer Science, November 2008, Pages: 29-43
- [09] Hill, Mark D., Marty, Michael R. – “Adamhl’s Law in the Multicore Era”, *Computer Volume 41 Issue 7*, June 2008, Pages: 33-54
- [10] John L. Gustafson – “Reevaluating Amdahl’s law”, *Communications of the ACM Volume 13 Issue 5*, May 1988, Pages: 532-535
- [11] Jay Hoeflinger, Thomas Jackson, Bob K. – “Producing scalable performance with OpenMP: Experiments with two CFD applications”, *Parallel Computing Volume 27 Issue 4*, March 2001, Pages: 391-413
- [12] Edward L. Turner, Hong Hu – “A parallel CFD rotor code using OpenMP”, *Advances in Engineering Software Volume 32 Issue 8*, June 2001, Pages: 665-671

- [13] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, Thomas Turnbull – “Implementation of Strassen’s Algorithm for Matrix Multiplication” , *ACM/IEEE Conference on Supercomputing*, Article No.: 32, October 1996, Pages: 112-120
- [14] Beny Neta, Heng-Ming Tai – “LU factorization on parallel computers”, *Computers & mathematics with applications Volume 11 Issue 6*, June 1985, Pages: 573-579
- [15] G. Cybenko – “Dynamic load balancing for distributed memory multiprocessors”, *IEEE Transactions on Parallel and Distributed Computing Volume 7 Issue 2*, October 1989, Pages: 279-301
- [16] C. Walshaw, M. Cross, M. Everett – “Parallel dynamic graph partitioning for adaptive unstructured meshes”, *Journal of Parallel and Distributed Computing Volume 47 Issue 2*, September 1997, Pages: 102-108
- [17] A. T. Chronopoulos, A. M. Wissink, M. Benche – “Static load balancing for CFD simulations on a network of workstations”, *IEEE International Symposium on Network Computing and Applications Volume 8 Issue 10*, October 2001, Pages: 364-367
- [18] David C. Yu, Huili Wang – “A new Parallel LU Decomposition Method”, *Power Systems - IEEE Transactions Volume 5 Issue 1*, February 1990, Pages: 303-310
- [19] Dogan Kaya, Ken Wright – “Parallel algorithms for LU decomposition on a shared memory multiprocessor”, *Applied mathematics and computation Volume 19 Issue 8*, January 2005, Pages: 925-937
- [20] C. M. Bergman – “Parallelization of CFD codes”, *Computer Methods in Applied Mechanics and Engineering Volume 89 Issue 3*, August 1991, Pages: 523-528
- [21] C. S. Ierotheou, C. R. Forsey, M. Leatham – “The implementation of an aeronautical CFD flow code onto distributed memory parallel systems”, *International Journal for Numerical Methods in Fluids Volume 32 Issue 8*, April 2000, Pages: 903-920
- [22] Lucy T. Zhang, Gregory J. Wagner, Wing K. L. – “A parallelized Meshfree Method with Boundary Enrichment for Large-Scale CFD”, *Journal of Computational Physics Volume 176 Issue 2*, March 2002, Pages: 483-506
- [23] A. Povitsky, M. Wolfshtein – “Parallelization efficiency of CFD problems on MIMD computer”, *Computers & Fluids Volume 26 Issue 4*, May 1997, Pages: 359-371
- [24] M. Darwish, T. Saad, Z. Hamdan – “Parallelization of an additive multigrid solver”, *Numerical Heat Transfer Volume 54 Issue 2*, August 2008, Pages: 157-184
- [25] R. Winkelmann, J. Hauser, R. D. Williams – “Strategies for parallel and numerical scalability of CFD codes”, *Computer Methods in Applied Mechanics and Engineering Volume 174 Issue 3-4*, May 1999, Pages: 433-456

- [26] Michael Schafer – *Computational Engineering: Introduction to Numerical Methods*, Book published by Springer Science & Business, 2006, Pages: 200-215
- [27] P. G. Richards, S. J. Zhang – “A parallel unstructured Chimera grid method”, *ACM Southeast Regional Conference Volume 6 Issue 9*, June 2005, Pages: 301-302
- [28] R. Banos Navarro, C. Gil Montoya – “Graph and Mesh Partitioning: An overview of the Current State-of-the-Art”, *Mesh Partitioning Techniques and Domain Decomposition Methods Volume 30 Issue 7*, March 2007, Pages: 785-801
- [29] G. L. Miller, S. Teng, W. Thurston, S.A. Vavasis – “Automatic mesh partitioning in Sparse Matrix Computations: Graph Theory Issues and Algorithms”, *IMA volumes in Mathematics and its Application Volume 14 Issue 5*, September/October 1993, Pages: 1029-1046
- [30] G. Karypis, V. Kumar- “METIS – family of multilevel partitioning algorithms”, *Dpt. Of Computer Science and Engineering Volume 25 Issue 3*, March 1999, Pages: 504-517
- [31] Dinesh K. Kaushik, David E. Keyes, Barry F. Smith – “High-performance parallel implicit CFD”, *Parallel Computing Volume 27 Issue 4*, March 2001, Pages: 337-362
- [32] Ian S. Duff – “The impact of high-performance computing in the solution of linear systems: trends and problems”, *Journal of Computational and Applied Mathematics Volume 123 Issue 1-2*, November 2000, Pages: 515-530
- [33] R. D. Williams – “Performance of dynamic load balancing algorithms for unstructured mesh calculations”, *Concurrency: Practice and Experience Volume 3 Issue 5*, June 1991, Pages: 457-481
- [34] M. J. Berger, S.H. Bokhari – “A partitioning strategy for non-uniform problems on multiprocessors”, *IEEE Transactions on Computers Volume 36 Issue 5*, May 1987, Pages: 570-580
- [35] S.T. Barnard, H. D. Simon – “A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems”, *Concurrency: Practice and Experience Volume 6 Issue 2*, March 2006, Pages: 532-541
- [36] G. Karypis, V. Kumar – “A fast and high quality multilevel scheme for partitioning irregular graphs”, *SIAM Journal on Scientific Computing Volume 20 Issue 1*, March 1998, Pages: 359-392
- [37] B. W. Kernighan, S. Lin, “An efficient heuristic procedure for partitioning graphs”, *Bell system technical journal Volume 49 Issue 1*, September 1970, Pages: 299-307
- [38] B. Hendrikson, R. Reland – The CHACO USER GUIDE, Technical Report, 1994, Pages: 134-156
- [39] F. Pellegrini – “Scotch and Libscotch USER GUIDE”, *Research Report*, 2006, Pages: 56-70

- [40] C. Walshaw, M. Cross – “JOSTLE – Multilevel Graph Partitioning Software: An overview”, *Mesh partitioning techniques and domain decomposition techniques Volume 5 Issue 4*, June 2007, Pages: 27-58
- [41] C. B. Allen – “Parallel Simulation of lifting rotor wakes in forward flight”, *International Journal of Numerical Analysis and Modeling Volume 4 Issue 1*, October 2007, Pages: 1-15
- [42] J. Kang, B. Feng, L. Xiao – “Auto-CFD-now: A pre-compiler for effectively parallelizing CFD applications on networks of workstations”, *Journal of Supercomputing Volume 38 Issue 2*, November 2006, Pages:189-217
- [43] C. B. Allen – “Parallel simulation of unsteady hovering rotor wakes”, *International Journal for Numerical Methods in Engineering Volume 68 Issue 6*, April 2006, Pages: 632-649
- [44] E. Anderson, Z. Bai, C. Bischof, J. D. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK USER GUIDE, Release 2.0, 2nd ed. SIAM, 1995, Pages:110-142
- [45] C. Addison, Y. Ren, M. Waveren – “OpenMP issues Arising in the Development of Parallel BLAS and LAPACK libraries”, *Scientific Programming Volume 11 Issue 2*, April 2003, Pages:95-104
- [46] Iain S. Duff – “Sparse numerical linear algebra: direct methods and preconditioning”, *Technical Report RAL-TR-96-047*, October 1997, Pages:27-62
- [47] Horst D. Simon – “Partitioning of unstructured problems for parallel processing”, *Computer Science Volume 2 Issue 2-3*, June 1994, Pages: 135-148
- [48] F. Magoules, F.X. Roux – “Algorithms and Theory for Substructuring and Domain Decomposition Methods”, *Computer Science Volume 4 Issue 2*, October 2007, Pages:89-118
- [49] Stephen M. Watt – “Pivot-Free Block Matrix Inversion”, *Maple Conference Volume 4 Issue 1-2*, July 2006, Pages: 151-155
- [50] H. Saberi Najafi, M. Shams Solary – “Computational algorithms for computing the inverse of a square matrix, quasi-inverse of a non-square matrix and block matrices”, *Applied Mathematics and Computation Volume 183 Issue 1*, December 2006, Pages 539-550
- [51] V. Strassen, “Gaussian Elimination is Not Optimal”, *Numerical Mathematics Volume 13 Issue 3*, January 1969, Pages: 354-356
- [52] M. J. Flynn – “Some computer organizations and their effectiveness”, *IEEE Transactions on Computers Volume 21 Issue 9*, September 1972, Pages: 948-960
- [53] A. Grama, A. Gupta, G. Karypis, V. Kumar – *Introduction to Parallel Computing, Second Edition*, Book published by Addison Wesley, 2003, Pages: 67-154

- [54] Michael Jay Quinn – *Parallel Programming in C with MPI and OpenMP*, Book published by McGraw-Hill Professional, 2004, Pages: 120-148
- [55] Yoon S., Jameson A. - “Lower- Upper Symmetric- Gauss-Seidel Method for the Euler and Navier-Stokes Equations”, *AIAA Journal Volume 26 Issue 14*, September 1988, Pages: 1025-1026
- [56] Gilbert Strang – *Introduction to Linear Algebra*, Book published by Wellesley-Cambridge Press and SIAM, 2003, Pages:205-321
- [57] James E. Gentle – *Matrix Algebra: Theory, Computations and Applications in Statistics*, Book published by Springer, 2007, Pages: 287-310
- [58] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery – *Numerical Recipes in Fortran*, Book published by Cambridge University Press, 1992, Pages:34-67
- [59] Joe F. Thompson, B. K. Soni, N.P. Weatherill – *Handbook of grid generation*, Book published by CRC Press, 1999, Pages:111-119
- [60] S. Akl - *The Design and analysis of parallel algorithms*, Book published by Prentice-Hall, 1989, Pages:23-202
- [61] P.J.Eberlein, "On the Schur Decomposition of a Matrix for Parallel Computation", *IEEE Transactions on Computers Volume 36 Issue 2*, February 1987, Pages: 167-174
- [62] Mahmood, Juliusz Dichter, “An efficient parallel algorithm for the inverse of a banded matrix in the maximum entropy sense”, *IEEE SMC Volume 4 Issue 6*, October 2002, Pages: 244-286
- [63] I.S. Duff, H.A. van der Vorst, “Developments and trends in the parallel solution of the linear systems”, *Parallel Computing Volume 25 Issue 13-14*, December 1999, Pages: 1931-1970
- [64] P.R. Amestoy, I.S. Duff, “Vectorization of a multiprocessor multi-frontal code”, *Internat. J. Supercomputer Applications Volume 3 Issue 3*, September 1989, Pages: 41-59
- [65] T.A. Davis and I.S. Duff, “Unsymmetric-pattern multi-frontal methods for parallel sparse LU factorization”, *SIAM Journal on Matrix Analysis and Applications Volume 18 Issue 1*, January 1991, Pages:140-158
- [66] I.S. Duff, J.K. Reid, “The multi-frontal solution of indefinite sparse symmetric linear systems”, *ACM Transactions on Mathematical Software Volume 9 Issue 3*, September 1983, Pages: 302-325
- [67] J.W.H. Liu, “The multi-frontal method for sparse matrix solution: Theory and practice”, *SIAM Review Volume 34 Issue 1*, March 1992, Pages: 82-109
- [68] Patrick R. Amestoy, Abdou Guermouche, Jean-Yves L’Excellent, Stephane Pralet, “Hybrid scheduling for the parallel solution of linear systems”, *Parallel Computing Volume 32 Issue 2*, February 2006, Pages: 136-156

- [69] Antony Jameson – “A perspective on computational algorithms for aerodynamic analysis and design”, *Progress in Aerospace Sciences Volume 37 Issue 2*, February 2001, Pages: 197-243
- [70] Antony Jameson – “Accelerated Solution Methods for the Euler Equations”, *Theoretical Aerodynamics Contractors Workshop Volume 3 Issue 4*, January 1991, Pages:83-92
- [71] N. Gregory, P. G. Wilby – “NPL 9615 and NACA 0012 A Comparison of Aerodynamic Data”, *Aeronautical Research Council C.P. No. 1261*, February 1973, Pages: 306-310
- [72] O. Buneman – “A compact non-iterative Poisson solver”, Report 294, Institute for plasma Research, Stanford University, California, March 1969, Pages: 265-274
- [73] H. H. Wang – “A parallel method for tri-diagonal equations”, *ACM Transactions on Mathematical Software Volume 7 Issue 2*, June 1981, Pages: 170-183
- [74] R. Courant, K. Friedrichs and H. Lewy – “On the partial difference equations of mathematical physics”, *IBM Journal Volume 100 Issue 3*, March 1967, Pages: 32-74
- [75] Jennifer A. Scott – “A parallel frontal solver for finite element applications”, *International Journal for Numerical Methods in Engineering Volume 29 Issue 4*, December 2001, Pages: 395-417
- [76] H. Dym, I. Gohberg – “Extensions of Band Matrices with Band Inverses”, *Linear Algebra and Its Applications Volume 36 Issue 1*, March 1981, Pages: 1-24