Development of Parallel Meshless Methods for Moving Geometry Simulations

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy
by
Juan Jacobo Angulo

April 2014

Abstract

Computational fluid dynamics methods to simulate flows around geometries in relative motion are important for the aerospace industry. Traditional methods like finite-volume techniques are better suited for static simulations where the geometry of the problem does not change, or where only small movements are found. The meshless method can provide a solution for these problems where the geometry changes significantly and different bodies can move in relation to one another.

A meshless method to select stencils from overlapping and moving point distributions, and a corresponding flow solver capable of solving the Euler equations on those stencils, have been developed previously. This work expands the existing meshless formulation by including the capabilities to simulate viscous flows in laminar and turbulent regimes and by implementing different parallel computing techniques in an effort to improve the computational efficiency.

The treatment of viscous and turbulent flows is performed by augmenting the original Euler meshless scheme by using central-differences to discretise the viscous terms in the Navier-Stokes equations. The Spalart-Allmaras turbulence model is used to model the turbulent viscosity term and complete the closure of the system of equations to be solved. Validation of the method was carried out by calculating several well-known test cases and comparing the results to published data.

The parallel implementation of the flow solver follows a distributed approach with asynchronous communications using message-passing standards. The parallel flow solver method is tested with two three-dimensional geometries, running in dedicated parallel machines with processor numbers ranging in the thousands. Results show good agreement to published data and very good parallel scalability.

Preliminary testing of the stencil selection method, showed that the computational cost of the operations needed to find stencils for each point in the domain can vary dramatically for all points. Furthermore, this cost cannot be predicted a-priori, making it very difficult to perform an appropriate domain decomposition. With this in mind, three types of implementation are used for the parallel stencil selection scheme: a distributed memory approach, a shared memory approach and hybrid method combining the two previous ones. Using the shared and hybrid implementations, the negative effects of using a poor domain decomposition are reduced. Four test cases are studied using the parallel stencil selection procedure coupled with the parallel flow solver. Two of these cases are static, and two of them are simulations over moving geometries. The fourth case introduces a 6 degree-of-freedom simulation to calculate the movement of a store being released from an aircraft and showcases the full capabilities of the method. These parallel tests show important reductions in the calculation times and open the door for the meshless scheme to be used in the future for more realistic cases.

Acknowledgements

I would like to extend my gratitude to my supervisors Professor Ken Badcock and Professor George Barakos for their assistance and support, and to all the members of the Computational Fluid Dynamics Laboratory at the University of Liverpool, past and present.

To my friends in Liverpool and back home, I thank you for your continuous support during both the good and hard times, and especially to Fleur: your love and support during these complicated times have been invaluable. I love you.

Last, but certainly not least, I would like to thank my family, who have believed in me at all stages of my life, and to whom I owe everything. Mireya, Rafael and Andrea. I love you very much.

This research has been financially supported by the Engineering and Physical Sciences Research Council (EPSRC) and BAE Systems through a CASE award. This work made use of the facilities of N8 HPC provided and funded by the N8 consortium and EPSRC (Grant No.EP/K000225/1). The Centre is co-ordinated by the Universities of Leeds and Manchester.

Declaration

I confirm that the thesis is my own work, that I have not presented anyone else's work as my own and that full and appropriate acknowledgement has been given where reference has been made to the work of others.

Juan Jacobo Angulo
April 2014

List of Publications

Angulo, J. J., Kennett, D. J., Timme, S., and Badcock, K. J., "Parallel Methods for a Semi-Meshless Euler and Navier-Stokes Solver", Submitted to AIAA Journal.

Angulo, J. J., Kennett, D. J., Timme, S., and Badcock, K. J., "Parallel Semi-Meshless Stencil Selection for Moving Geometry Simulations," *AIAA Paper 2013–2854*, Presented at the 21st AIAA Computational Fluid Dynamics Conference, San Diego, California, Jun 2013.

Kennett, D. J., Angulo, J. J., Timme, S., and Badcock, K. J., "Semi-Meshless Stencil Selection on Three-Dimensional Anisotropic Point Distributions with Parallel Implementation," *AIAA Paper 2013–0867*, Presented at the 51st AIAA Aerospace Sciences Meeting, Grapevine, Texas, Jan 2013.

Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "Semi-Meshless Stencil Selection for Anisotropic Point Distributions," *International Journal of Computational Fluid Dynamics* Vol. 26, Nos. 9–10, 2012, pp. 463–487

Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "An Implicit Meshless Method for Application in Computational Fluid Dynamics," *International Journal for Numerical Methods in Fluids* Vol. 71, No. 8, 2012, pp. 1007–1028

Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "An Implicit Semi-Meshless Scheme with Stencil Selection for Anisotropic Point Distributions," *AIAA Paper 2011–3234*, Presented at the 20th AIAA Computational Fluid Dynamics Conference, Honolulu, Hawaii, June 2011.

Table of Contents

List of Figures

List of Tables

List of Symbols

A	=	Jacobian matrix ($= \partial \boldsymbol{R}/\partial \boldsymbol{w}$)
a	=	speed of sound
b, c, d	=	shape function derivatives
c	=	chord length
C_d	=	drag coefficient
C_f	=	skin friction coefficient
C_l	=	lift coefficient
C_m	=	moment coefficient
C_p	=	pressure coefficient
d	=	distance to closest wall
e	=	specific total energy
E	=	total energy
E_p	=	parallel efficiency
$\mathbf{f}$, $\mathbf{f^v}$	=	inviscid, viscous fluxes in the x direction
$\mathbf{g}$, $\mathbf{g^v}$	=	inviscid, viscous fluxes in the y direction
$\mathbf{h}$, $\mathbf{h^v}$	=	inviscid, viscous fluxes in the z direction
h	=	specific enthalpy
i	=	star point
I	=	identity matrix
j	=	neighbour point
k	=	Cartesian direction (1=x, 2=y, 3=z)
L	=	level of the sub-division by quadrants
m	=	pseudo-time level
$m_{\boldsymbol{p}}$	=	order of polynomial
M_∞	=	freestream Mach number
$\mathbf{n}$	=	unit normal vector
n	=	real-time level
N	=	number of total points in the domain
$\mathbf{p}$	=	vector of primitive variables
$\boldsymbol{p}$	=	polynomial
p	=	pressure

Pr, Pr_t	=	Prandtl number, turbulent Prandtl number
$\mathbf{q}$	=	heat flux vector
$\mathbf{R}$	=	residual vector
R	=	gas constant
Re	=	Reynolds number
S_p	=	parallel speed-up
t	=	time
T	=	temperature
u	=	velocity component in the x direction
v	=	velocity component in the y direction
$\mathbf{v}$	=	Cartesian velocity vector
w	=	velocity component in the z direction
x, y, z	=	Cartesian coordinates
$\mathbf{x}$	=	Cartesian coordinate vector
$\mathbf{w}$	=	vector of conserved variables

Greek Symbols

α	=	freestream angle of attack
α_s	=	non-parallelisable fraction of any parallel algorithm
γ	=	ratio of specific heats
δ	=	boundary layer thickness
ϵ	=	tolerance
$\boldsymbol{\eta}$	=	basis vector
$\boldsymbol{\tau}$	=	stress tensor
κ	=	thermal conductivity
μ, μ_t	=	viscosity, turbulent viscosity
$\tilde{\nu}$	=	turbulent eddy viscosity (transport variable for Spalart-Allmaras model)
$\boldsymbol{\xi}$	=	vector of basis coefficients
ρ	=	density
ϕ	=	any function
$\hat{\phi}$	=	meshless approximation to function
ψ	=	slope limiter
ς	=	pseudo-time

Acronyms

6-DOF	=	six-degree of freedom
AGARD	=	Advisory Group for Aerospace Research and Development
BILU	=	block-incomplete lower–upper
CFD	=	computational fluid dynamics
CFL	=	Courant-Friedrichs–Lewy
DEM	=	diffuse element method
DLR	=	German Aerospace Center
EFG	=	element-free Gelerkin
FEM	=	finite element method
FPM	=	finite point method
GCR	=	generalised conjugate residual
HPC	=	high-performance computing
MIMD	=	multiple-instruction-multiple-data
MISD	=	multiple-instruction-single-data
MLPG	=	meshless local Petrov-Galerkin
MLS	=	moving least squares
MPI	=	message passing interface
NUMA	=	non-uniform memory access
NS	=	Navier–Stokes
ONERA	=	French Aerospace Lab
OSF	=	open-source fighter
PDE	=	partial-differential equation
PMB	=	parallel multiblock
PML	=	parallel meshless
RANS	=	Reynolds–averaged Navier–Stokes
RKPM	=	reproducing kernel particle method
SA	=	Spalart–Allmaras
SISD	=	single-instruction-single-data
SIMD	=	single-instruction-multiple-data
SMC	=	symmetric multi-core
SPH	=	smoothed particle hydrodynamics

Chapter 1

Introduction

1.1 Motivation

The analysis of complex flows around bodies in relative motion is required by the aerospace industry. Example applications include the release of stores from aircraft, the opening of cavity doors, the deployment of control surfaces, helicopter blades in rotation, flapping wings, and wind turbine blades. This requirement has driven the development of numerical tools that can successfully deal with complex movable configurations. Conventional mesh generation techniques become difficult or impossible to apply when used to calculate flows over bodies in relative motion. For this reason, several techniques have been developed in the last few years to tackle the simulation of flow around bodies with parts in relative motion.

One technique is the meshless method, which discretises the domain by using a set of points. Each point in the domain has a sub-domain of neighbouring points, called a stencil or cloud. These clouds are then used to approximate the spatial derivatives in the partial differential equations to be solved. The meshless method is attractive for moving-body problems, as points can move independent of one another during a time-dependent simulation.

Fluid simulation techniques are known to be computationally intensive and meshless schemes are no exception. For the meshless method described in this work to be useful in aerospace applications, it needs to provide fast and accurate predictions of turbulent flows. Even though the speed and memory size of modern processors has advanced, they are still not sufficient to carry out the computations required by the industry. For this reason it is obvious that parallel processing is required. The application of parallel computing to meshless schemes is still an open area of research and the method can be divided into two problems: selecting the stencils for all points and then solving the governing equations. Finding a balance between the parallel efficiency of the stencil selection operations and actually solving the equations proves not to be straight forward, which translates into an interesting topic of research.

The current work documents the research carried out to implement parallel algorithms for the solution of turbulent flows in aerospace applications using the meshless method.

1.2 Objectives and Thesis Outline

The main aim of this thesis is the development of parallel methods applied to a meshless fluid dynamics scheme for the simulation of turbulent flows over moving geometries. Three objectives were drawn and achieved in this work.

- The treatment of viscous and turbulent flows was included into an existing inviscid meshless flow solver.
- Parallel computing methods were implemented for the flow solver to reduce calculation times.
- A new parallel algorithm for the selection of stencils used by the flow solver was devised.

Chapter 2 describes some of the previous work carried out in the field of simulation of flows over moving geometries, as well as an introduction of some of the challenges found when using parallel computers to simulate these problems. It also contains a basic introduction on parallel computing terminology. Chapters 3 and 4 describe the mathematical and numerical formulation used to model aerodynamic flows using the meshless scheme as well as the description of the implementation of the laminar and turbulence models. Several test cases that validate the laminar and turbulent capabilities of the solver are presented in Chapter 5. The thesis continues with the description of the computational methods used to parallelise the meshless flow solver on Chapter 6, and the presentation of two test cases that assess the efficiency of the parallel implementation on Chapter 7. This is then followed by the detailed description of the method used to select stencils in parallel in Chapter 8, along with validation cases that test the full capabilities of the parallel meshless scheme in Chapter 9. Four test cases are used for this purpose, including a case that combines the calculation of stencils, aerodynamic loads and rigid-body motion, all performed in parallel. Finally in Chapter 10, conclusions are drawn, together with suggestions for possible future work.

Chapter 2

Theoretical Background and Literature Review

2.1 Overview of Numerical Methods for Moving Geometries

Among the most commonly used techniques applied to simulating flows around moving geometry, we can name the following three methods: self-adapting grids, sliding grids, and Chimera or overset Grids. Each of them has its own advantages and disadvantages.

Self-Adapting Grids (Cartesian / Unstructured)

The foremost advantage of using self-adapting meshing is the automation that the procedure offers in grid generation. Apart from this, Cartesian or unstructured meshes in conjunction with tree data structures become a natural choice for solution-adaptive grids and dynamic flow computations involving moving bodies [1–3].

Several studies have shown the capabilities of adapting grids to compute inviscid and low Reynolds number flows [4–6]. In spite of the success involving inviscid flow computations, the main drawback of using adaptive Cartesian or tetrahedral meshes for viscous flows over complex geometries is the fact that automatically filling boundary layers with isotropic cells results in a high number of control volumes, ultimately leading to large grids [7–9].

Several researchers have tried dealing with these problems [2, 10–12]. One of the possible solutions is to use hybrid grids, where the viscous layer is filled with stretched, body-fitted structured elements, and the rest of the domain is filled with Cartesian or unstructured grids. Other methods include developing advanced pre-processors to generate the viscous grid by a suitable projection procedure [2, 13–15]. All of these methods can hamper automatic grid generation, so the fact remains that for practical applications, the required grids for calculating three-dimensional turbulent flows over complex geometries are too big using adaptive Cartesian or automatic triangulation techniques.

The parallel implementation of automatic grid generators has received a good amount of attention from researchers [16–21]. Several techniques like Octree subdivision, Delaunay triangulation and the Advancing-Front Technique have proven successful in dealing with different fluid problems with good parallel efficiency. The main drawback of using this type of method is still the fact that even though they are generated quickly, the grids generally are not well suited for viscous and turbulent flow computations because of the increase in grid size needed to capture the flow characteristics near solid boundaries.

Sliding Grids

The concept of sliding meshes in Computational Fluid Dynamics (CFD) was first introduced for the analysis of turbo-machinery [22–24]. This method allows for the integration of two domains (meshed separately) by interpolating the flow variables along the surface joining the two domains. Sliding meshes have been successfully used and validated in different studies where the movement between the domains is known a priori, including helicopter rotor-fuselage interaction [25, 26], turbo-machinery [27–29] and wind turbine blades [30, 31].

The sliding planes method can be implemented in parallel without much difficulty. Each separate processing unit can calculate the flow solution on its assigned data, as long as the code provides the information about the movement of cells around the sliding interface [25,32]. The main issues with this method are the interpolation needed to transfer the values from one grid to the other, and the fact that the movement of the geometry is restricted by the mating surface. This makes the method work well for problems with rotating bodies, but makes its use almost impossible for problems where the movement of the geometry is not known a-priori.

Chimera (Overset Grids)

The Chimera technique [33–35] is most often associated with finite volume/difference schemes, and its functionality is based on overlapping different grids belonging to each body or moving part. The method uses interpolations to estimate flow properties in the overlap region and besides allowing for the treatment of movable geometries, it also helps in reducing the meshing time as different parts of the domain can be meshed separately and then joined together. Normally structured grids are used with this technique.

Chimera has been successfully used for aerodynamic computations [36–39], but there are still some drawbacks with the technique. The main difficulties include: 1) The fact that the procedure to cut the grids, generate the interpolating region and interpolate the flow variables can be time consuming. 2) The complexity of the interconnectivity is perhaps as difficult as dealing with an unstructured grid, resulting in orphan points and bad quality of interpolation stencils [40]. 3) The fact that interpolation is generally used to connect grids implies that conservation is not strictly enforced [40].

The parallel implementation of the overset grids method is still a field of active research. Several studies have been successful at implementing the Chimera scheme in parallel [41–44]. Even so, the parallel efficiency of the connectivity methods still poses a problem. The main difficulty found is that the cost of the hole-cutting and search operations varies greatly for different regions in the computational domain, and finding an estimate of these costs to perform a correct decomposition of the domain is very difficult. In most cases, this ultimately results in low parallel efficiencies.

Meshless Methods:

Traditional methods used in CFD (Finite Volumes, Finite Element, Finite Difference, etc) use grids or meshes as the underlying structures where the partial derivatives are discretised. Contrary to grid based methods, Meshless schemes do not require a connected grid since they can discretise the derivatives from local clouds, formed at each of the points in the domain (See Fig. 2.1). Meshless methods can be used to model problems with large geometry deformations, making them an excellent choice for applications where different bodies can interact with each other.

Although meshless methods can provide a solution for moving geometries, they are not without their drawbacks. Among the most important ones we note: 1) Finding suitable candidates to form the local clouds can be time consuming. 2) Similar to Chimera methods, by forming the local clouds of points it is possible that conservation is not strictly enforced. 3) Because of the nature of the local clouds, achieving spatial higher-order accuracy may not be as straight-forward as with traditional finite-volume methods. Even with these drawbacks, meshless methods have been shown capable of providing accurate solutions to complex problems [45, 46] and become an interesting proposition for the simulation of flows over moving geometries in aerospace applications.

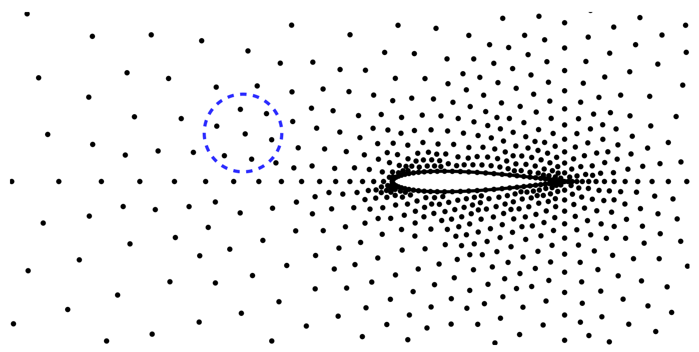


Figure 2.1: Points across a 2D domain. Local Meshless Stencil.

2.2 Meshless Numerical Methods in Fluid Dynamics

Meshless methods for CFD is an active area of research. Important developments have been made in the last few years that have contributed to the understanding of the principles that allow the approximation functions to be built and used for solving the Navier-Stokes equations.

Main Developments Documented in the Literature

The first steps towards developing meshless methods for CFD were made almost 30 years ago. The starting point for meshless research is known as the Smoothed Particle Hydrodynamics (SPH) method and dates back to the 1970s. SPH was developed by Lucy [47] and Monaghan [48] between 1977 and 1982 to model problems in Astrophysics such as explosions of stars of particle clouds. The idea behind SPH is to replace the fluid by a set of moving particles and transforming the governing partial differential equations into the kernel estimates integrals. SPH uses a pseudo-particle interpolation method to compute smooth field variables. Each pseudo-particle has mass, Lagrangian position, velocity and internal energy. Other quantities are derived by interpolation or from constitutive relations. These particles have a spatial distance ("smoothing length"), over which their properties are "smoothed" by a kernel function. Any physical quantity is then obtained by summing the relevant properties of all the particles which lie within two smoothing lengths. Although the particles are not connected in SPH, the partitioning of the domain into volume elements is difficult, especially in three dimensions.

The original ideas from SPH have had a big influence in the development of meshless methods in general. Several techniques have been developed over the years to improve the effectiveness of this method. Most of the advances were produced by incorporating powerful interpolation techniques, initially developed for data processing and surface generation. Swegle et al. [49] used dispersion analysis of the linearised equations to find the origin of the so-called "tensile instability" and proposed an artificial viscosity to stabilize it. Dyka [50] then proposed a different stabilization method by means of stress particles. Further progress was made by Liu and Chen [51] and Liu et al. [52] by developing the Reproducing Kernel Particle Method (RKPM). The reproducing kernel in this method is similar to the SPH Lagrangian method with one major difference: the development of a correction function for boundary effects [52]. With this function, the tensile instability was eliminated. The SPH method has been successfully applied to a wide range of problems such as free surface, impact and explosion simulation, heat conduction and many other computational mechanics problems [53, 54]. Even with the many improvements that have been made throughout the years, SPH is still viewed by some as unstable and inaccurate when compared with other methods for complex fluid simulation [55], unless a large number of particles is used.

Parallel to the development of Lagrangian particle methods like SPH, Nayroles et al. [56] introduced the use of moving least square approximations in their Diffuse Element Method (DEM). The Diffuse Element Method uses moving least squares interpolation to replace the Finite-Element Modelling (FEM) functions, valid in one element, with a weighted minimum squares approximation, valid for a small localised domain around one point [57]. The approximation function is smoothed by introducing continuous functions instead of discontinuous coefficients. The fact that these weighting functions vanish at a certain distance from the main node allows for the preservation of the local character of the approximation. It can be seen that for DEM, each of the points can be considered as a particular type of finite element, with one singular integration point, a variable number of nodes and a diffuse domain of influence.

Belytschko et al. [58] then extended the idea of least squares approximation by developing a method where the spatial discretisation was made by using moving least squares and a Galerkin formulation. This scheme was called Element-Free Galerkin method (EFG) and was originally devised to solve progressive crack growth in structural mechanics. The method is seen as an extension of the DEM by Nayroles, and introduced two main improvements: 1) It used an auxiliary background mesh of regular cells in order to create a structure to define the quadrature points, thus allowing for the numerical integration to be performed. 2) It was able to enforce the essential boundary conditions by using Lagrange multipliers. The method showed good accuracy, as well as convergence rates which rivalled finite element methods, even though it was computationally more expensive than FE models. The EFG method has found applications in many fields such as fracture, crack and wave propagation, acoustics and fluid flow [59].

An important step towards true meshless methods was the Meshless Local Petrov-Galerkin method (MLPG), proposed by Atluri et al. [60]. The MLPG method arose from the finite element community and is based on the weak form of a given PDE [61]. MLPG incorporates the moving least squares approximations for trial and test functions to discretise the local weak form of the equations. The method is based on a Petrov-Galerkin formulation in which weight and trial functions used in the discretisation of the equations do not need to be the same. This gives the method a "local" nature in which the integral is satisfied over a local domain [61]. The MLPG approach has been used successfully to solve different problems, including work on incompressible flows [62], fracture mechanics [63], and three-dimensional elasto-statics and dynamics [63].

More recently, several methods that can be referred to as Finite Point Methods (FPMs) have been developed. They are usually based on the strong form of the PDEs. In general, FPMs are based on least squares fitting of functions to discrete points. Batina [64] proposed the use of a polynomial based approximation in conjunction with least squares to compute the derivatives of the fluid governing equations. His method provided approximations to both the Euler and Navier-Stokes equations and was successfully used to solve viscous flows about complex aircraft configurations. The first official use of the term Finite Point Methods was provided by Oñate [65]. He combined a weighted least square approximation of the unknowns over each local cloud with a stabilised point collocation procedure, eliminating any numerical instability. FPMs have been successfully used in several problems, including compressible inviscid and viscous flows [66,67].

Katz and Jameson [68] developed a formal meshless scheme which compared favourably with conventional finite volume methods in terms of accuracy and efficiency for the Euler and Navier-Stokes equations. The success of their method is attributed to its local extremum diminishing property, which they generalised to handle local clouds of points instead of mesh-based schemes. The method adopts an edge-based connectivity to describe local points and uses Taylor series expansions with weighted least squares for the reconstruction of the gradients found in the PDEs.

Parallel Computing and Meshless Methods

Even though in recent years meshless schemes that are suitable for simulation of complex flows are becoming more common, most of the published work focuses on the mathematical description of the methods, without addressing the computational efficiency. Some researchers have implemented meshless schemes that solve the governing equations in parallel [69–72], but few have addressed the problem of parallelising the selection of local stencils. References [69, 73, 74] are among the few published works that describe the stencil selection in parallel and their method, while showing great parallel efficiency, is based on triangulation and is aimed at working on isotropic point distributions to simulate incompressible flows. To the best of the authors knowledge there has not been any published work that deals with parallel implementations for meshless schemes aimed at simulating flows over complex three-dimensional, movable geometries.

2.3 Parallel Computing Concepts

In order to simulate most scientific problems it is necessary to perform a large number of numerical computations. Historically, the desire to run increasingly complex problems has been running ahead of the capabilities of the time. This has provided a driving force for the development of parallel computing. A variety of parallel computer architectures have been made available during the years. One way to classify these systems is according to Flynn's taxonomy [75]. It uses the relationship between the *instruction stream* and the *data stream* to classify the four different possible architectures:

- **SISD:** Single Instruction stream operating on a Single Data stream. This is a standard sequential computer, such as personal computer (PC).

- **SIMD:** Single Instruction stream operating on Multiple Data stream. A set of processors execute the same instruction on different sets of data. The shared memory eliminates the need for message passing constructs.

- **MIMD:** Multiple Instruction streams operating on Multiple Data streams. These are the most versatile parallel computers. They are essentially a set of different processors that can run the same or different programs with the same or different data sets. Each processor controls its own memory and runs asynchronously. Communication between processors is accomplished via message-passing constructs.

- **MISD:** Multiple Instruction streams operating on a Single Data stream computers are included for completeness as there are few, if any, commercial examples.

Most computers now found in scientific applications and all the machines used in this work fall into the MIMD classification. The following naming conventions for hardware components are used throughout the thesis:

- Central processing unit (CPU), also referred to as “core”: the component that carries out the instructions of a program by performing the basic arithmetical, logical, control, and input/output operations of the system.

- Processor: physical chip containing one or more independent central processing units. All processors used in this thesis are of the multi-core type, hence they contain two or more cores.

- Computing node: self-contained computer with one or more processors.

An important sub-classification of an MIMD machine is obtained according to their memory distribution:

- **Shared Memory:** Central processing units share a global memory space (See Fig.2.2(a)). The key feature is the use of a single address space across the whole system, so that all CPUs have access to the same view of memory. Symmetric multi-core (SMC) chips found in most modern computers fall into this category.

- **Distributed Memory:** Processing units have their own private memory space as shown in Fig.2.2(b). Access to data assigned to other CPUs must be done through a network. Common clusters of sequential workstations with dedicated networks fall into this category.

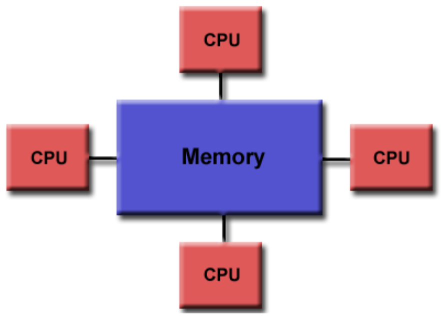
It is interesting to note that most modern clusters and high-performance-cumputing (HPC) systems have a combination of the memory distributions described above, as several nodes containing symmetric multi-core processors are interconnected through a network as depicted in Fig.2.3.

In modern SMC systems, memory access becomes the main bottleneck as the number of processing units is increased. Different CPUs would require access to the same memory space and the "system bus", which can be thought of as a pathway between the CPUs and the memory is not big enough to transfer all the data needed by all cores at the same rate as when only one core is being used. To alleviate this problem, processor manufacturers design modern chips so that each independent core inside the chip has access to its own bus and its own memory space. This type of architectures is known as Non-Uniform Memory Access (NUMA). As these still are shared memory architectures, access to the entire memory space within the system is still possible, but cores can access their own memory space much faster than accessing the space assigned to other CPUs. For this reason care needs to be taken to ensure the data to be processed is stored locally within the space assigned to each core.

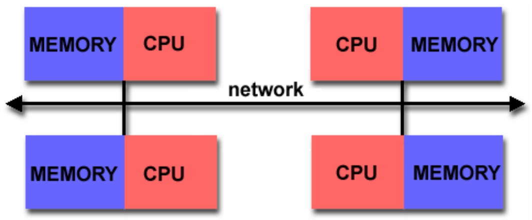
A parallel algorithm is often effective and efficient only on a specific target architecture which must be carefully considered during the development. High level programmers normally do not need to deal with the network topology. However, it is useful to consider a few aspects of the networks that are relevant for the design of the algorithms [76].

The standard network model involves two parameters that define the data transfer rate. The first is *network latency L [s]*, which is the time needed to initiate the connection between two processing units. The second is the *network bandwidth B [bytes/s]*, which is the rate at which data is exchanged. Because of the latency, it is better to send one long message rather than a set of short messages, even if the total amount of data to be transferred is the same.

On distributed memory machines, the communication between CPUs is made by message-passing directives. These directives have two basic types: point-to-point and global communications; respectively referring to message passing among specific cores in the system, or message passing to all cores. Message passing can also be classified as *blocking* and *non-blocking*. Blocking messages stop execution of the code until the



(a) Shared Memory System



(b) Distributed Memory System

Figure 2.2: Parallel Architectures

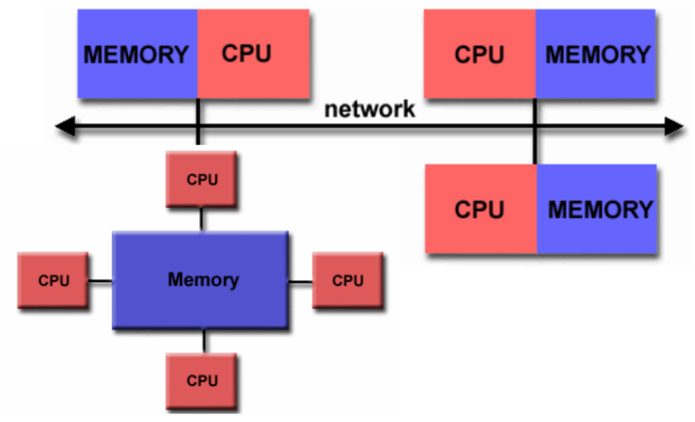


Figure 2.3: Combined Memory System

message is received. In contrast, non-blocking messages continue with the execution of the code as soon as the communication order is issued.

There are two metrics normally used to measure the performance of parallel algorithms [77]. The first metric is speed-up. Speed-up indicates how much faster an application runs on p parallel CPUs compared to on one:

$$S_p = \frac{T_1}{T_p} \tag{2.1}$$

where T_p and S_p are the time the application takes and the speed-up for p cores, respectively, and T_1 is the time the application takes on one core. Often, a linear speed-up is not possible to achieve as there is extra work involved in distributing work to the CPUs and coordinating them. In addition, an optimal serial algorithm may be able to do less work overall than an optimal parallel algorithm for certain problems, so the achievable speed-up may be sub-linear in p, even on theoretical ideal machines.

The second metric is efficiency and is given by

$$E_p = \frac{S_p}{p}$$

where, E_p and S_p are the efficiency and speed-up for p cores, respectively. An efficiency of 1.0 (100%) indicates that every core is being used to the full extent of its capabilities. Usually, efficiency measures that are significantly lower than 100% are due to communication overheads or the unbalanced distribution of the problem over the CPUs involved [78].

Historically, there have been two schools of thought when dealing with parallel algorithms: Amdahl's law [79] and Gustafson's law [80]. Amdahl's law focuses on the theoretical speed-up limits for parallel algorithms running on an increasing number of CPUs while the problem size remains fixed. This is known as "strong scalability". He argued that the execution time can be divided between two categories: the time spent doing non-parallelisable work, and the time spent doing parallel work. Amdahl's law argues that there is a limit to the possible speed-up based on the portion of the code that is sequential. Using this idea, the attainable parallel speed-up follows the relation

$$S_p = \frac{1}{\beta + \dfrac{(1-\beta)}{p}} \quad = \quad \frac{p}{\beta p + (1-\beta)}$$

where β is the portion of the code that must be computed sequentially. The time required for the sequential portion of the code is βT_1, and the time required for the parallel portion is $(1-\beta)T_1/p$. It is clear that as p tends to infinity, the upper bound for the speed-up becomes $S_\infty = \frac{1}{\beta}$.

Amdahl's equations assume however, that the computation problem size does not change when running on an increasing number of machines, hence, the fraction of the problem that is parallelisable remains fixed. On the other hand, Gustafson noted that problem sizes grow as computers become more powerful, and that higher speed-ups than originally predicted by Amdahl's law were in fact possible when using massively parallel machines. Gustafson's law [80] addresses the theoretical limits introduced by Amdahl by concluding that speed-up should be measured by scaling the problem to the number of cores, and not by fixing the problem size. This approach is referred to as "weak scalability." In Gustafson's law the parallel speed-up is defined as

$$S_p = p - \alpha_s(p-1)$$

where α_s is the non-parallelisable fraction of any parallel algorithm. Following Gustafson's law, an application is then called *scalable* if an increase in the size of the problem can be countered by a corresponding increase in the number of CPUs, and the time required for the application remains constant.

Another important concept in parallel computing is domain decomposition versus control decomposition. In domain decomposition, the domain of the input data is partitioned and the partitions are assigned to different CPUs. In control decomposition, program tasks are divided and distributed among processing units. [77]. This decomposition is balanced if the amount of work assigned to each core is equal. The attempt to balance the decomposition is known as "load balancing" [77].

The granularity of an application indicates the amount of processing that can be completed between required message passing events. A "fine-grained" application has few operations between message passing events. A "coarse-grained" application has numerous operations to perform between message passing events. If the grain is too small, communication can dominate the time required to complete the application.

The final concept to be introduced here is the one of *threads*, which are instruction sequences that can run concurrently and are managed by the operating system at run time. Threads can be used to parallelise code on multi-core architectures as every core can execute a separate set of instructions concurrently.

2.4 Challenges of Parallel Computing Applied to Meshless Methods

The problem of applying parallel computing to the meshless method is an interesting one. The meshless method starts by finding a group of neighbours (associated stencil) for each point in the domain and then, it solves the governing equations on each of these points. Finding the stencils and solving the governing equations are in fact two different problems that need to be tackled separately.

The main difficulties with the implementation of the parallel method are to correctly load balance the problem and to maintain parallel communications as well as memory consumption to a minimum. The computational cost per point for the solution of the governing equations is roughly the same for all the points in the domain. This is not the case for the selection of the stencils. All points in the domain are surrounded by other points that are possible candidates to form part of the stencils. There may be cases where some points are surrounded by only a few candidates, making the calculations for the stencil selection quite fast. On the other hand, there will be points that are surrounded by many candidates and the process of selecting the appropriate stencils is slow.

Finding a proper balance that increases the parallel efficiency of the stencil selection as well as the solution of the governing equations, while maintaining low memory usage, is the main difficulty to solve.

Chapter 3

Governing Equations

The Navier-Stokes equations are a system of partial differential-equations (PDEs) that define the conservation of mass, momentum and energy of fluids. They form the basis of the CFD formulation. Most aerodynamic flows are characterised by Reynolds numbers well above the critical value for transition, and thus turbulence needs to be taken into account. Viscosity plays a major role in many engineering cases and can be viewed as having two major components: a laminar one and a turbulent one. The laminar viscosity is usually a function of temperature and can be estimated using Sutherland´s formula. The turbulent viscosity depends on the mean flow characteristics and needs to be evaluated separately. In this work the turbulent viscosity is calculated by using turbulence models.

In this section we describe the mean flow equations and the Spalart-Allmaras turbulence model.

3.1 Navier-Stokes Equations

The motion of viscous fluids can be described by the Navier-Stokes equations. In three dimensions, the equations are written in differential conservative form as:

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial}{\partial x}(\mathbf{f} - \mathbf{f^v}) + \frac{\partial}{\partial y}(\mathbf{g} - \mathbf{g^v}) + \frac{\partial}{\partial z}(\mathbf{h} - \mathbf{h^v}) = 0 \tag{3.1}$$

where $\mathbf{w}$ is the vector of conserved variables, $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$ are the inviscid fluxes in the x, y and z directions respectively, and $\mathbf{f^v}$, $\mathbf{g^v}$ and $\mathbf{h^v}$ are the viscous fluxes along the same directions. They are defined as:

$$\mathbf{w} = \left\{ \begin{array}{c} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{array} \right\} \quad \mathbf{f} = \left\{ \begin{array}{c} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ (e+p)u \end{array} \right\} \quad \mathbf{g} = \left\{ \begin{array}{c} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho uw \\ (e+p)v \end{array} \right\} \quad \mathbf{h} = \left\{ \begin{array}{c} \rho v \\ \rho uw \\ \rho vw \\ \rho v^2 + p \\ (e+p)w \end{array} \right\}$$

$$\mathbf{f^v} = \left\{ \begin{array}{c} 0 \\ \tau_{xx} \\ \tau_{xy} \\ \tau_{xz} \\ u\tau_{xx} + v\tau_{xy} + w\tau_{xz} - q_x \end{array} \right\} \quad \mathbf{g^v} = \left\{ \begin{array}{c} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{yz} \\ u\tau_{xy} + v\tau_{yy} + w\tau_{yz} - q_y \end{array} \right\}$$

$$\mathbf{h^v} = \left\{ \begin{array}{c} 0 \\ \tau_{xz} \\ \tau_{yz} \\ \tau_{zz} \\ u\tau_{xz} + v\tau_{yz} + w\tau_{zz} - q_z \end{array} \right\} \tag{3.2}$$

Pressure is related to the conservative variables via the equation of state for a perfect gas:

$$p = (\gamma - 1)\left[e - \frac{1}{2}\rho(u^2 + v^2 + w^2)\right] \tag{3.3}$$

The shear and normal stresses found in the viscous fluxes are:

$$\tau_{xx} = \frac{2}{3}(\mu + \mu_t)\frac{1}{Re}\left(2\frac{\partial u}{\partial x} - \left(\frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right) \tag{3.4}$$

$$\tau_{yy} = \frac{2}{3}(\mu + \mu_t)\frac{1}{Re}\left(2\frac{\partial v}{\partial y} - \left(\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z}\right)\right) \tag{3.5}$$

$$\tau_{zz} = \frac{2}{3}(\mu + \mu_t)\frac{1}{Re}\left(2\frac{\partial w}{\partial z} - \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right) \tag{3.6}$$

$$\tau_{xy} = (\mu + \mu_t)\frac{1}{Re}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \tag{3.7}$$

$$\tau_{xz} = (\mu + \mu_t)\frac{1}{Re}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \tag{3.8}$$

$$\tau_{yz} = (\mu + \mu_t)\frac{1}{Re}\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \tag{3.9}$$

$$q_x = -\frac{\gamma}{\gamma-1}\frac{1}{Re}(\frac{\mu}{Pr}+\frac{\mu_t}{Pr_t})\frac{\partial}{\partial x}(\frac{p}{\rho}) \tag{3.10}$$

$$q_y = -\frac{\gamma}{\gamma-1}\frac{M_\infty}{Re}(\frac{\mu}{Pr}+\frac{\mu_t}{Pr_t})\frac{\partial}{\partial y}(\frac{p}{\rho}) \tag{3.11}$$

$$q_z = -\frac{\gamma}{\gamma-1}\frac{1}{Re}(\frac{\mu}{Pr}+\frac{\mu_t}{Pr_t})\frac{\partial}{\partial z}(\frac{p}{\rho}) \tag{3.12}$$

where Re, the Reynolds Number, and a, the speed of sound are defined by:

$$Re = \frac{\rho_\infty \, c \, a_\infty}{\mu_\infty} \tag{3.13}$$

$$a = \sqrt{\gamma R T} \tag{3.14}$$

The values of γ and Pr are 1.4 and 0.72 respectively in this work. In the viscous fluxes, μ and μ_t represent the dynamic laminar and dynamic turbulent eddy viscosities respectively. Here, μ_t is determined using the Spalart-Allmaras turbulence model and μ is determined using Sutherland's law:

$$\mu = \frac{\overline{\mu}}{\mu_\infty} = \left(\frac{\overline{T}}{T_\infty}\right)^{3/2} \frac{T_\infty + S^*}{\overline{T} + S^*} \tag{3.15}$$

where S^* is the Sutherland constant of 110.33K for air, T_∞ is the freestream temperature of 255.56K for air.

The above equations have been non-dimensionalised using the following relations:

$$x = \frac{\overline{x}}{c} \qquad y = \frac{\overline{y}}{c} \qquad z = \frac{\overline{z}}{c} \tag{3.16}$$

$$u = \frac{\overline{u}}{u_\infty} \qquad v = \frac{\overline{v}}{v_\infty} \qquad w = \frac{\overline{w}}{w_\infty} \tag{3.17}$$

$$\rho = \frac{\overline{\rho}}{\rho_\infty} \qquad e = \frac{\overline{e}}{\rho_\infty u_\infty} \qquad t = \frac{\overline{t} u_\infty}{c} \tag{3.18}$$

where c is the chord length. The subscript ∞ identifies freestream values, and the overbar, dimensional values.

3.2 Turbulence Modelling (Spalart - Allmaras Model)

When simulating turbulent flows, very fine computational grids and small time-steps are needed to capture the complex flow structures that develop as the calculation progresses. For most industrial applications, these requirements make the calculations infeasible for resolving all scales on the grid. Instead of solving for all the flow characteristics directly, an approximated solution can be calculated. One such approximation is done by modelling the effects of the small-scale motions on the computed mean-flow values (large-scale flow). This approach is known as turbulence modelling for the Reynolds-Averaged Navier-Stokes (RANS) equations. In order to quantify the influence of the turbulence on the resolved flow, a closure model needs to be introduced. There are several different approaches to turbulence modelling [81,82] and usually, each of them is better suited to different applications [83–86]. The current work is aimed at simulating aerodynamic applications with compressible flows at medium to high Mach numbers. As a first approach to introducing turbulence modelling to the meshless scheme, the Spalart-Allmaras (S-A) model [87] is implemented.

The Spalart-Allmaras turbulence model solves a differential expression for the turbulence variable $\tilde{\nu}$. The model includes the treatment of transition to turbulence, but in the present study the flow is assumed to be fully turbulent. Assuming this simplification, the model in conservative dimensional form is

$$\frac{D\tilde{\nu}}{Dt} = C_{b1}\tilde{S}\tilde{\nu} + \frac{1}{\sigma}[\nabla \cdot ((\nu + \tilde{\nu})\nabla\tilde{\nu}) + C_{b2}(\nabla\tilde{\nu})^2] - C_{w1}f_w \left[\frac{\tilde{\nu}}{d}\right]^2 \tag{3.19}$$

with the material on the left hand side derivative described as

$$\frac{D\tilde{\nu}}{Dt} = \frac{\partial\tilde{\nu}}{\partial t} + \mathbf{v} \cdot \frac{\partial\tilde{\nu}}{\partial\mathbf{x}} \tag{3.20}$$

where $\mathbf{v} = [u, v, w]^T$ and $\mathbf{x} = [x, y, z]^T$. The term $\tilde{\nu}$ is called the turbulent eddy viscosity and contributes to the Navier-Stokes equations (Eq. 3.1) through the turbulent viscosity

$$\mu_t = \rho\tilde{\nu}f_{v1} \tag{3.21}$$

where the viscous damping function f_{v1} is given by

$$f_{v1} = \frac{\chi^3}{\chi^3 + C_{v1}^3} \tag{3.22}$$

and χ is the ratio of the kinematic eddy turbulent viscosity to the kinematic laminar viscosity ($\nu = \mu/\rho$),

$$\chi = \frac{\tilde{\nu}}{\nu} \tag{3.23}$$

The production term is modelled by

$$\tilde{S} = S f_{v3} + \frac{\tilde{\nu}}{\kappa \nu^2 d^2} f_{v2} \tag{3.24}$$

$$f_{v2} = \frac{1}{(1+\chi/C_{v2})^3} \qquad f_{v3} = \frac{(1+\chi f_{v1})(1-f_{v2})}{\max(\chi, 0.001)} \tag{3.25}$$

where d is the distance from the nearest solid wall and κ is the von Karman constant, equal to 0.41. S is the magnitude of the vorticity, written as

$$S = \sqrt{2\Omega_{ij}\Omega_{ij}} \tag{3.26}$$

where Ω_{ij} is the mean rate of rotation tensor so Eq. 3.26 becomes

$$S = \sqrt{\left(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}\right)^2} \tag{3.27}$$

The values in the destruction term are

$$f_w = g\left[\frac{1+{C_{w3}}^3}{g^6+{C_{w3}}^6}\right]^{\frac{1}{6}} \tag{3.28}$$

$$g = r + C_{w2}(r^6 - r) \tag{3.29}$$

$$r = \frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2} \tag{3.30}$$

where C_{w2} and C_{w3} are constants. For large r the function f_w reaches a constant so large values of r can be truncated to 10.

The various constants used in the model have the following values

$$\begin{aligned} C_{b1} &= 0.1355 & C_{b2} &= 0.622 \\ C_{v1} &= 7.1 & C_{v2} &= 5.0 \\ C_{w1} &= \frac{C_{b1}}{\kappa^2} + \frac{1+C_{b2}}{\sigma} & C_{w2} &= 0.3 \\ C_{w3} &= 2 & \sigma &= \frac{2}{3} \\ \kappa &= 0.41 \end{aligned}$$

To non-dimensionalise the S-A transport equation we use the same reference values as with the N-S equations, plus the following:

$$\mu^* = \frac{\overline{\mu}}{\mu_\infty}, \qquad \mu_t^* = \frac{\overline{\mu_t}}{\mu_\infty}, \qquad \tilde{\nu}^* = \frac{\overline{\tilde{\nu}}}{\tilde{\nu}_\infty} \tag{3.31}$$

where the superscript * denotes non-dimensional values. The superscript * can however be dropped for convenience, and the S-A equation can be written in non-dimensional form as

$$\begin{aligned}\frac{\partial\tilde{\nu}}{\partial t} + u\frac{\partial\tilde{\nu}}{\partial x} + v\frac{\partial\tilde{\nu}}{\partial y} + w\frac{\partial\tilde{\nu}}{\partial z} = C_{b1}\tilde{S}\tilde{\nu} + \frac{1}{\sigma Re_\infty}[\nabla\cdot((\nu+\tilde{\nu})\nabla\tilde{\nu}) \\ + C_{b2}(\nabla\tilde{\nu})^2] - \frac{C_{w1}f_w}{Re_\infty}\left[\frac{\tilde{\nu}}{d}\right]^2\end{aligned} \tag{3.32}$$

where the auxiliary functions are redefined in non-dimensional form as

$$\chi = \frac{\tilde{\nu}}{\nu} \qquad f_{v1} = \frac{\chi^3}{\chi^3 + {C_{v1}}^3} \tag{3.33}$$

$$f_{v2} = \frac{1}{\left(1 + \frac{\chi}{C_{v2}}\right)^3} \qquad f_{v3} = \frac{(1+\chi f_{v1})(1-f_{v2})}{\max(\chi, 0.001)} \tag{3.34}$$

$$f_w = g\left[\frac{1+{C_{w3}}^3}{g^6 + {C_{w3}}^6}\right]^{\frac{1}{6}} \qquad g = r + C_{w2}(r^6 - r) \tag{3.35}$$

$$r = \frac{\tilde{\nu}}{Re_\infty\,\tilde{S}\kappa^2 d^2} \qquad \tilde{S} = Sf_{v3} + \frac{\tilde{\nu}}{\kappa\nu^2 d^2}f_{v2}\cdot\frac{1}{Re_\infty} \tag{3.36}$$

Chapter 4

Solution Method

4.1 Approximation of Continuous Functions from Scattered Data

Scattered data approximation deals with the problem of reconstructing a function or its derivatives from given disperse data. While the idea of scattered data approximation is not new, it has recently become a fast growing area of research. For a given domain in space, discretised by a set of N points, it is possible to define a local cloud for each of the points, as shown in Fig. 4.1. These local clouds, or stencils, are then used to approximate the required functions. In this work, the central point of each of these local stencils is referred to as the “star point” and denoted by the sub-index i.

There are several different meshless methods that deal with interpolation from scattered data. A modern overview of these methods can be found in Ref. [88]. Some of the most commonly used ones include Radial Basis Functions, Moving Least Squares approximations and Reproducing Kernel Particle methods, among others. All of these methods can be used to obtain the partial derivatives of a function ϕ at each of the star points in the domain by interpolating scattered data from the points contained in the associated stencil of the star point. This can be written as:

$$\frac{\partial \phi_i}{\partial x} = \sum_{j=0}^{n_i} a_j \phi_j, \quad \frac{\partial \phi_i}{\partial y} = \sum_{j=0}^{n_i} b_j \phi_j, \quad \frac{\partial \phi_i}{\partial z} = \sum_{j=0}^{n_i} c_j \phi_j \tag{4.1}$$

where i denotes the main point, j represents each of the points in the stencil, n_i is the number of points in the stencil with $j = 0$ being the star point and a_j, b_j and c_j are coefficients independent of the function ϕ. These coefficients, called shape functions, are found using Polynomial Basis Functions combined with the least-squares method. The following description of the method assumes a two-dimensional problem. The generalisation to three-dimensional problems is trivial since the procedure is the same.

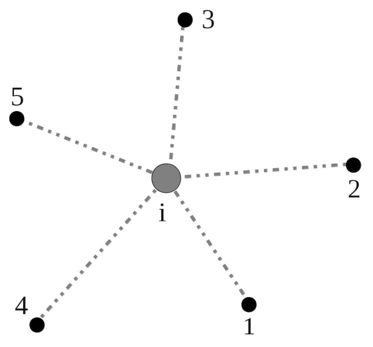


Figure 4.1: Local Cloud of Points.

For a function $\phi(\boldsymbol{x})$ defined at discrete values inside the local cloud, an approximation $\hat{\phi}(\boldsymbol{x})$ can be constructed using polynomials ($\boldsymbol{p}$) in the form:

$$\hat{\phi}(\boldsymbol{x}) = \boldsymbol{p}(\boldsymbol{x})^T \alpha \tag{4.2}$$

where

$$\boldsymbol{p}(\boldsymbol{x}) = [1 \quad x \quad y \quad \dots \quad p_{m_{\boldsymbol{p}}}(\boldsymbol{x})]^T \tag{4.3}$$

$$\alpha = [\alpha_0 \quad \alpha_1 \quad \dots \quad \alpha_{m_{\boldsymbol{p}}}] \tag{4.4}$$

In this work, the approximation to the function ϕ is obtained by using a first order polynomial ($m_{\boldsymbol{p}} = 3$):

$$\hat{\phi}(x, y) = \alpha_0 + \alpha_1 x + \alpha_2 y \tag{4.5}$$

where the coefficients α_0, α_1, and α_2 can be determined using a least-squares curve fit. Performing a least-squares fit in a given cloud of points results in three equations represented in matrix form by

$$\begin{bmatrix} n_i & \Sigma x_i & \Sigma y_i \\ \Sigma x_i & \Sigma x_i^2 & \Sigma x_i y_i \\ \Sigma y_i & \Sigma x_i y_i & \Sigma y_i^2 \end{bmatrix} \begin{Bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{Bmatrix} = \begin{Bmatrix} \Sigma f_i \\ \Sigma x_i f_i \\ \Sigma y_i f_i \end{Bmatrix}$$

The solution of the system of equations requires the inversion of a 3 x 3 matrix which is performed for every cloud in the computational domain. Having solved these equations for α_0, α_1 and α_2, the spatial derivatives are now known since by differentiating Eq. (4.5) it is obvious that

$$\frac{\partial \hat{\phi}(x, y)}{\partial x} = \alpha_1 \qquad \frac{\partial \hat{\phi}(x, y)}{\partial y} = \alpha_2$$

Using this method it is possible to approximate the derivatives of any of the primitive values found in the governing equations. As an example, the derivatives of the velocity in the horizontal direction can be written as:

$$\frac{\partial u}{\partial x} = \sum_{j=0}^{n_i} \alpha_{1_j} \cdot u \tag{4.6}$$

where α_j corresponds to the shape function a_j from equation 4.1. In addition to defining the derivatives of the fluxes $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$ in the governing equations, this same method can also be used to find the shear stresses and heat fluxes needed for the viscous fluxes.

4.2 Spatial Discretisation of Non-Viscous Fluxes

For inviscid flow, equation 3.1 is re-written as:

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} + \frac{\partial \mathbf{g}}{\partial y} + \frac{\partial \mathbf{h}}{\partial z} = 0 \tag{4.7}$$

where $\mathbf{w}$ is the vector of conserved variables, and $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$; are the inviscid flux vectors. This system of equations is hyperbolic in nature, hence a centered form of spatial discretisation will be unstable. In order to correctly solve the governing equations we compute upwind fluxes along the co-ordinate directions at a fictitious interface formed between the star point and each of the neighbours, as shown in Fig. 4.2

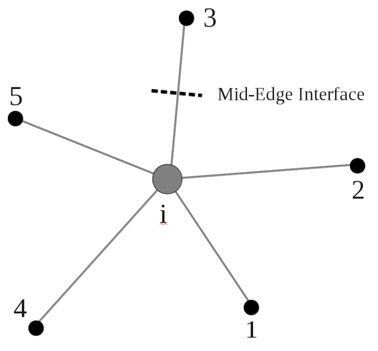


Figure 4.2: Mid-Edge Interface.

Then, the discretisation takes the form:

$$\frac{\mathrm{d}\mathbf{w}_i}{\mathrm{d}t} = -\sum_{j=0}^{n_i} \left(a_{j-\frac{1}{2}}\, \mathbf{f}_{j-\frac{1}{2}} + b_{j-\frac{1}{2}}\, \mathbf{g}_{j-\frac{1}{2}} + c_{j-\frac{1}{2}}\, \mathbf{h}_{j-\frac{1}{2}} \right) \tag{4.8}$$

where $b_{j-\frac{1}{2}}$, $c_{j-\frac{1}{2}}$ and $d_{j-\frac{1}{2}}$ are the shape functions calculated from the polynomial-least squares reconstruction evaluated at the mid-edge interface. The fluxes $\mathbf{f}_{j-\frac{1}{2}}$, $\mathbf{g}_{j-\frac{1}{2}}$ and $\mathbf{h}_{j-\frac{1}{2}}$ are evaluated using the approximate Riemann solver of Roe [89] with an appropriate entropy correction technique. Using this procedure, the mid-edge fluxes are calculated with:

$$\mathbf{f}_{j-\frac{1}{2}} = \frac{1}{2}(\mathbf{f}(\mathbf{p_L}) + \mathbf{f}(\mathbf{p_R})) - \frac{1}{2}|A(w_L, w_R)|(w_R - w_L) \tag{4.9}$$

where $\mathbf{p_L}$ and $\mathbf{p_R}$ are the vectors of primitive values at the left and right hand side of the interface, and A is the Jacobian matrix evaluated based on the Roe's average properties $\tilde{W}(w_L, w_R)$. For a first order accurate scheme, the vectors $\mathbf{p_L}$ and $\mathbf{p_R}$ simply become:

$$\mathbf{p_L} = \mathbf{p}_i$$
$$\mathbf{p_R} = \mathbf{p}_j$$

To increase the accuracy of the solver in the presence of shocks, a higher order scheme is used. Then, the vectors $\mathbf{p_L}$ and $\mathbf{p_R}$ are obtained by extrapolating the values at i and j, based on a reconstructed gradient as in Ref. [45]:

$$\begin{aligned} \mathbf{p_L} &= \mathbf{p}_i + \psi_{ij}\, \mathbf{l}_{ij} \cdot \nabla \mathbf{p}_i \\ \mathbf{p_R} &= \mathbf{p}_j - \psi_{ij}\, \mathbf{l}_{ij} \cdot \nabla \mathbf{p}_j \end{aligned} \tag{4.10}$$

where $\mathbf{l}_{ij}$ is the vector formed between the star and neighbouring point, ψ_{ij} is an appropriate flux limiter, and $\nabla\phi$ denotes the gradient of ϕ. A sufficient condition to avoid introducing oscillation in the solution process is that no new local extrema are formed during reconstruction [90]. The idea behind the slope limiters consists of finding a value ψ_{ij} ($\in [0, 1]$) in each stencil that will limit the gradient in the piecewise-linear reconstruction of the solution. The following procedure was proposed by Barth and Jespersen [91] and is used in this work:

1. Find the largest negative ($\delta\phi_i{}^{min} = \min(\phi_j - \phi_i)$) and positive ($\delta\phi_i{}^{max} = \max(\phi_j - \phi_i)$) difference between the solution in the immediate neighbours and the star point in the current stencil.

2. Compute the unconstrained reconstructed value at each star point ($\overline{\phi_{ij}} = \phi_i + \mathbf{l}_{ij} \cdot \nabla \mathbf{p}_i$).

3. For each point j in the stencil, compute a maximum allowable value of a function φ_{ij} defined as

$$\varphi_{ij} = \begin{cases} \min(1, \frac{\delta\phi_i{}^{max}}{\overline{\phi_{ij}} - \phi_i}), & \text{if } (\overline{\phi_{ij}} - \phi_i) > 0 \\ \min(1, \frac{\delta\phi_i{}^{min}}{\overline{\phi_{ij}} - \phi_i}), & \text{if } (\overline{\phi_{ij}} - \phi_i) < 0 \\ 1, & \text{if } (\overline{\phi_{ij}} - \phi_i) = 0 \end{cases}$$

4. Select $\psi_{ij} = \min(\varphi_{ij})$

4.3 Spatial Discretisation of Viscous Fluxes

The discretisation of the viscous fluxes requires a slightly different approach since they involve second order derivatives. Furthermore, the viscous fluxes in the governing equations are diffusive in nature, so a central difference scheme can be employed.

The same least squares coefficients used to discretise the partial derivatives of the Euler (inviscid) fluxes, are used to compute the gradients in the viscous fluxes. The gradients are calculated at each of the star points, so a least squares reconstruction centered at the points is used. After finding the gradients of the primitive variables, the same shape functions calculated at mid edge from the least squares coefficients are used again to calculate the second order derivatives:

$$\nabla\phi = \frac{\partial\phi}{\partial x}\ \hat{\mathbf{i}} + \frac{\partial\phi}{\partial y}\ \hat{\mathbf{j}} + \frac{\partial\phi}{\partial z}\ \hat{\mathbf{k}} \tag{4.11}$$

$$\frac{\partial^2\phi}{\partial x^2} \approx \sum_{j=0}^{n_i} b_{j-\frac{1}{2}} \frac{\partial\phi}{\partial x} \tag{4.12}$$

To calculate the flux derivatives, simple arithmetic averages of the flow variables are used:

$$\phi_{ij} = \frac{1}{2}(\phi_i + \phi_j) \tag{4.13}$$

In the case of the second order derivatives, the averages of the gradients are modified as described in Ref. [92] to suppress the odd-even decoupling. This is:

$$\nabla\phi_{ij} = \frac{1}{2}(\nabla\phi_i + \nabla\phi_j) - \left[\frac{1}{2}(\nabla\phi_i + \nabla\phi_j)\cdot\frac{\mathbf{t}_{ij}}{|\mathbf{t}_{ij}|} - \frac{\phi_j - \phi_i}{|\mathbf{t}_{ij}|}\right]\cdot\frac{\mathbf{t}_{ij}}{|\mathbf{t}_{ij}|} \tag{4.14}$$

where $\mathbf{t}_{ij}$ is the coordinate vector from i to j with components $\{x, y, z\}$. As an example of the discretisation, the derivatives of the shear stress τ_{xy} in the governing equations can be calculated as follows:

$$\frac{\partial(\tau_{xy_{ij}})}{\partial x} = \frac{\partial}{\partial x}\left(\frac{\mu}{Re}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right) \approx \sum_{j=0}^{n_i} b_{j-\frac{1}{2}} \frac{\overline{\mu_{ij}}}{Re}\left(\overline{\frac{\partial u_{ij}}{\partial y}} + \overline{\frac{\partial v_{ij}}{\partial x}}\right) \tag{4.15}$$

where

$$\begin{aligned}
\overline{\mu_{ij}} &= \frac{\mu_i + \mu_j}{2} \\
\overline{\frac{\partial u_{ij}}{\partial y}} &= \frac{1}{2}\left(\frac{\partial u_i}{\partial y} + \frac{\partial u_j}{\partial y}\right) - \left[\frac{1}{2}(\frac{\partial u_i}{\partial y} + \frac{\partial u_j}{\partial y})\cdot\frac{y_{ij}}{|\mathbf{t}_{ij}|} - \frac{u_j - u_i}{|\mathbf{t}_{ij}|}\right]\cdot\frac{y_{ij}}{|\mathbf{t}_{ij}|} \\
\overline{\frac{\partial v_{ij}}{\partial x}} &= \frac{1}{2}\left(\frac{\partial v_i}{\partial x} + \frac{\partial v_j}{\partial x}\right) - \left[\frac{1}{2}(\frac{\partial v_i}{\partial x} + \frac{\partial v_j}{\partial x})\cdot\frac{x_{ij}}{|\mathbf{t}_{ij}|} - \frac{v_j - v_i}{|\mathbf{t}_{ij}|}\right]\cdot\frac{x_{ij}}{|\mathbf{t}_{ij}|}
\end{aligned}$$

4.4 Spatial Discretisation of the Turbulence Model

Equation 3.32 which describes the Spalart-Allmaras model can be re-written in similar manner to the Navier Stokes treatment above, after some algebraic manipulation. This means separating the equation into a time derivative, convective and diffusive terms plus an algebraic source term:

$$\frac{\partial \tilde{\nu}}{\partial t} + \overbrace{\frac{\partial(H)}{\partial x} + \frac{\partial(I)}{\partial y} + \frac{\partial(J)}{\partial z}}^{ConvectiveTerms} = \overbrace{\frac{1}{Re}\left(\frac{\partial(H^d)}{\partial x} + \frac{\partial(I^d)}{\partial y} + \frac{\partial(J^d)}{\partial z}\right)}^{DiffusiveTerms} + \overbrace{K}^{SourceTerm} \tag{4.16}$$

where

$$H = u\tilde{\nu}, \qquad I = v\tilde{\nu}, \qquad J = w\tilde{\nu}$$

$$H^d = \frac{\nu + \tilde{\nu}}{\sigma}\left(\frac{\partial \tilde{\nu}}{\partial x}\right), \qquad I^d = \frac{\nu + \tilde{\nu}}{\sigma}\left(\frac{\partial \tilde{\nu}}{\partial y}\right), \qquad J^d = \frac{\nu + \tilde{\nu}}{\sigma}\left(\frac{\partial \tilde{\nu}}{\partial z}\right)$$

The algebraic source term K can be divided into four components as follows:

$$K = K_1 + K_2 + K_3 + K_4 \tag{4.17}$$

$$\begin{aligned}
K_1 &= C_{b1}\tilde{S}\tilde{\nu} \\
K_2 &= \frac{C_{b2}}{\sigma Re_\infty}\left[\left(\frac{\partial \tilde{\nu}}{\partial x}\right)^2 + \left(\frac{\partial \tilde{\nu}}{\partial x}\right)^2 + \left(\frac{\partial \tilde{\nu}}{\partial x}\right)^2\right] \\
K_3 &= -\frac{C_{w1}f_w}{Re_\infty}\left[\frac{\tilde{\nu}}{d}\right]^2 \\
K_4 &= \tilde{\nu}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)
\end{aligned}$$

The turbulence model is treated in a similar way to the Navier-Stokes equations with the convective terms discretised using the same upwind scheme as for the inviscid terms, with the left and right sides of the Riemann problem as in Eq. (4.10). The diffusive terms are discretised using a central difference scheme in the same way as the viscous fluxes of the mean-flow equations in Eq. (4.14) and the source term is evaluated at each of the star points. With the addition of the turbulent transport equation, the discretisation of the Navier-Stokes system of Eq. (3.1) becomes:

$$\frac{\mathrm{d}\mathbf{w}_i}{\mathrm{d}t} = -\sum_{j=0}^{n_i}\Big(b_{j-\frac{1}{2}}\left(\mathbf{f}_{j-\frac{1}{2}} - \mathbf{f^v}_{ij}\right) + c_{j-\frac{1}{2}}\left(\mathbf{g}_{j-\frac{1}{2}} - \mathbf{g^v}_{ij}\right) + d_{j-\frac{1}{2}}\left(\mathbf{h}_{j-\frac{1}{2}} - \mathbf{h^v}_{ij}\right)\Big) - \mathbf{K}_i \tag{4.18}$$

where the convective and diffusive terms in the Spalart-Allmaras equation are added to the inviscid and viscous fluxes respectively, and the algebraic source term is included in the right-hand side of the equation.

4.5 Boundary Conditions

Ghost points located outside of the boundary elements are used in order to impose boundary conditions along the geometry. The location of these ghost points is determined by reflecting the flow field points that are close to the surface, as shown in Fig. 4.3.

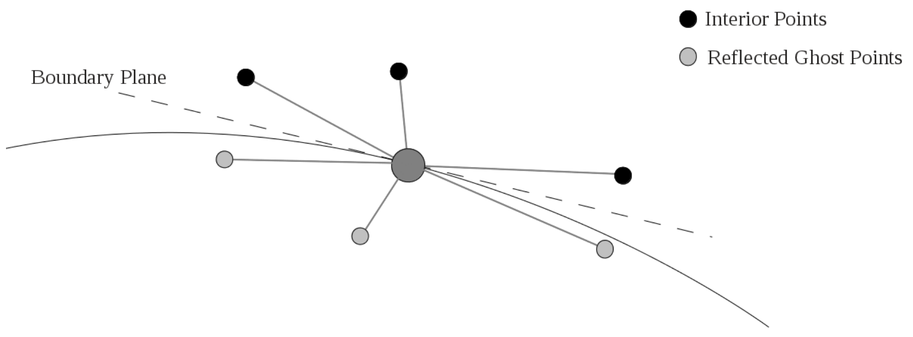


Figure 4.3: Ghost Points

For inviscid flow, "slip conditions" are enforced on solid walls by setting the variables at the ghost points so that the velocity normal to the boundary is zero:

$$\mathbf{u}\cdot\mathbf{n} = 0$$

For viscous flows it is necessary to enforce a "no-slip condition" at the solid walls. This is achieved by setting the tangential velocity on the boundary wall to equal the wall velocity. Considering a static wall, this means $\mathbf{u} = 0$. For the Navier-Stokes equations, the velocity values at the ghost point must have the opposite value to that of its equivalent interior point.

$$u_g = -u_i, \qquad v_g = -v_i, \qquad w_g = -w_i$$

where the subscripts g and i denote the variables at the ghost and interior points, respectively. The pressure and density at the ghost points are determined by first order extrapolation of the values at the interior and surface points. At the far field, the ghost points have the values of the freestream quantities.

For the turbulence model, the turbulent eddy viscosity at the surface walls is set to zero ($\tilde{\nu} = 0$). At the freestream, $\tilde{\nu}$ depends on the freestream kinematic laminar viscosity (ν_∞). Values of $\tilde{\nu}_\infty \leq 0.1\nu_\infty$ are acceptable [87].

4.6 Integration to Steady-State

Once the spatial discretisation has been calculated, the time integration is performed. It is helpful to first write Eq. (4.18) in terms of a residual:

$$\frac{\partial \mathbf{w}}{\partial t} = -\mathbf{R}(\mathbf{w})_i \tag{4.19}$$

As a first integration stage a simple explicit forward difference iteration in pseudo-time (ς) is used to smooth out the initial flow field:

$$\Delta \mathbf{w} = \mathbf{w}^{m+1} - \mathbf{w}^m \tag{4.20}$$

$$\mathbf{w}^{m+1} = \mathbf{w}^m - \Delta\varsigma \mathbf{R}(\mathbf{w}^m) \tag{4.21}$$

where $\mathbf{R}(\mathbf{w})$ is the residual vector, consisting of the right-hand side of Eq. (4.8) and the superscript m denotes the time level in pseudo-time ς.

In order to increase the rate of convergence an implicit integration scheme is used:

$$\frac{\Delta \mathbf{w}}{\Delta\varsigma} = -\mathbf{R}(\mathbf{w}^{m+1}) \tag{4.22}$$

This represents a system of non-linear algebraic equations. In order to simplify the solution procedure the residual $\mathbf{R}(\mathbf{w}^{m+1})$ is linearised as follows:

$$\begin{aligned} \mathbf{R}(\mathbf{w}^{m+1}) &= \mathbf{R}(\mathbf{w}^m) + \frac{\partial \mathbf{R}(\mathbf{w})}{\partial \varsigma}\Delta\varsigma + O(\Delta\varsigma^2) && (4.23) \\ &\approx \mathbf{R}(\mathbf{w}^m) + \frac{\partial \mathbf{R}(\mathbf{w})}{\partial \mathbf{p}}\frac{\partial \mathbf{p}}{\partial \varsigma}\Delta\varsigma && (4.24) \\ &\approx \mathbf{R}(\mathbf{w}^m) + \frac{\partial \mathbf{R}(\mathbf{w})}{\partial \mathbf{p}}\Delta \mathbf{p} && (4.25) \end{aligned}$$

where $\mathbf{p}$ is the vector of primitive variables, $\Delta \mathbf{p} = \mathbf{p}^{m+1} - \mathbf{p}^m$, and $\frac{\partial \mathbf{R}}{\partial \mathbf{p}}$ is the flux Jacobian matrix with respect to the primitive variables at each point. Choosing a Jacobian with respect to primitive variables makes the differentiation simpler.

After linearising the flux residual R^{m+1} in pseudo-time, Eq. (4.22) becomes a system of linear equations to be solved for the primitive variables $\mathbf{p}$.

$$\left(\frac{\mathbf{I}}{\Delta\varsigma}\frac{\partial \mathbf{w}}{\partial \mathbf{p}} + \frac{\partial \mathbf{R}}{\partial \mathbf{p}} \right) \Delta \mathbf{p} = -\mathbf{R}(\mathbf{w}^m) \tag{4.26}$$

where $\frac{\partial \mathbf{w}}{\partial \mathbf{p}}$ is the transformation matrix between conservative and primitive variables.

For the solution of this system to steady-state, an approximate form of the Jacobian matrix with a sparsity pattern from a first order spatial discretisation is used. The linear system is solved by using an iterative solver with a preconditioner based on block incomplete lower-upper (BILU) factorisation [93]. The size of $\Delta\varsigma$ is determined by a local time-step estimate [68] in order to accelerate convergence to a steady-state.

4.7 Iterative Linear Solver

As mentioned before, an iterative method is used to solve the linear system described in Eq. (4.26). The method is based on successive approximations to the solution of a system:

$$Ax = b \tag{4.27}$$

where A is the coefficient matrix, b is the right hand side and x the vector of unknowns. The basic successive approximation approach is worked out by defining a residual vector as $r = b - Ax$. The method used for the solution of the system is the Generalised Conjugate Residual algorithm (GCR) [94, 95], which is shown in Algorithm 1.

Algorithm 1 GCR

```
1:  Compute $r_0 = b - Ax_0$. Set $p_0 = r_0$.
2:  For $j = 0, 1, ...,$ until convergence, Do:
3:      $\alpha_j = \frac{(r_j, Ap_j)}{(Ap_j, Ap_j)}$
4:      $x_{j+1} = x_j + \alpha_j p_j$
5:      $r_{j+1} = r_j - \alpha_j Ap_j$
6:      Compute $\beta_{ij} = -\frac{(Ar_{j+1}, Ap_i)}{(Ap_i, Ap_i)}$, for $i = 0, 1, ..., j$
7:      $p_{j+1} = r_{j+1} + \sum_{i=0}^{j} \beta_{ij} p_i$
8:  EndDo
```

To compute the scalars β_{ij} in the algorithm, the vector Ar_j and the previous Ap_i's are required. In order to limit the number of matrix-vector products per iteration to one, we can proceed as follows: Follow line 5 by a computation of Ar_{j+1} and then compute Ap_{j+1} after line 7 from the relation:

$$Ap_{j+1} = Ar_{j+1} + \sum_{i=0}^{j} \beta_{ij} Ap_i$$

4.8 Preconditioning

Both efficiency and robustness of an iterative solver can be improved by using a technique called *preconditioning* [95]. Put simply, preconditioning is a way of transforming the original linear system into an equivalent system having the same solution, but that is easier to solve with an iterative solver. Numerically, the preconditioner is a non-singular matrix (M) with the property that the system $Mx = b$ is less expensive to solve than the original $Ax = b$. After applying the preconditioner, algorithm 1 becomes:

Algorithm 2 Preconditioned GCR

```
1:  Compute $r_0 = M^{-1}(b - Ax_0)$. Set $p_0 = r_0$.
2:  For $j = 0, 1, ...,$ until convergence, Do:
3:      $\alpha_j = \frac{(r_j, Ap_j)}{(Ap_j, M^{-1}Ap_j)}$
4:      $x_{j+1} = x_j + \alpha_j p_j$
5:      $r_{j+1} = r_j - \alpha_j M^{-1}Ap_j$
6:      Compute $\beta_{ij} = -\frac{(Ar_{j+1}, Ap_i)}{(Ap_i, M^{-1}Ap_i)}$, for $i = 0, 1, ..., j$
7:      $p_{j+1} = r_{j+1} + \sum_{i=0}^{j} \beta_{ij} p_i$
8:  EndDo
```

4.9 Time-Accurate Integration

For time-accurate, unsteady simulations, Eq. (4.22) must be solved in real-time t, such that

$$\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = -\mathbf{R}^{n+1} \tag{4.28}$$

where the superscript n denotes the time level in real-time t. The time integration is done using Jameson's dual time-stepping method [96], in which Eq. (4.28) becomes

$$\mathbf{R}^* = \frac{3\mathbf{w}^{n+1} - 4\mathbf{w}^n + \mathbf{w}^{n-1}}{2\Delta t} + \mathbf{R}^{n+1} = 0 \tag{4.29}$$

where $\mathbf{R}^*$ is defined as the unsteady residual. This is a non-linear system of equations that cannot be solved directly. Instead, Eq. (4.29) can be viewed as a modified pseudo-time steady state problem, which can be solved iteratively for $\mathbf{w}^{n+1}$ by introducing a derivative with respect to the fictitious pseudo-time ς, as explained in Ref. [45]. Finally, the following system for the updates is obtained:

$$\left(\left(\frac{1}{\Delta\varsigma} + \frac{3}{2\Delta t}\right)\frac{\partial\mathbf{w}}{\partial\mathbf{p}} + \frac{\partial\mathbf{R}}{\partial\mathbf{p}}\right)\Delta\mathbf{p} = -\left(\frac{3\mathbf{w}^m - 4\mathbf{w}^n + \mathbf{w}^{n-1}}{2\Delta t} + \mathbf{R}^m\right) \tag{4.30}$$

Chapter 5

Laminar and Turbulent Results

In this chapter, results that test the capabilities of the meshless flow solver to calculate viscous and turbulent flows are presented. The results obtained from the meshless flow solver are compared to computational results from an established CFD solver, and to experimental data whenever it is available. The solver used to compare the meshless results is the Parallel Multi-Block (PMB) code from the University of Liverpool which is a proven research code that has been developed and validated for two decades [97]. Examples of research performed with PMB can be found among others, in [97–100]. For simplicity, the meshless flow solver is referred to from now on as Parallel Meshless or PML.

Four different two-dimensional test cases are selected for this validation study. The first case consists of a NACA0012 aerofoil in steady-state laminar flow. The second, involves simulating the steady-state flow over a circular cylinder at low Reynolds numbers. Finally, the third and fourth tests are based on a RAE2822 aerofoil in steady-state turbulent flow. The test cases are summarised in Table. 5.1

Table 5.1: Flow conditions for the test cases.

Case Number	Flow Type	Geometry	Mach Number	Reynolds Number	Angle of Attack
1	Viscous	NACA0012	0.50	5.0e3	3.00°
2	Viscous	2D Cylinder	0.10	5.0 - 40.0	0.00°
3	Turbulent	RAE2822	0.69	5.7e6	-2.35°
4	Turbulent	RAE2822	0.73	6.5e6	2.79°

The point distributions for all the test cases were obtained from block-structured grids which were created using ICEM-CFD. For all the cases, the stencils for each of the points in the domain are selected by using the connectivity of the structured grids. This

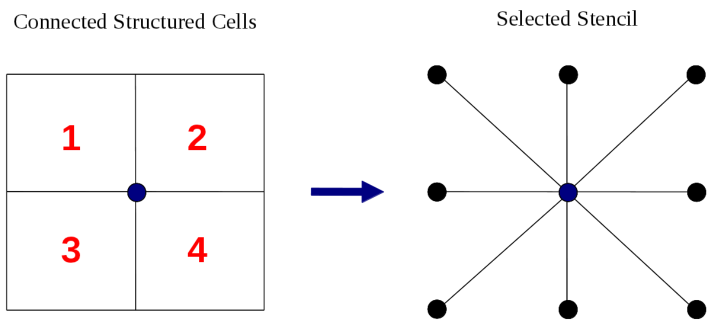


Figure 5.1: Stencil selection for PML cases.

results in stencils with 9 neighbours for internal points, and 6 neighbours for boundary points. Fig. 5.1 shows an example of the stencil selection for internal points.

5.1 NACA0012 Laminar Case

The NACA0012 aerofoil has been used extensively in the literature as a test case for CFD codes. It is used here as a validation case for the implementation of laminar flows into PML. A relatively low Reynolds number of 5000 is selected in order to simulate a laminar regime. For the calculation, subsonic flow conditions are imposed at the farfield, with a Mach number of 0.5 and an angle of attack of 3 degrees. The point distribution used for the case is obtained from a C-type structured grid and contains 516 points along the aerofoil surface and 129 points in the normal direction. The chord length of the aerofoil is set to 1.0 and the first wall spacing for this grid is 1.0×10^{-4}. The total number of points is 86,753 and the point distribution for the case is shown in Figures 5.2 and 5.3.

The calculation was considered to have converged when the L2 norm of the residuals is reduced by six orders of magnitude. As an indication of the computational cost of the method, the calculations took 759 seconds to converge in PML and 587 seconds in PMB, using the same machine. Figures 5.4 and 5.5 show the pressure coefficient and stream-wise velocity flow fields obtained from PML for this case showing highly separated flow over the aerofoil. The results for the surface pressure coefficient obtained from PML and PMB are shown in Fig. 5.6, while Fig. 5.7 shows a comparison of the calculated skin friction over the aerofoil, where the skin friction has been normalized by the reference freestream values. Both these figures show very good agreement between both solvers. Finally, velocity profiles inside the boundary layer, as well as the separation point over the top of the aerofoil are calculated. Figure 5.8 shows the velocity profiles at different positions along the chord direction. The results show almost identical velocity profiles for both codes, and very good agreement in the separation point location with PML calculating it to be at 46.9% of the chord length, while PMB calculates 46.2%.

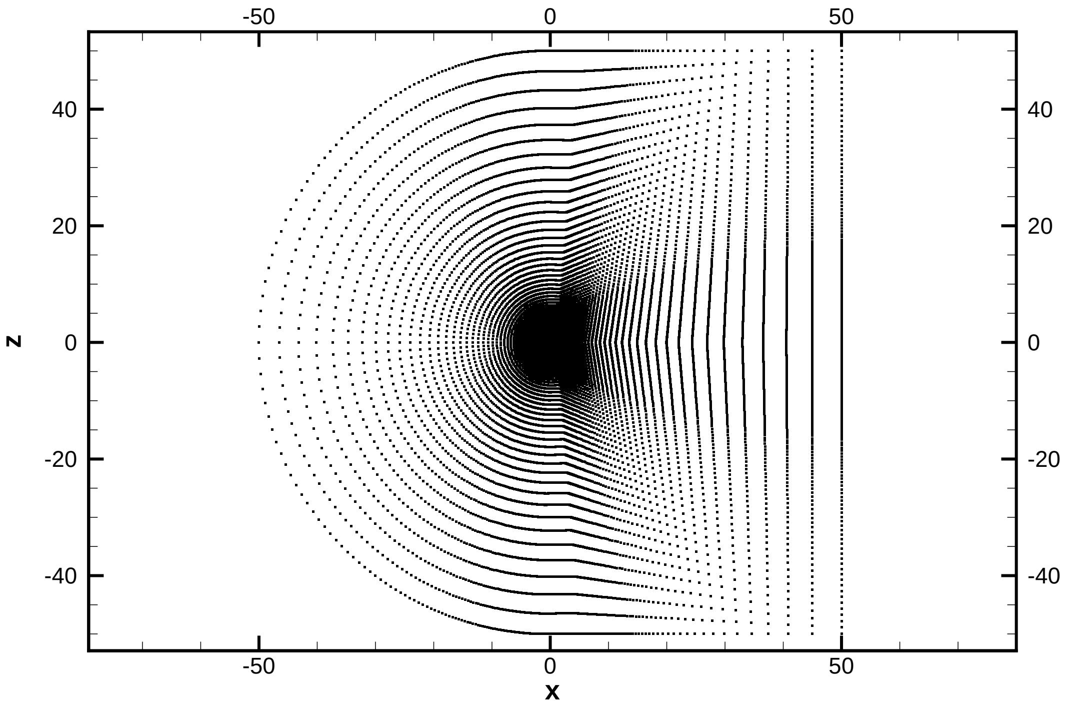


Figure 5.2: Entire domain view of point distribution for NACA0012 case.

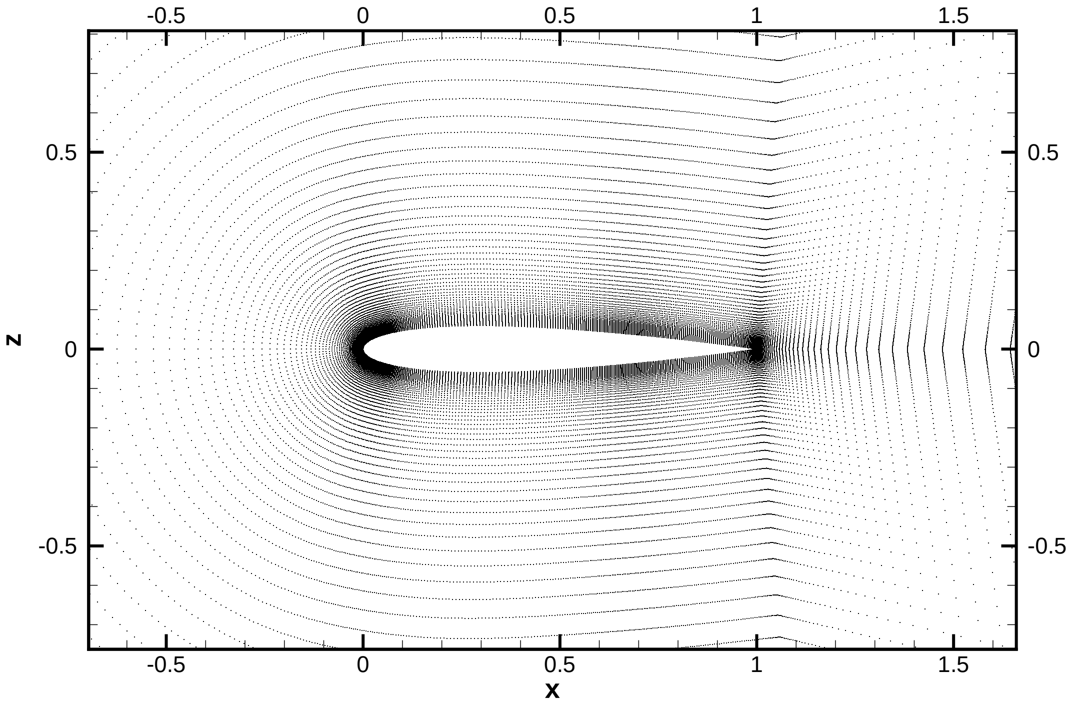


Figure 5.3: Close-up view of point distribution for NACA0012 case (1e-4 first grid spacing).

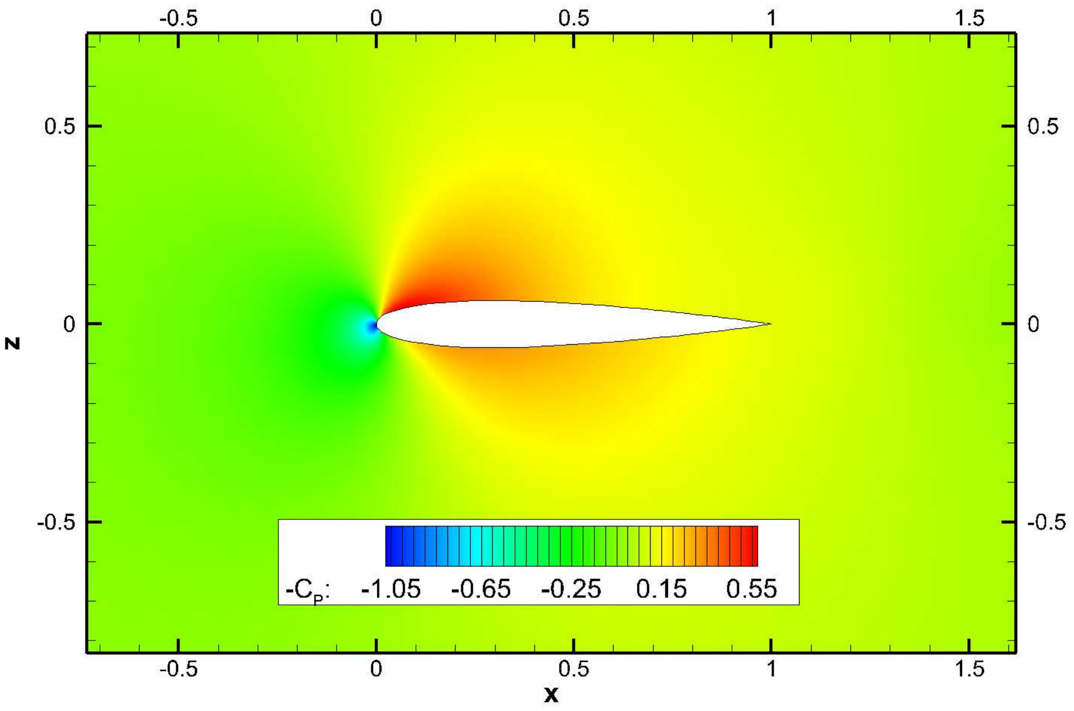


Figure 5.4: Pressure coefficient contours for NACA0012 case.

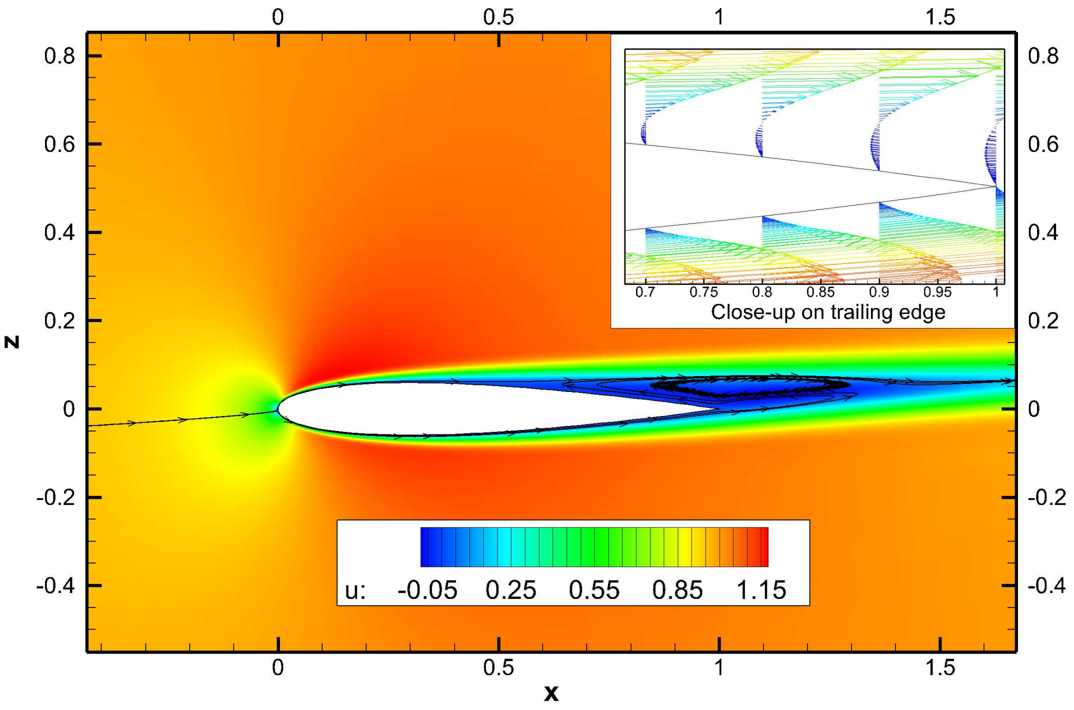


Figure 5.5: Stream-wise velocity contours for NACA0012 case.

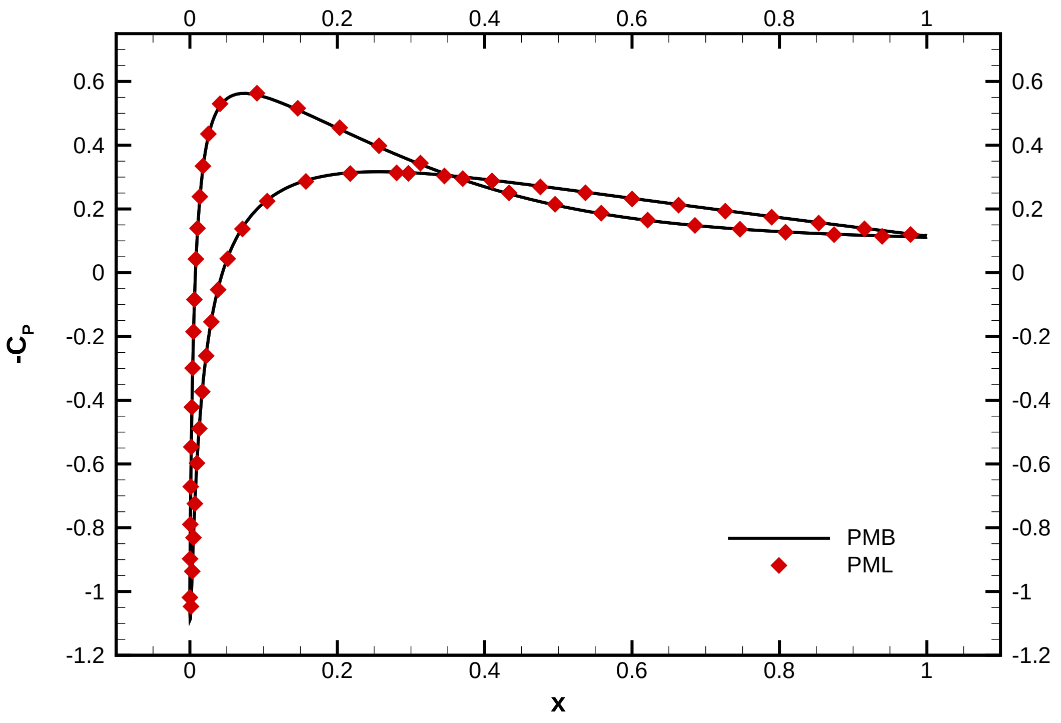


Figure 5.6: Surface pressure coefficient for NACA0012 case.

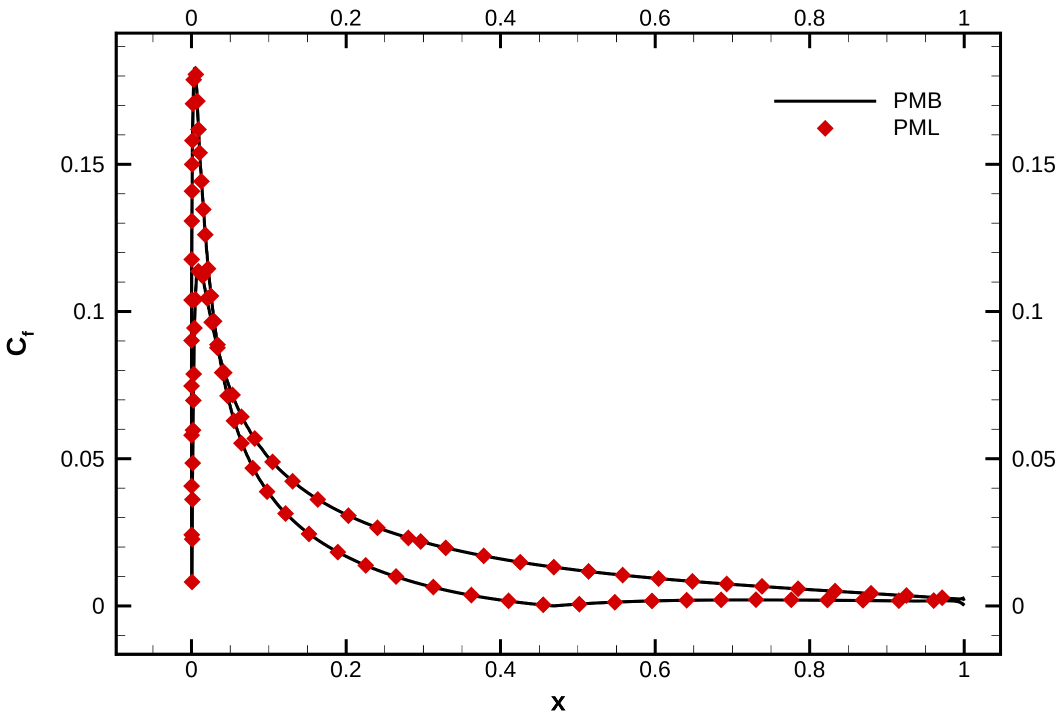
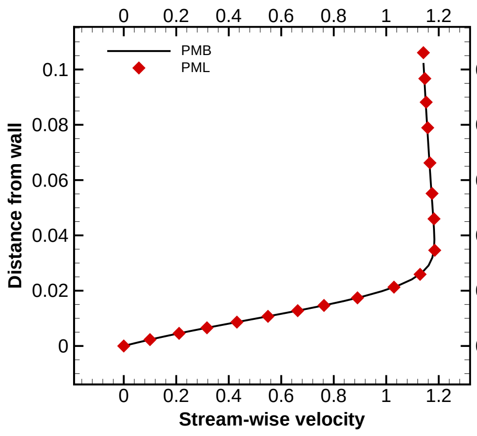
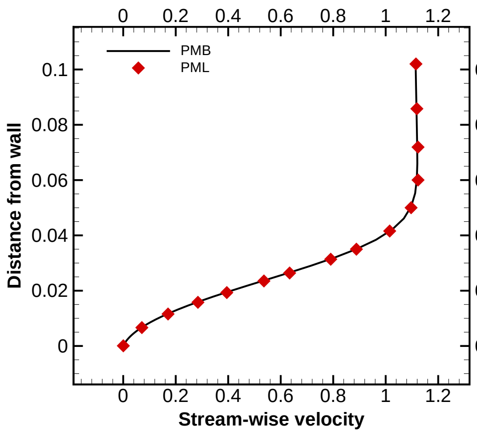


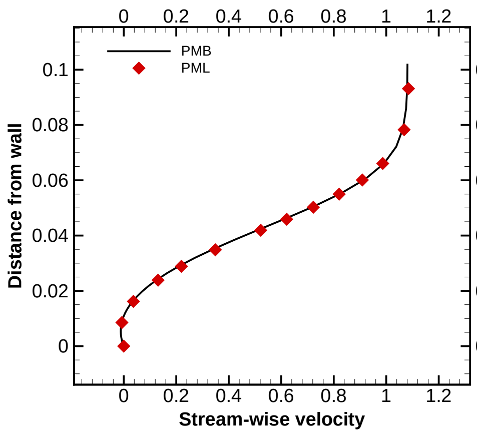
Figure 5.7: Surface skin friction for NACA0012 case.



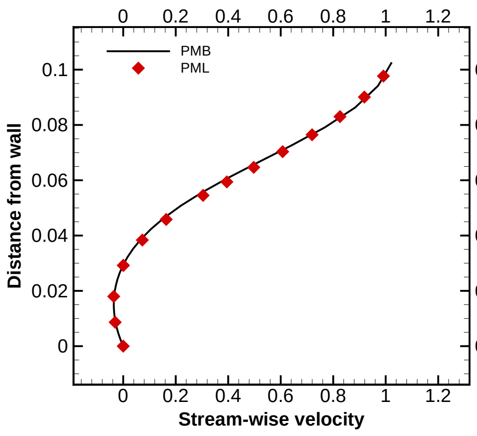
(a) x = 0.2



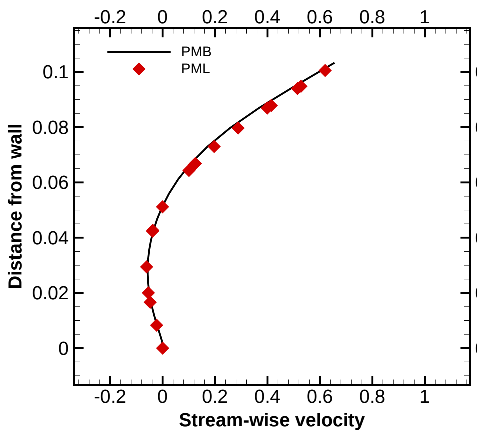
(b) x = 0.4



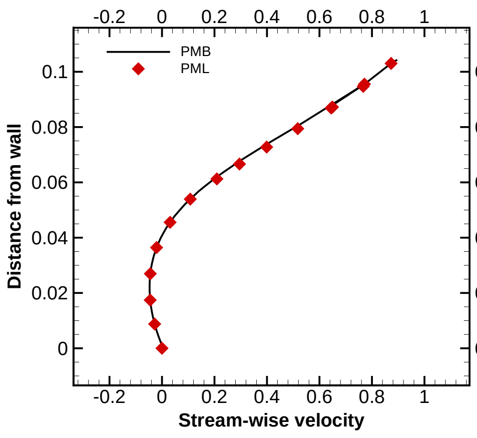
(c) x = 0.6



(d) x = 0.8



(e) x = 0.9



(f) x = 1.0

Figure 5.8: Comparison of stream-wise velocity profiles inside the boundary layer for NACA0012 case.

5.2 Cylinder Laminar Flow

This test case involves simulating the steady-state flow over a circular cylinder in the laminar regime. The flow past circular cylinders has been extensively studied in the literature and it is well known that at low Mach and Reynolds numbers, the flow exhibits steady behaviour with two counter rotating vortices in the wake of the cylinder. Wind-tunnel experiments by Tritton et al. [101] and water-tunnel experiments by Taneda [102] show that for low Mach numbers, the steady flow around the cylinder exists in regimes with Reynolds numbers of up to around 60. After this point, unsteady behaviour with vortex-shedding starts occurring. To avoid any vortex-shedding during the simulations, the case is run with a Mach number of 0.1 and a variable Reynolds number in the range of 5 to 40 (based on the diameter of the cylinder). The angle of attack is set to zero degrees.

The computational domain used for the calculations extends to $40D$ from the centre of the cylinder, where D is the diameter. Three point distributions were generated for this case to perform a grid sensitivity study. All of these were generated from O-type structured grids. The first grid, called *"fine"* contains 488 points along the cylinder surface and 261 points in the radial direction, for a total of 254,753 points, with the first grid spacing set at 1.0×10^{-4}. The other two grids were generated by starting from the *fine* grid and successively deleting every other layer of points, thus ending up with the family of grids summarised in Table 5.2. The point distribution from the *fine* grid is shown in Figs. 5.9 and 5.10.

The results calculated by PML show that as expected for Reynolds numbers under 40, two static, counter-rotating vortices are generated in the wake of the cylinder. This is shown in Fig. 5.11, where the pressure contours and velocity streamlines for the PML solution at a Reynolds number of 26 are plotted. The image in Fig. 5.12 was taken from the experimental tests of [102], at the same flow conditions. Comparing the results from PML to the image, it is clear that the flow structures present in the experiment are predicted well by the solver.

The results obtained from PML for skin friction and drag are compared to experimental data obtained by Williamson [103] on a pressurized wind-tunnel using a buried wired-gauge technique. Figure 5.13 shows the skin friction coefficient versus circle angle on the cylinder, where the skin friction has been again normalized by the reference freestream values. In the figure, the circle angle $\theta = 0$ represents the stagnation point to the left of the cylinder. Good agreement is observed between the simulations and the experimental data. The figures also show that as expected, the solution improves by using finer grids. Even so, PML results for this case show not to be particularly sensitive to the grid density.

Finally, the drag coefficients calculated by PML using the finest grid at different Reynolds numbers are compared to experimental data from [101] and numerical results from [104]. Fig. 5.14 shows that the drag coefficient calculated by PML is in good agreement with both sets of results.

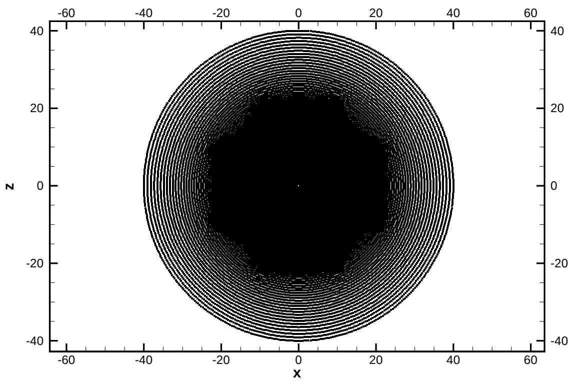


Figure 5.9: Entire domain view of the point distribution for cylinder case.

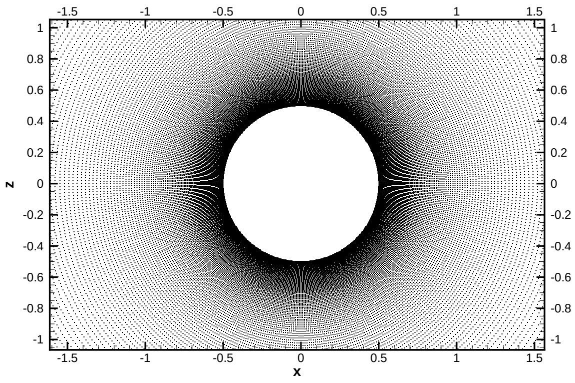


Figure 5.10: Close-up view of the point distribution for cylinder case (1e-4 first grid spacing).

Table 5.2: Grid definition for the two-dimensional cylinder test case.

Grid Name	Description	Total Points	First Grid Spacing
Fine	488x261 edge points	254,736	1.0e-4
Medium	248x131 edge points	64,976	1.0e-4
Coarse	128x66 edge points	16,896	1.0e-4

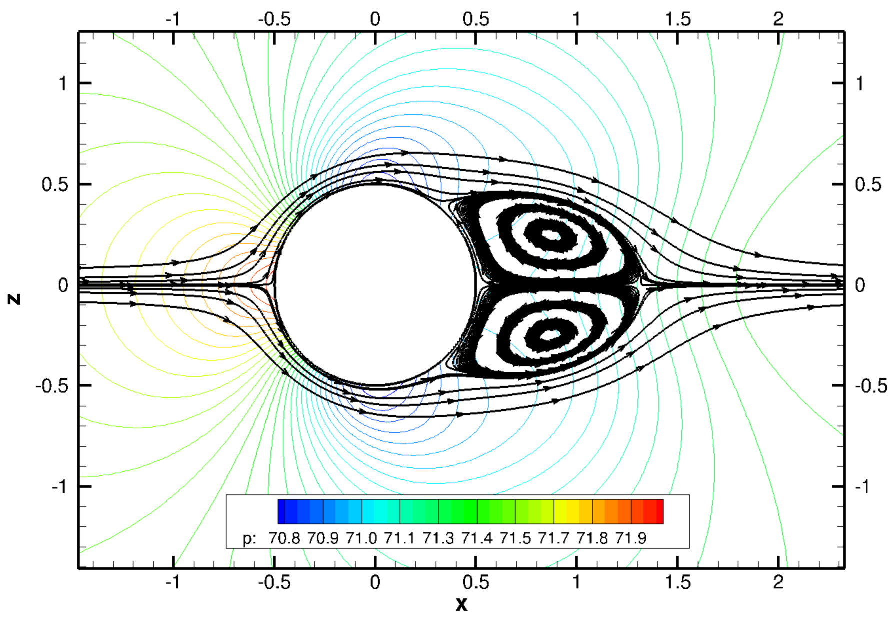


Figure 5.11: Calculated streamlines and pressure contours for two-dimensional cylinder case at Re = 26

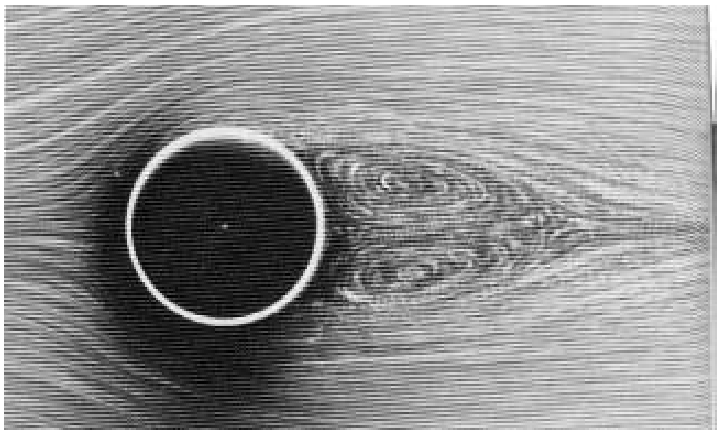


Figure 5.12: Flow streamlines (image from experiment) for two-dimensional cylinder case at Re = 26. Taken from Ref. [102]

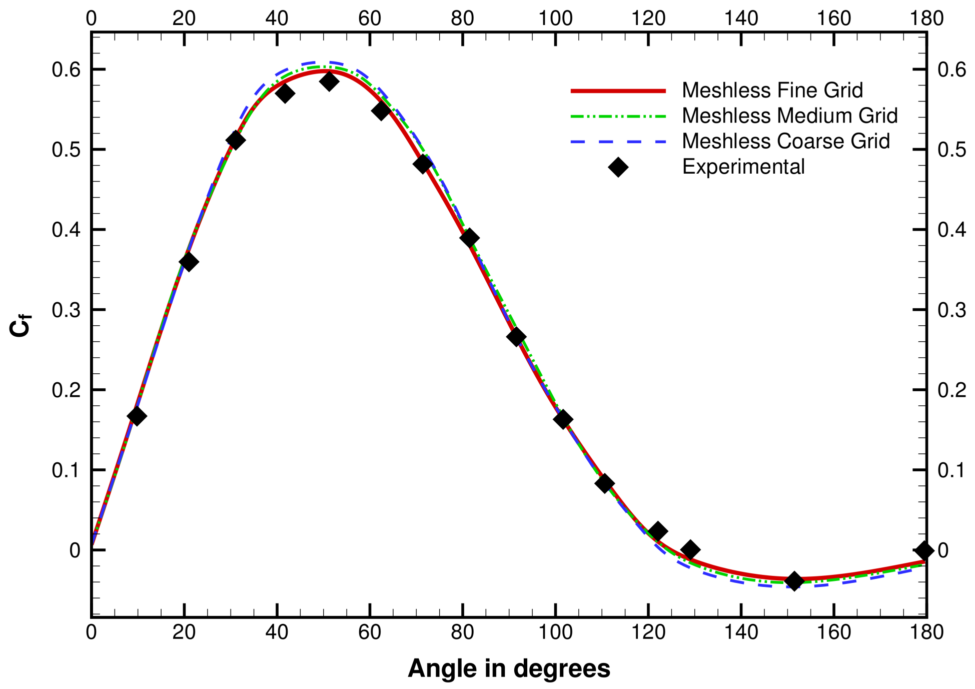


Figure 5.13: Skin friction for two-dimensional cylinder case

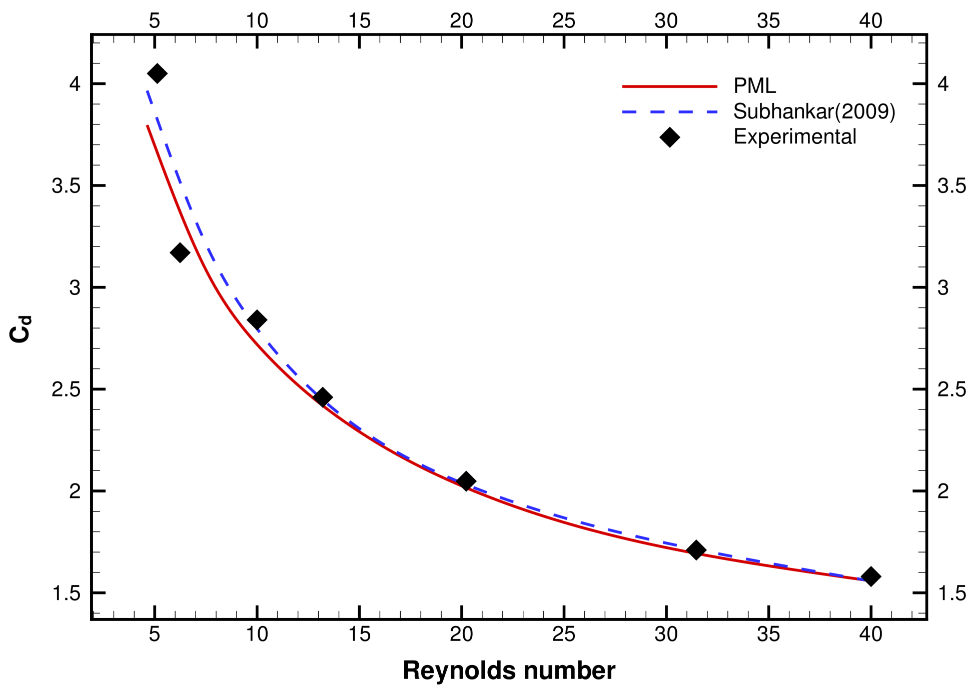


Figure 5.14: Drag coefficient for different Reynolds numbers from 5 to 40 for the two-dimensional cylinder case

5.3 RAE2822 Turbulent Case

To test the turbulent implementation of the meshless solver, the flow over an RAE2822 aerofoil under two different flow conditions was studied. The RAE2822 aerofoil has been used extensively for the validation of CFD codes due to the availability of experimental data for comparison purposes. The two tests used here correspond to cases 2 and 9 from the experimental data base from Ref. [105]. The computations shown here have slightly different flow conditions than the original experimental results to account for the wind tunnel corrections. The corrections made to the flow conditions follow the method from [106]. Details of the original and corrected flow conditions for the cases are shown in Table 5.3. The Spalart-Allmaras model was used with the default parameters in both PML and PMB.

The grid used for the cases was converted from a C-type structured grid and contains 516 points along the aerofoil surface and 129 points in the normal direction. The total number of points is 86,736 and the grids were clustered towards the surface to correctly solve the thin boundary layers. The first grid spacing at the walls for the cases is set to 5.0x10^{-6}. The point distribution is shown in figures 5.15 and 5.16.

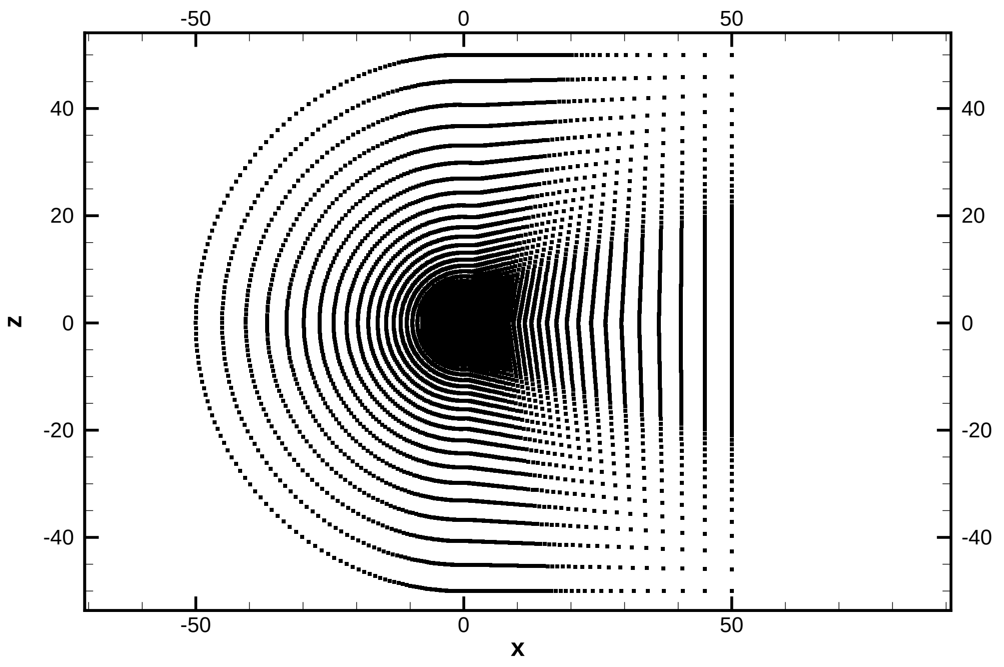


Figure 5.15: Entire domain view of the point distribution for RAE2822 cases

For both test cases, the simulations were started by performing 100 explicit steps, and then continued with implicit integration until a reduction of six orders of magnitude in the normalised mean flow residuals was achieved.

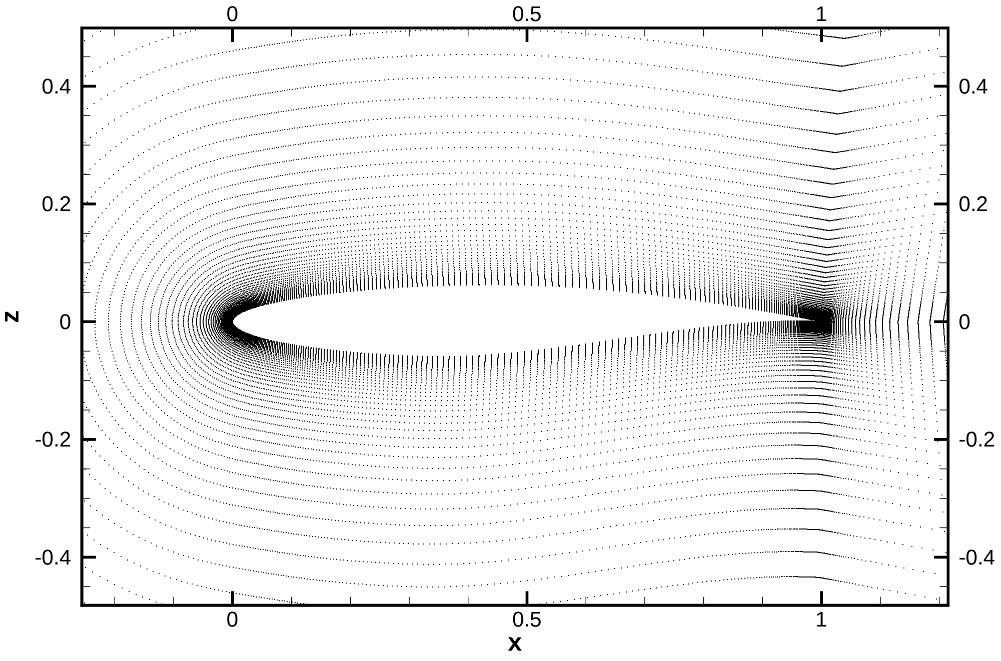


Figure 5.16: Close-up view of the point distribution for RAE2822 cases (5e-6 first grid spacing)

RAE2822 Subsonic

The pressure contours of the flow around the aerofoil are shown in Fig. 5.17, while the stream-wise velocity contours are shown in Fig. 5.18. The surface pressure coefficient calculated from PML is shown in Fig. 5.19, along with results from PMB and experimental data for comparison. There is very good agreement in terms of pressure with PMB and the experimental data. Figure 5.20 shows the maximum calculated turbulent eddy viscosity ($\tilde{\nu}$) at different vertical slices along the stream-wise direction and again, very good correlation is found between both solvers. The velocity profiles calculated from PML and PMB at different positions along the stream-wise direction on the upper surface of the aerofoil are shown in Fig. 5.21. Similar plots comparing the profiles of turbulent eddy viscosity ($\tilde{\nu}$) from both solvers are shown in Fig. 5.22. Even though there are small differences in the eddy viscosity profiles, the agreement of the velocity profile results inside the boundary layer is excellent.

Table 5.3: Flow conditions for the RAE2822 test cases.

	Experimental conditions			Computational conditions		
Case No	α	M	Re	α	M	Re
Case 2	-2.18°	0.676	5.7x10^6	-2.35°	0.685	5.7x10^6
Case 9	3.19°	0.73	6.5x10^6	2.79°	0.734	6.5x10^6

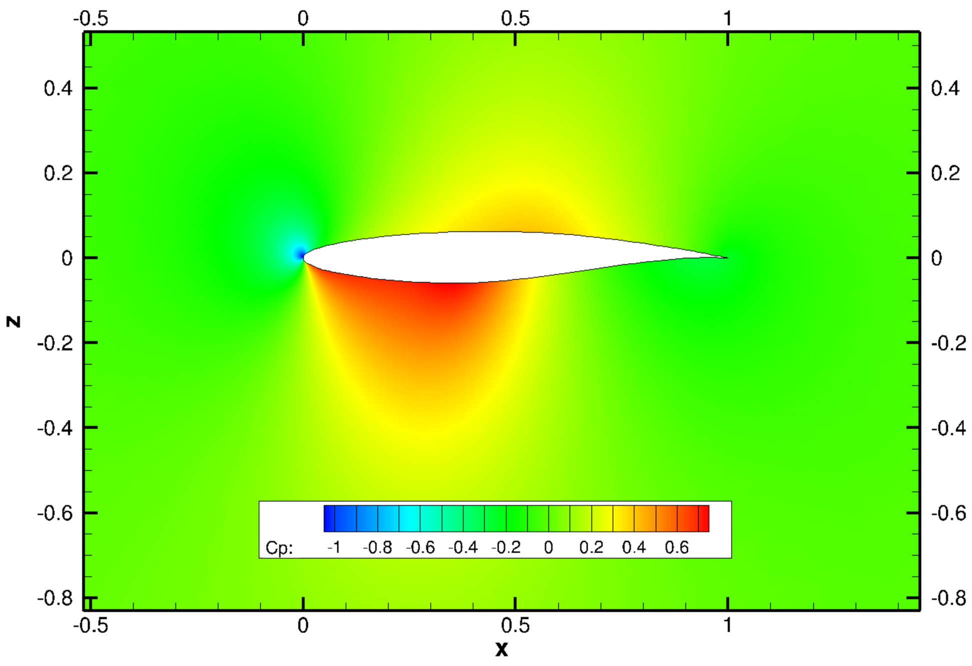


Figure 5.17: Pressure coefficient contours for RAE2822 turbulent flow case with subsonic conditions.

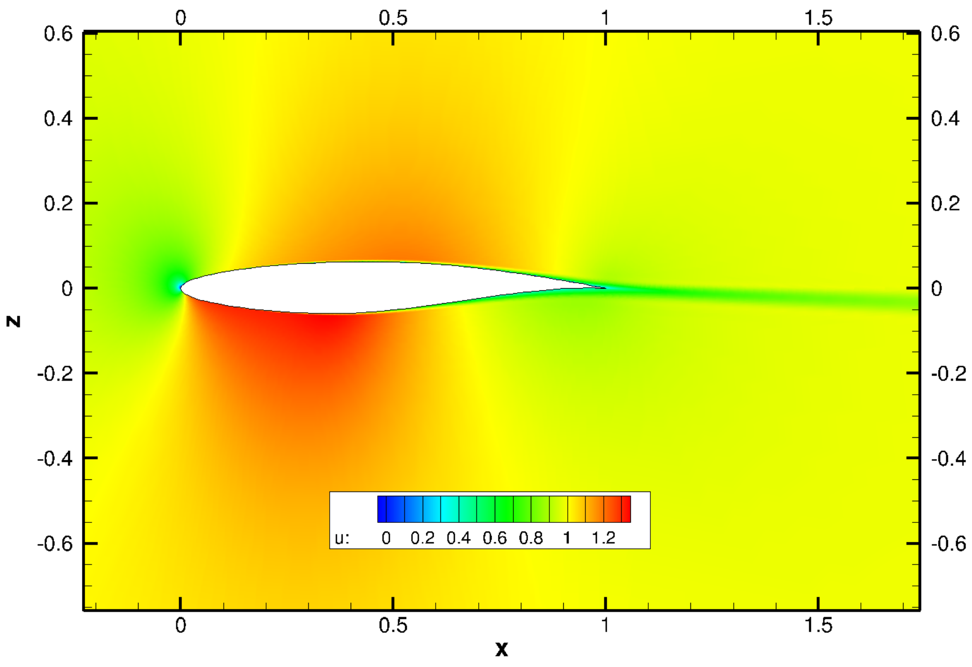


Figure 5.18: Stream-wise velocity contours for RAE2822 turbulent flow case with subsonic conditions.

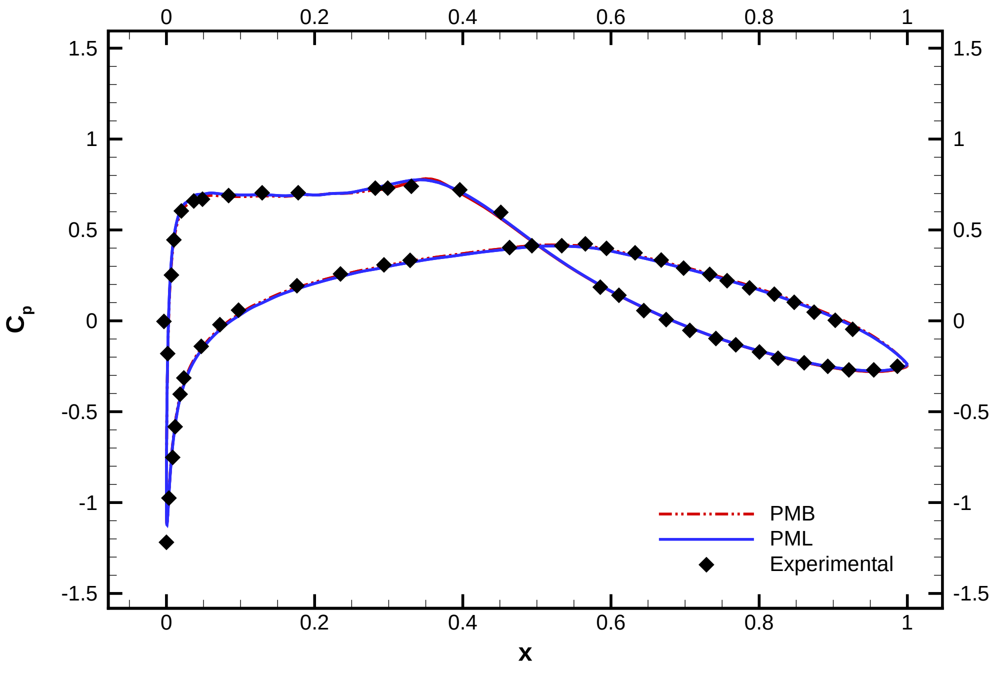


Figure 5.19: Surface pressure coefficient for RAE2822 turbulent flow case with subsonic conditions.

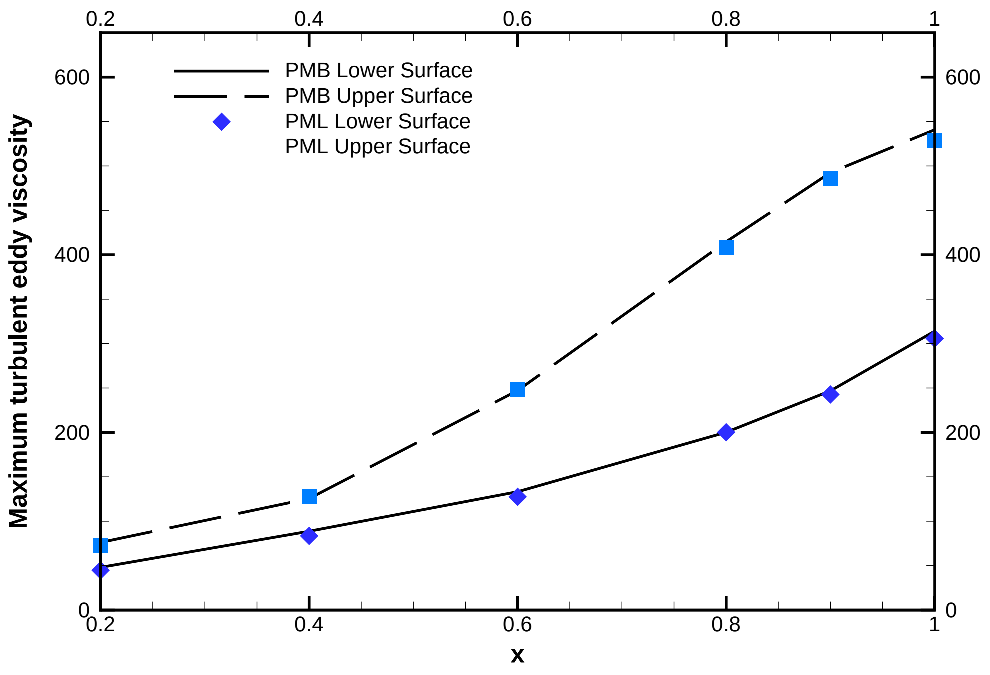


Figure 5.20: Maximum value of turbulent eddy viscosity at different vertical slices along the aerofoil for RAE2822 subsonic case.

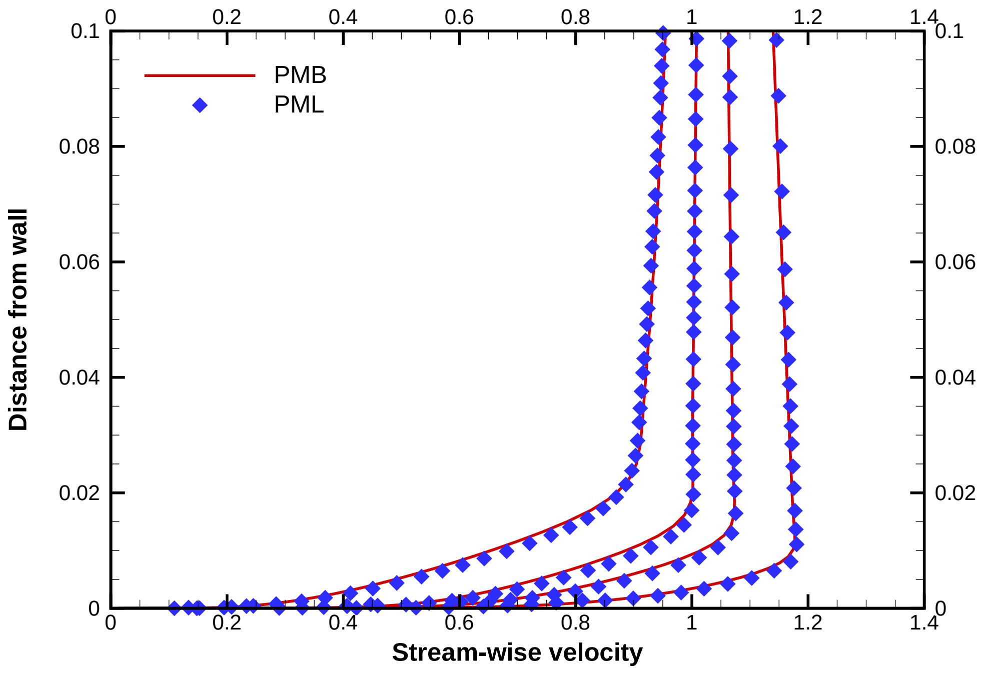


Figure 5.21: Stream-wise velocity at different slices along the RAE2822 upper surface for subsonic case (Slices from left to right: x=1.0, x=0.9, x=0.8 and x=0.6).

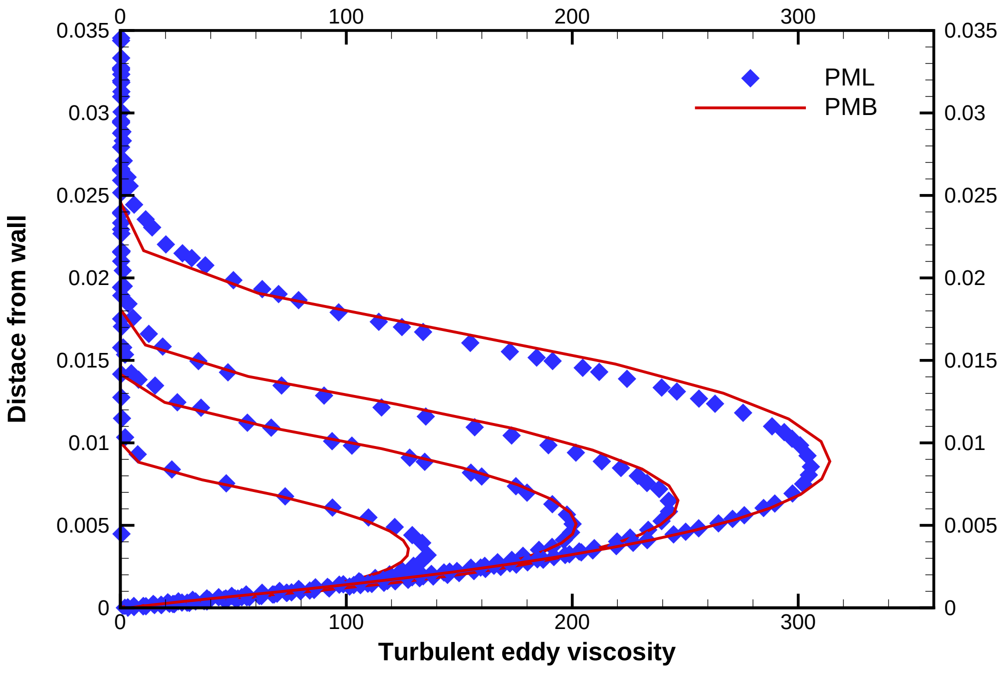


Figure 5.22: Turbulent eddy viscosity at different slices along the RAE2822 upper surface for subsonic case (Slices from left to right: x=0.6, x=0.8, x=0.9 and x=1.0).

RAE2822 Transonic

The calculated flow-field with contours for pressure coefficient, stream-wise velocity and turbulent viscosity are shown in Figs. 5.23, 5.24 and 5.25, respectively. In the figures, a strong shock over the upper surface of the aerofoil can be clearly appreciated, as well as the thin turbulent boundary layer expanding after the shock. The maximum turbulent eddy viscosity calculated from both solvers at different vertical slices is shown in Fig. 5.26, with PML matching the PMB results almost perfectly on the lower surface, and only showing small differences on the upper surface. Profiles of stream-wise velocity and turbulent eddy viscosity inside the boundary layer, at different locations on the upper surface of the aerofoil are shown in Figs. 5.27 and 5.28 respectively. Here, a similar behaviour is found to that of the subsonic case, with small differences found between PML and PMB in the turbulent eddy viscosity profiles, but with good agreement on the velocity profiles. Fig. 5.29 shows the surface pressure coefficients calculated from the meshless solver and PMB compared to experimental data. The results from PML show good overall agreement with the data from PMB and the experimental results, with a well formed shock over the aerofoil. Even though the shape of the shock is predicted well and there are very few differences between the shock calculated from both flow solvers, there is a discrepancy in the location of the shock calculated from both solvers when compared to the experiments. The prediction of the shock strongly depends on the performance of the turbulence model [86]. This fact opens the door for future implementations of different turbulence models into PML.

A summary of the calculated values of the drag, lift and moment coefficients for the two RAE2822 cases is presented in Table 5.4. Even though the results from PML and PMB are very similar, the values obtained from PML are very encouraging as they approximate better the experimental data from Ref. [105], especially at subsonic conditions.

Finally, as a comparison of the computational efficiency of the methods, Table 5.5 shows the calculation times needed to reduce the mean-flow residuals five orders of magnitude for both PMB and PML. During these tests PMB was approximately 40% faster than PML. Even though the finite-volume method used by PMB is more computationally efficient than PML, the meshless method used in this work proves valuable by providing the capability of simulating moving geometries, as it will be demonstrated in Chapter 9.

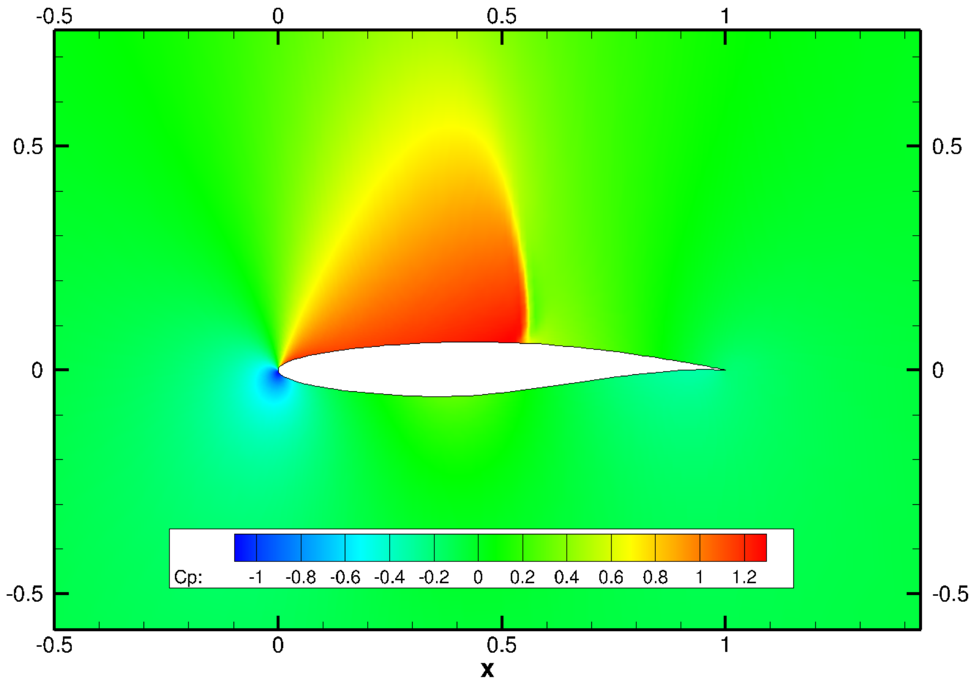


Figure 5.23: Pressure coefficient contours for RAE2822 case with transonic conditions.

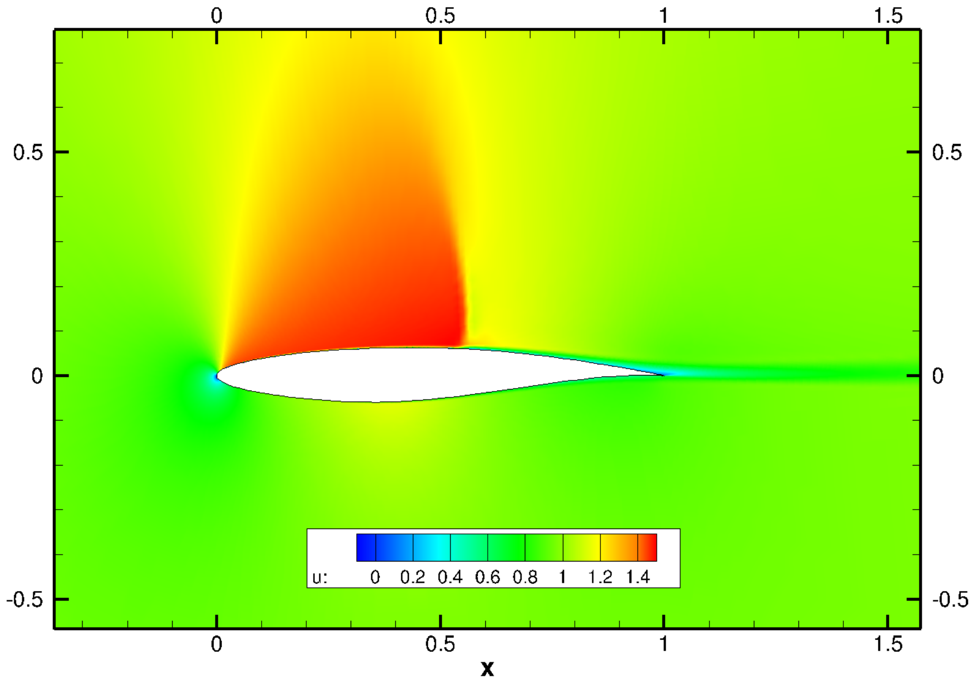


Figure 5.24: Stream-wise velocity contours for RAE2822 case with transonic conditions.

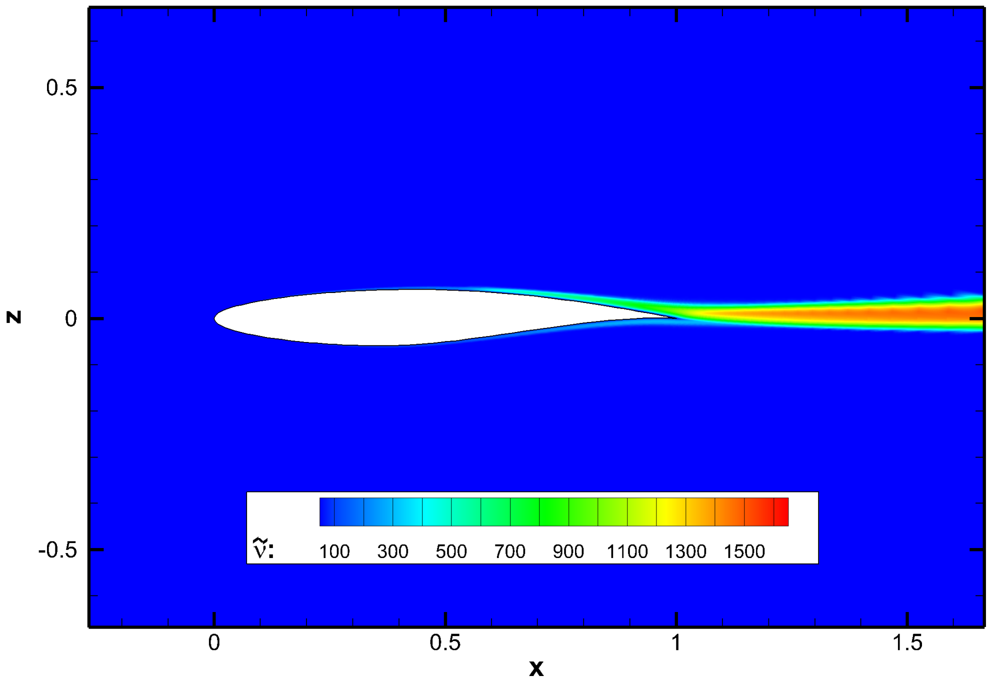


Figure 5.25: Turbulent eddy viscosity contours for RAE2822 case with transonic conditions.

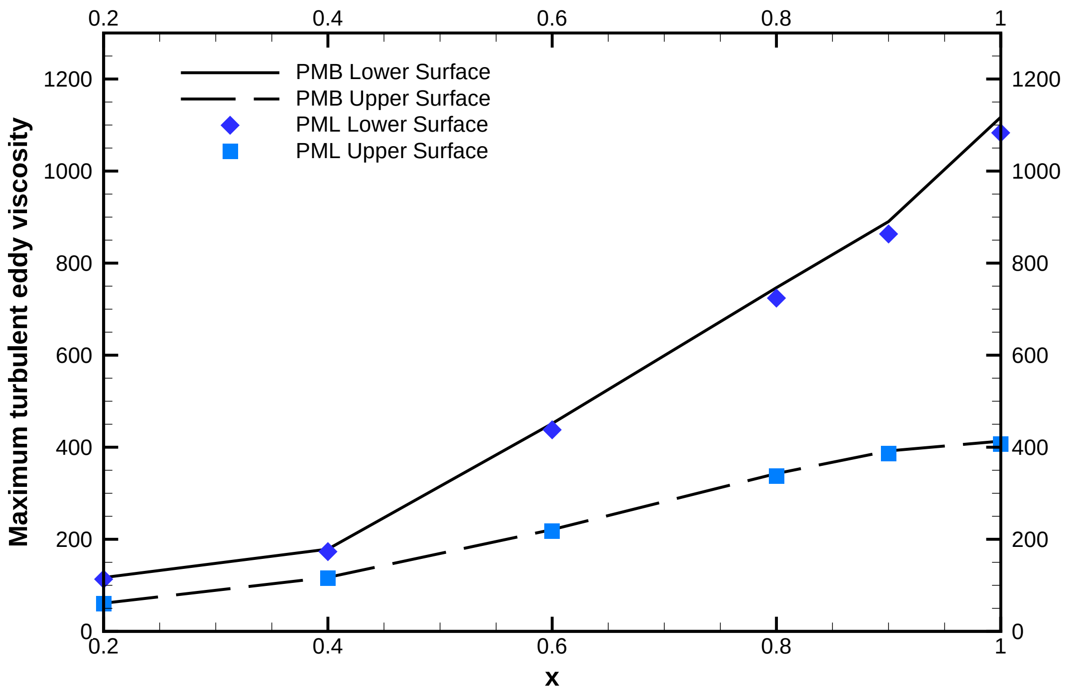


Figure 5.26: Maximum value of turbulent eddy viscosity at different vertical slices along the aerofoil for RAE2822 transonic case.

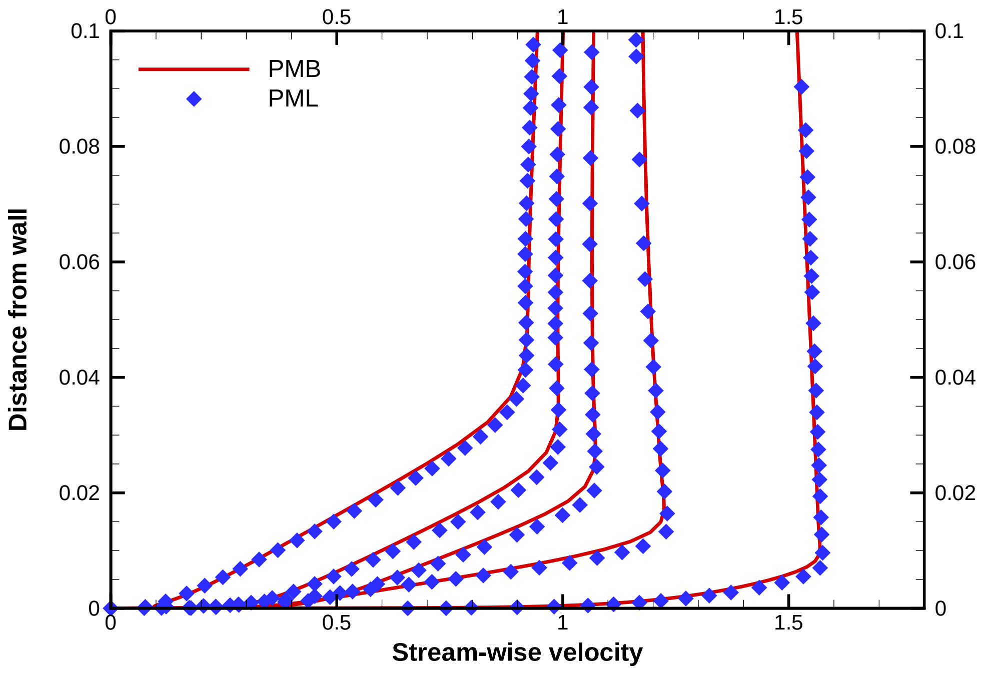


Figure 5.27: Stream-wise velocity at different slices along the RAE2822 upper surface for transonic case (Slices from left to right: x=1.0, x=0.9, x=0.8, x=0.6 and x=0.4).

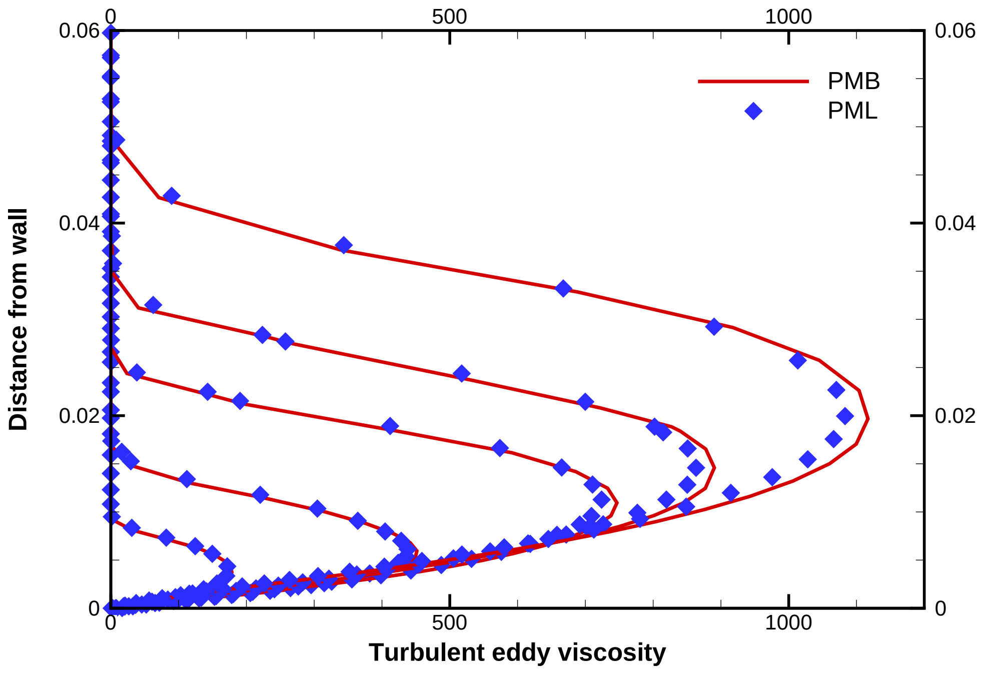


Figure 5.28: Turbulent eddy viscosity at different slices along the RAE2822 upper surface for transonic case (Slices from left to right: x=0.4, x=0.6, x=0.8, x=0.9 and x=1.0).

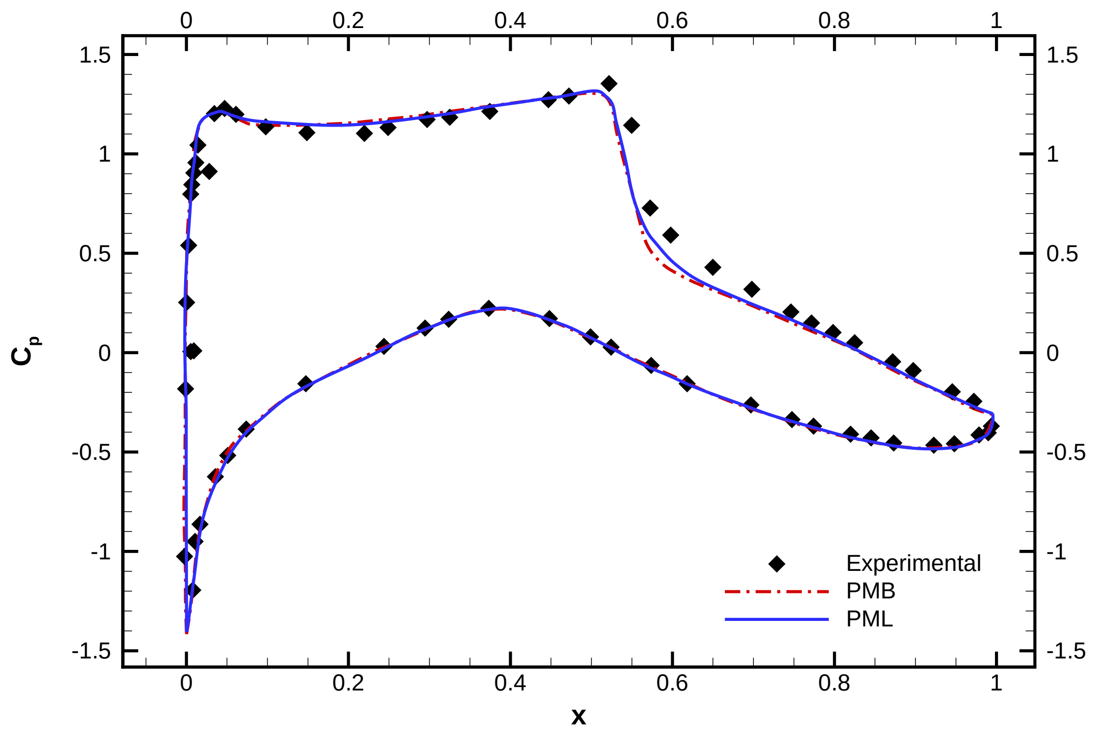


Figure 5.29: Surface pressure coefficients for RAE2822 case with transonic conditions.

Table 5.4: Summary of force and moment coefficients for RAE2822 test cases.

	Subsonic			Transonic		
Study	Cl	Cd	Cm	Cl	Cd	Cm
PML	-0.126	0.0084	-0.045	0.800	0.0162	-0.092
PMB	-0.126	0.0095	-0.076	0.783	0.0170	-0.091
Experimental	-0.121	0.0083	-0.028	0.803	0.0168	-0.099

Table 5.5: Calculation times for RAE2822 test cases.

	Subsonic		Transonic	
	PMB	PML	PMB	PML
Calculation time in seconds	864	1182	940	1212

Chapter 6

Parallel Implementation of the Flow Solver

If the meshless method described in this work is to be used to study real industrial applications its computational efficiency needs to be addressed. During the last three decades, the performance of computing processors increased dramatically, mainly by increasing the clock frequency of the CPUs. In the last few years however, the increase in CPUs clock speeds has been moderate, as power consumption has become the main limiting factor. The main development objective for hardware manufacturers has shifted from trying to achieve faster clock frequencies and now focuses on making more efficient processors with multiple cores per chip. In order to perform the large scale computations required in industrial applications, parallel processing becomes a necessity. In this chapter, a parallel implementation of the meshless flow solver is described. It uses a distributed memory approach and follows single-instruction-multiple-data (SIMD) techniques in which the separate CPUs perform the same computations on different sets of data. The implementation uses explicit message-passing interface (MPI) commands for parallel communication and assigns one set of instructions, called MPI process, to each CPU.

Domain decomposition:

Before the solver is run, the input point distributions are partitioned such that each MPI process is responsible for a distinct portion of the domain. A domain partitioning tool called preDomain was created for this purpose. This tool uses a readily available library called METIS [107] to decompose the domain. Before performing the domain decomposition, preDomain forms connecting edges between all of the points in the domain and their neighbouring points (in the stencil). A graph representing the connectivity of the domain is then created and passed onto the partitioning library. METIS then uses a multilevel recursive-bisection algorithm to partition the graph. During the flow calculations, the computational cost per point of forming the residuals and solving the system is very similar for all points in the domain. Assigning the same number of points to each MPI process, will then result in a near perfect load balance. For this reason, and to improve the parallel efficiency of the method, the criteria for the domain decomposition is to balance the number of points among partitions, and to have the least number of edges cut to decrease communication time. At each MPI process, the domain is then divided into two classes of points: interior points and halo points (Fig. 6.1). Halo points are not local to the processor but are included in local stencils. These points are the communication links between processors across communication boundaries (boundaries of the domain decomposition).

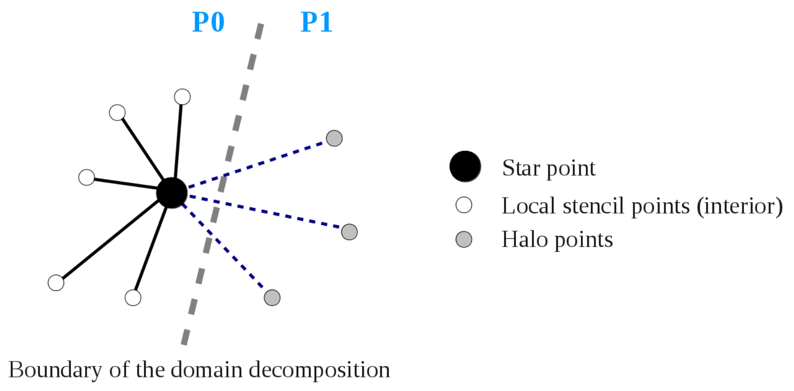

P0
P1
Star point
Local stencil points (interior)
Halo points
Boundary of the domain decomposition

Figure 6.1: Classification of points close to an inter-processor boundary.

In an effort to reduce the parallel overhead associated with the exchange of data, preDomain rearranges the list of points for each MPI process so that the halo points are stored first. This reordering combined with the use of non-blocking commands and pre-allocated buffers for communication, allows for the code to work asynchronously as it will be explained next.

As explained in Section 3.2, the Spalart-Allmaras turbulence model needs the distance of each point in the domain to its nearest solid wall. In parallel, when initialising the computations, all MPI processes interchange the coordinates of their boundary points and elements, and then CPUs work independently to find the nearest solid wall for all of its assigned points.

Solver Stage:

In broad terms, the parallel flow solver works with the following sequence of operations: form the residuals, update the solution and communicate the solution to other MPI processes where needed.

As it was explained in the introductory chapters, the parallel communication introduces an overhead in the computation originating from the latency to start the communication operations, and from the actual communication time. This overhead however, can be reduced by using asynchronous communications. To accomplish this, at each iteration, the code first updates the variables for the points that are needed by other MPI processes. It then issues a non-blocking receive command and sends the needed data across to other CPUs. The use of the non-blocking receive commands, allows for the processes to continue their calculations, while the parallel communications take place. The code immediately continues to update the rest of its points without waiting for any of the messages to be received. At the next iteration, when the remote data is needed, the code checks if the information has been received correctly and if not, the code waits until all communication is completed. In practice, this method should allow for the parallel overhead due to communication to be minimised, as it is hoped that communications finish by the time the processes check if the messages have been received correctly. The diagram shown in Fig. 6.2 shows the algorithm that each of the MPI processes follow for the parallel flow solver.

Parallel Linear Solver:

When using the implicit scheme, the solver performs what can be seen as two distinct types of iteration: an outer loop and an inner loop. In the outer loop, the solver calculates the residuals and stores the Jacobian matrix information for all the points in the domain. In the inner loop, the solution to the linear system of Eq. (4.26) is obtained.

To solve the linear system in parallel, the Jacobian matrix is divided among the processes using the same point-based domain decomposition as before. Each process stores the Jacobian matrix information for its local, as well as its halo points. At each iteration of the linear solver and before the matrix-vector multiplications, the values for the halo points need to be updated. The method again uses asynchronous communication to accomplish this, in an effort to reduce the communication overheads. The calculation of the BILU preconditioner is simplified by only using local points, as this avoids any parallel communication when forming the preconditioner [108]. This simplification can have an effect on the convergence rate of the linear solver which needs to be studied. The computational algorithm for the parallel linear solver can be visualised in the Fig. 6.3. The main bottleneck for the parallel linear solver is expected to be the global communications to reduce the dot-products in steps 2, 5 and 7 of the algorithm. The parallel overheads from the communications needed to perform the matrix-vector operations are reduced by using asynchronous communications, but the overheads caused by the global reduction operations cannot be reduced and means that the CPUs lose their independence with each iteration of the iterative linear solver.

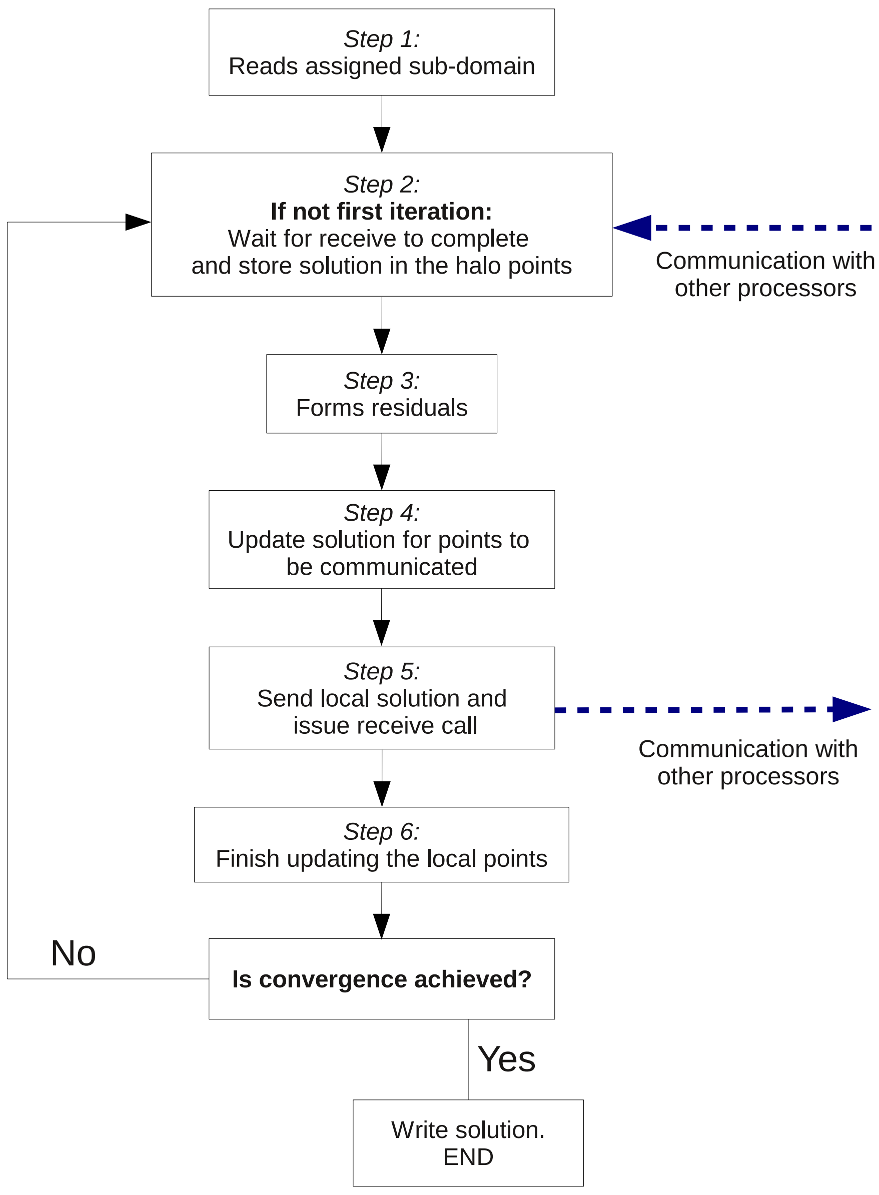

Step 1:
Reads assigned sub-domain
Step 2:
If not first iteration:
Wait for receive to complete
and store solution in the halo points
Communication with other processors
Step 3:
Forms residuals
Step 4:
Update solution for points to be communicated
Step 5:
Send local solution and issue receive call
Communication with other processors
Step 6:
Finish updating the local points
No
Is convergence achieved?
Yes
Write solution. END

Figure 6.2: Algorithm for the parallel flow solver

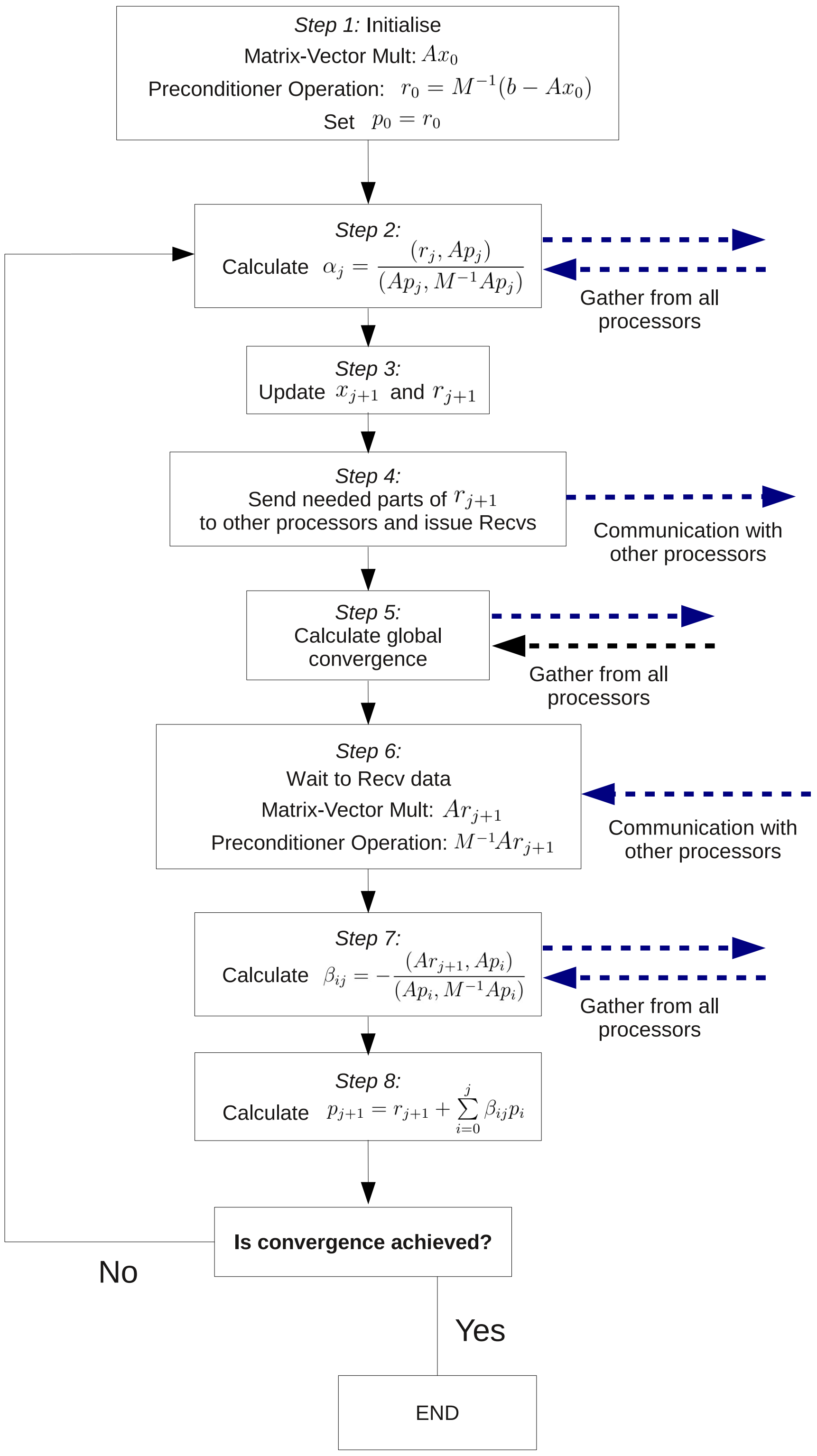

Step 1: Initialise
Matrix-Vector Mult: Ax_0
Preconditioner Operation: $r_0 = M^{-1}(b - Ax_0)$
Set $p_0 = r_0$
Step 2:
Calculate $\alpha_j = \frac{(r_j, Ap_j)}{(Ap_j, M^{-1}Ap_j)}$
Gather from all processors
Step 3:
Update x_{j+1} and r_{j+1}
Step 4:
Send needed parts of r_{j+1} to other processors and issue Recvs
Communication with other processors
Step 5:
Calculate global convergence
Gather from all processors
Step 6:
Wait to Recv data
Matrix-Vector Mult: Ar_{j+1}
Preconditioner Operation: $M^{-1}Ar_{j+1}$
Communication with other processors
Step 7:
Calculate $\beta_{ij} = -\frac{(Ar_{j+1}, Ap_i)}{(Ap_i, M^{-1}Ap_i)}$
Gather from all processors
Step 8:
Calculate $p_{j+1} = r_{j+1} + \sum_{i=0}^{j} \beta_{ij} p_i$
Is convergence achieved?
No
Yes
END

Figure 6.3: Algorithm for the parallel GCR

Chapter 7

Parallel Flow Solver Results

To study the efficiency of the parallel flow solver, two cases were selected. The geometries used are of two well known test cases: the Onera M6 wing and an aircraft configuration known as the DLR-F6. The Onera wing case serves as a good starting point for validating the parallel method and the much bigger aircraft case is used to further asses the capabilities of the parallel flow solver to cope with cases resembling more realistic industrial applications. In both cases, the point distributions used by PML are obtained from structured grids using the same method described in Chapter 5. The parallel flow solver was run on systems ranging from a small cluster of workstations to dedicated HPC machines. The results obtained were compared to data from PMB as well as other published data when available.

7.1 Onera M6 Wing Case

The Onera wing was developed in the 1970s as an experiment to study high Reynolds number flows with complex flow phenomena. Reference [109] contains the original experimental results for this test case, carried out in a pressurised wind-tunnel by the Advisory Group for Aerospace Research and Development (AGARD) of NATO, in 1979. The Onera wing has become a clasic validation case for CFD codes due to its complicated flow physics and availability of experimental data. The wing is a semi-span, defined by a symmetric aerofoil section, a leading edge sweep angle of 30 degrees, an aspect ratio of 3.8, and a taper ratio of 0.562. The point distribution used for this test case is formed of 1,201,075 points with a first wall spacing of $5 \times 10^{-4}c$. Since the objective of this test is to assess the efficiency of the parallel flow solver, the flow conditions are set to inviscid flow instead of turbulent. The freestream Mach number is 0.84 and the angle of attack is set to 3.6 degrees. The test case was also simulated with PMB using the same flow conditions and compared to the experimental data from [109].

The flow solver is run with a varying number of MPI processes ranging from 1 to 32, with each process running on one CPU (core). Figure 7.1 shows the flow solution for the

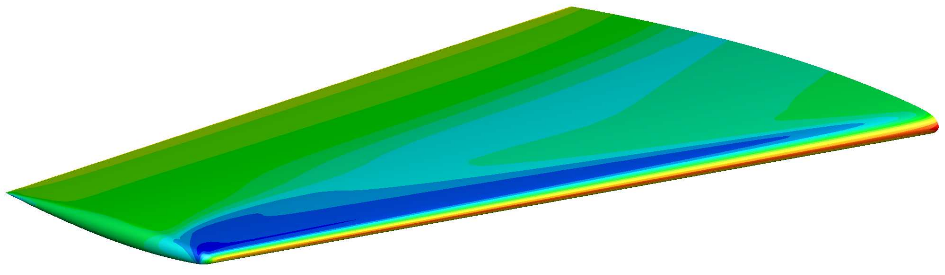
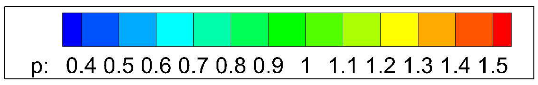
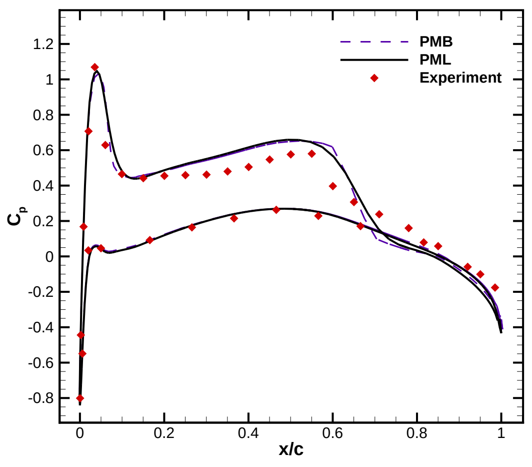



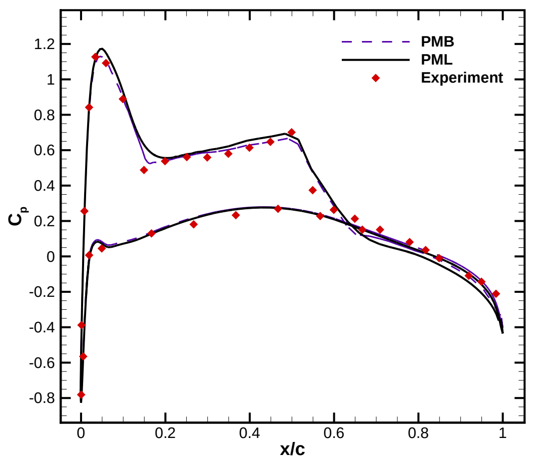
Figure 7.1: Calculated surface pressure for Onera M6 wing case.

test case, where the pressure contours are plotted. Figure 7.2 shows the surface pressure coefficients obtained from the run with 32 cores at six different span-wise locations locations, these are 20%, 44%, 65%, 80%, 90% and 96% of the wing span. These results show very good agreement when compared to PMB calculations. Both codes however have small differences with the experimental data. The biggest discrepancies between the calculations and the experimental data can be appreciated close to the root of the wing (20%). Here, the pressure recovery after the first shock on the upper surface is over-estimated by the code and the location and shape of the second shock is not calculated well. The second biggest difference can be seen at 80%, where two distinct shocks are found in the experiments which are not captured in the calculations. These differences are to be expected as the codes are simulating inviscid flow. Another possible source for the discrepancies close to the root is that during the experiments, an end-plate was located at the root of the wing, while the simulations assume a symmetry boundary condition. Even with these differences, PML proves capable of simulating complex-flow phenomena where strong, variable shocks are present.

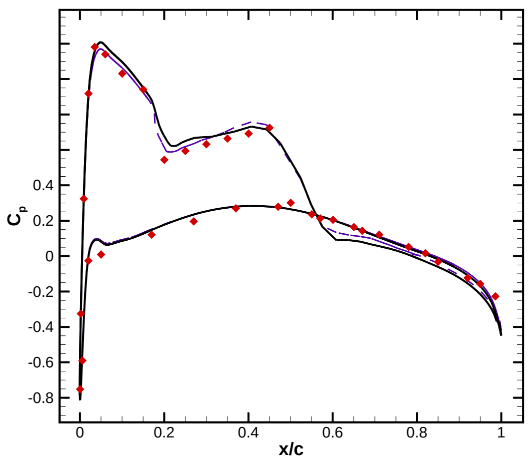
The efficiency of the parallel flow solver in explicit and implicit integration modes is evaluated separately. Even though the explicit integration scheme is only used for a few iterations to start the calculation, evaluating the explicit and implicit schemes independently allows to identify the performance gains of different operations when running in parallel. The parallel speed-up observed in explicit mode takes into account forming the residuals, the parallel communication of the solution, and the updating of the variables in the domain. Any differences observed between the speed-ups of the implicit and explicit schemes are then due to the operations required to solve the linear system described in Section 6.



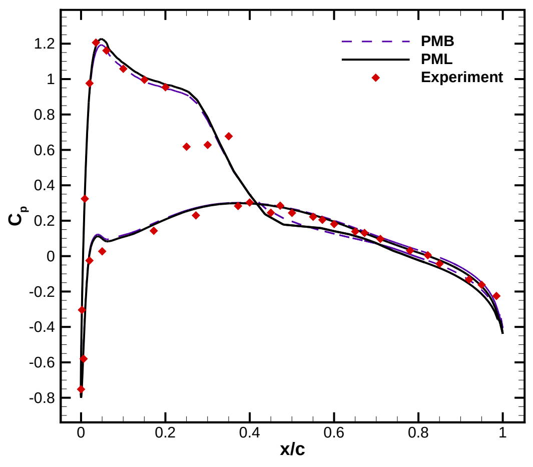
20%



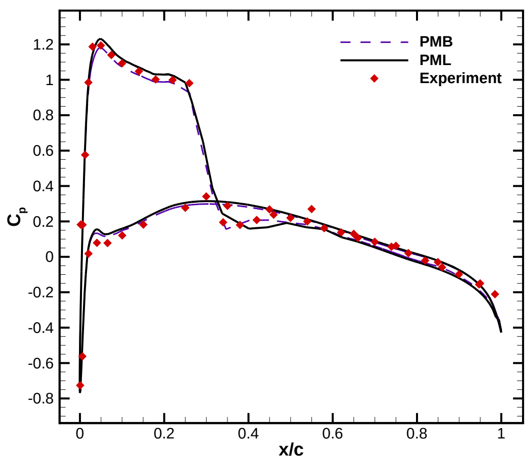
44%



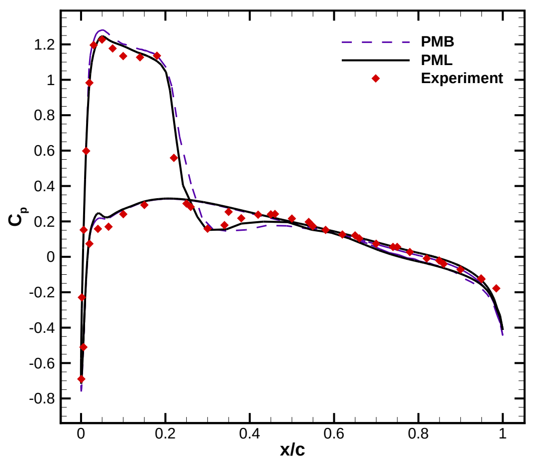
60%



80%

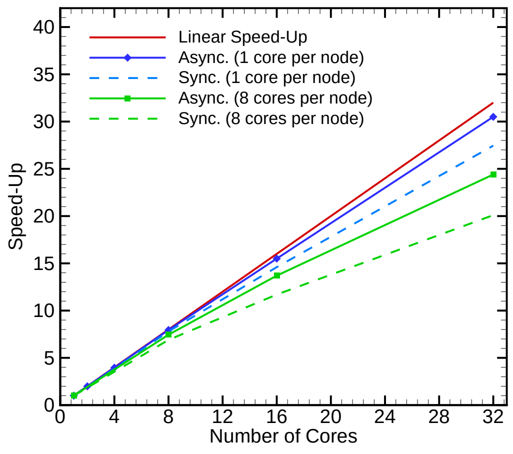


90%

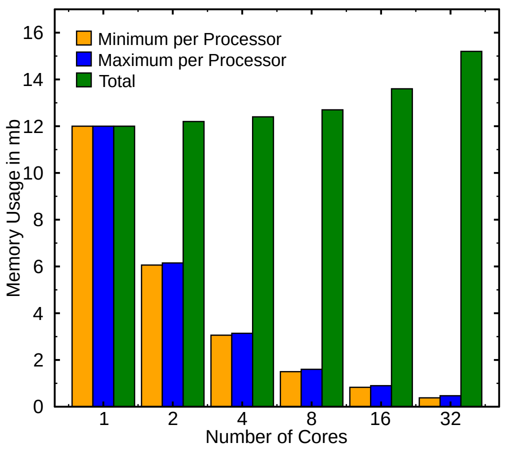


96%

Figure 7.2: Comparison of computed surface pressure coefficients with PMB and experimental data at different locations throughout the wing span for Onera M6 wing at $M_\infty = 0.86$ and $\alpha = 3.06\ ^\circ$.

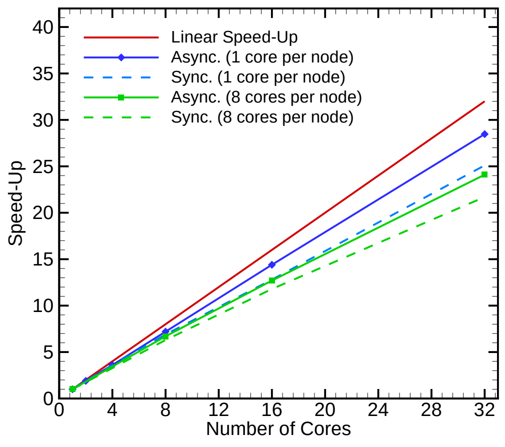


(a) Explicit iteration speed-up.

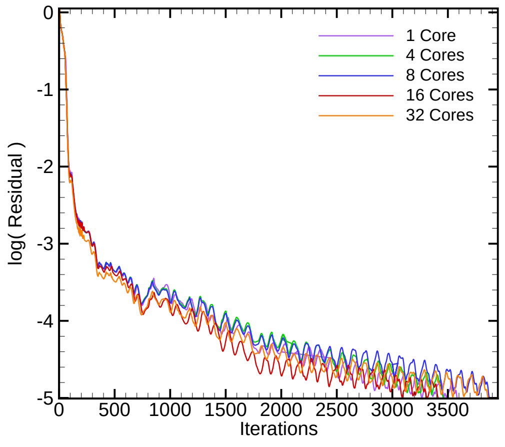


(b) Parallel flow solver memory usage.

Figure 7.3: Efficiency of the parallel flow solver for Onera M6 case using explicit integration with one MPI process per core.



(a) Implicit iteration speed-up.



(b) Parallel flow solver convergence histories.

Figure 7.4: Efficiency of the parallel flow solver for Onera M6 case using implicit integration with one MPI process per core.

Figure 7.3(a) shows the parallel speed-up per explicit iteration of the flow solver obtained in a system with eight cores per computing node, therefore, the results showing 32 cores were obtained with four computing nodes working with all their cores. In the figure, data for both the synchronous and asynchronous modes are shown for comparison. On asynchronous mode, running on one core per node, an almost-linear speed-up is achieved for the explicit iterations up to 32 processes. The advantage of using asynchronous communication in the code is clear. On average for this test case, the asynchronous communication proves 10% faster per iteration than the synchronous one, with the difference increasing as we run in more processes. From the figure, it is clear that the parallel speed-up is negatively affected when running on a higher number

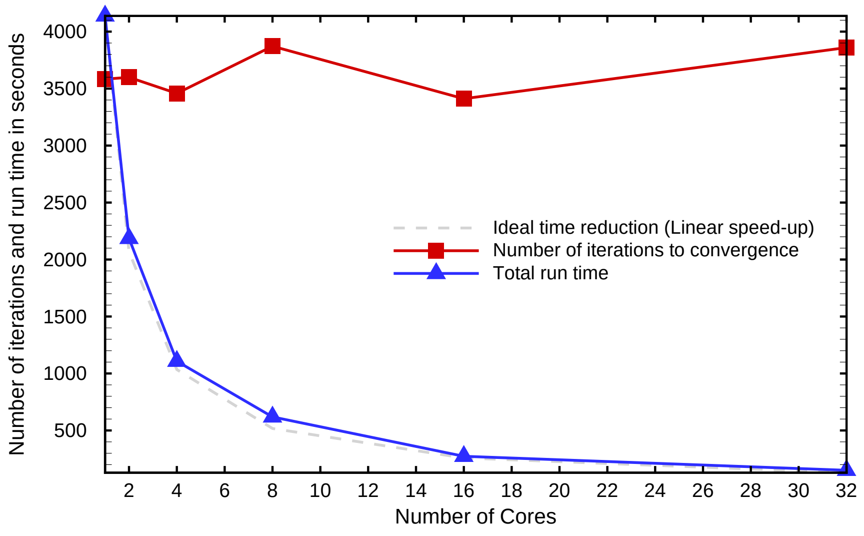


Figure 7.5: Total number of iterations to convergence and total calculation time. Onera M6 case.

of cores per node. This is a consequence of the NUMA architectures where memory access becomes the performance bottleneck of the method. The parallel solver proves to be scalable in terms of memory consumption. This can be seen in Fig. 7.3(b), which shows the usage per core. Here, the differences between the minimum and maximum consumptions show that the problem is well balanced in terms of data storage and that the increase in total memory usage is manageable.

The parallel speed-up for an implicit iteration is shown in Fig. 7.4(a). Here, an iteration means forming the residuals, performing the parallel communications and solving the linear system using four inner-iterations of the linear solver. The figure shows an important speed-up of the implicit method when running in parallel with efficiencies of around 90% when using 32 cores with one core per node. The same behaviour found in the explicit iterations when running more cores per node is observed here as well. During implicit iterations however, the adverse effects of running more cores per node are comparatively lower than when running in explicit mode. Even though the linear solver used during implicit integrations requires two synchronizations per iteration, Fig. 7.4(a) also shows that the asynchronous communications proves faster than the synchronous mode. These gains are mostly found in the outer loop defined in Chapter 6, when forming residuals and updating the solutions. Finally, Fig. 7.4(b) shows the normalized convergence histories for the case. The test case was run in explicit mode for 200 iterations before changing to implicit integration. In this graph, the effect of using a local BILU preconditioner with approximate linear solves is appreciated, as the convergence behaviour is slightly different when running on a different number of cores. This does not necessarily mean that using more CPUs

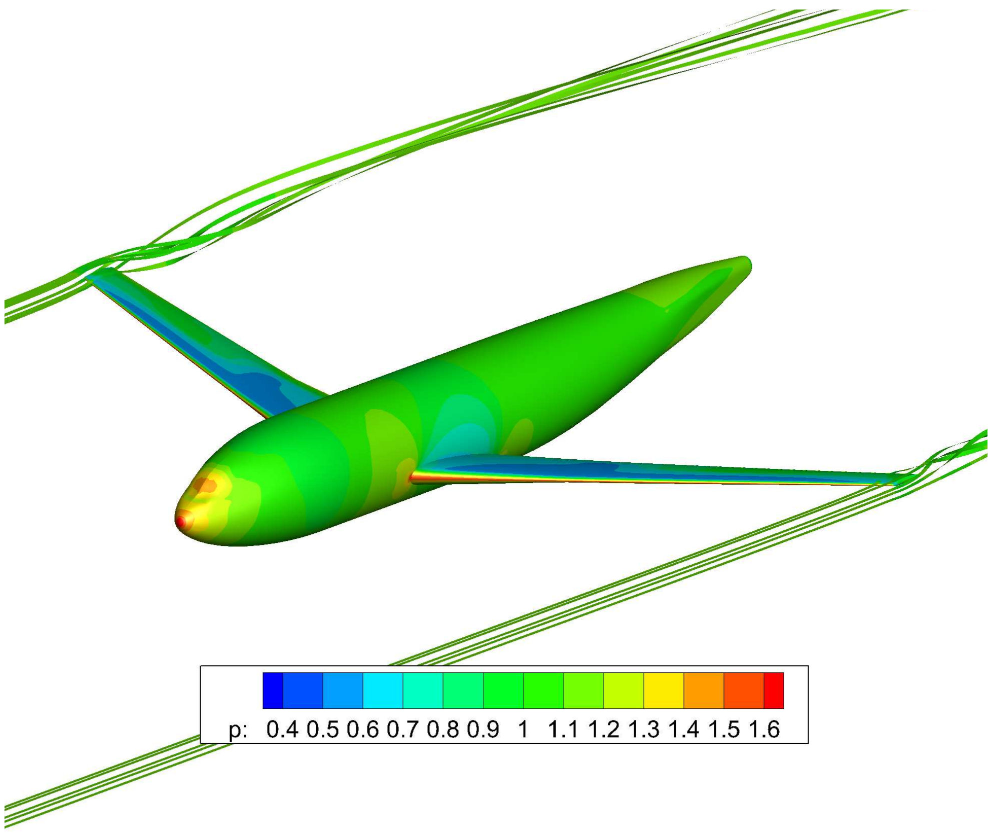
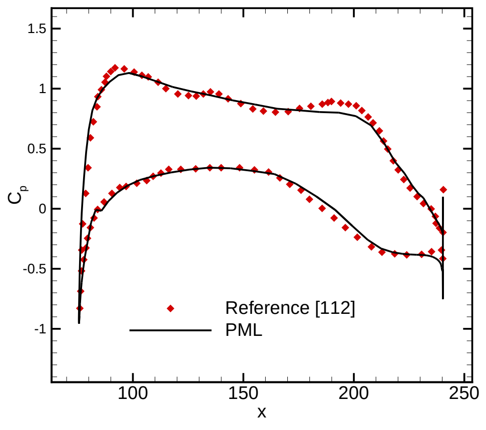


Figure 7.6: Surface pressure contours and flow streamlines for DLR-F6 case.

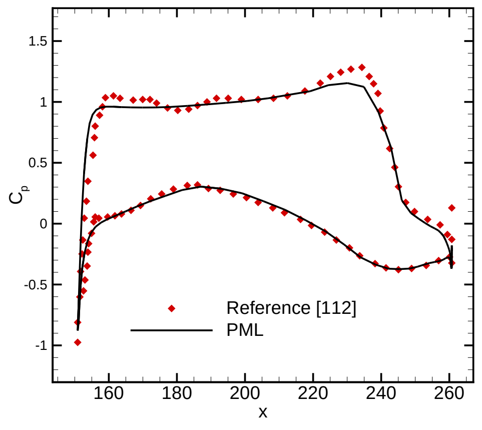
will result in a higher number of iterations needed for convergence. On the contrary, Fig. 7.5 shows that using 4 and 16 MPI processes, PML achieved convergence faster than when using one, two or eight processes. Even though convergence is somewhat affected by using the approximate parallel preconditioner, an important reduction in the total calculation times is achieved.

7.2 DLR-F6 Case

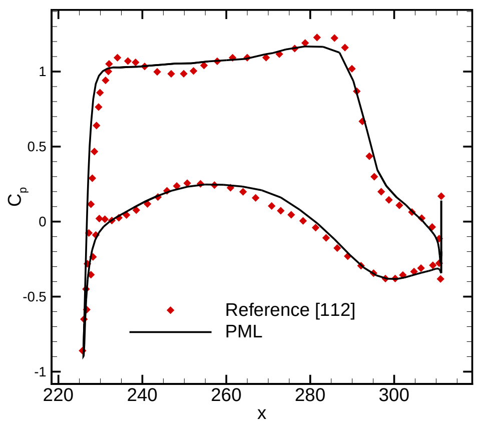
This case is based around a simplified wing-fuselage geometry of a commercial aeroplane and has been previously studied for CFD validation purposes [110, 111]. The flow conditions for the test are transonic inviscid flow with a Mach number of 0.8 and zero degrees angle of attack. The point distribution used for the model contains 45 million points and was created from a structured grid with a first wall spacing of $1 \times 10^{-3}c$, where c is the wing chord at half the span. The point distribution is much bigger than those normally used to solve the Euler equations on such a simple geometry. The high number of points is chosen to test the capabilities of the parallel algorithm to tackle cases resembling realistic applications. The parallel flow solver was run on a different



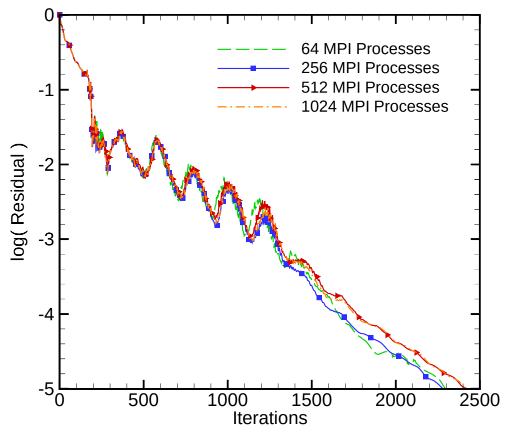
(a) 0.25% of wing span



(b) 0.50% of wing span



(c) 0.75% of wing span

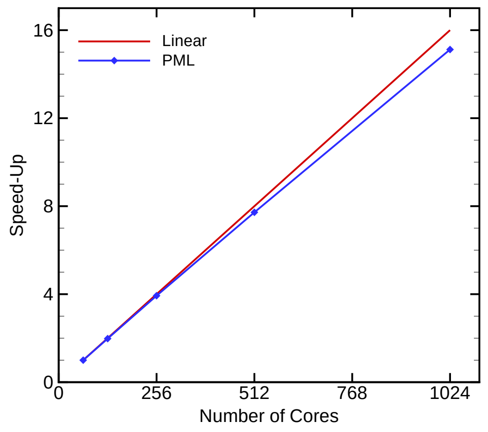


(d) Convergence histories

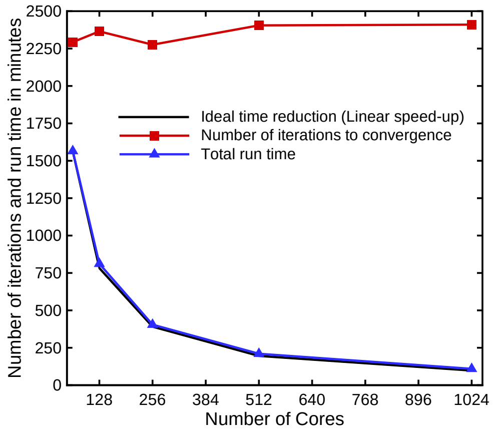
Figure 7.7: Surface pressure coefficients and convergence histories for DLR-F6 case.

number of cores ranging from 64 to 1024 on the facilities of Polaris N8 HPC cluster provided and funded by the N8 consortium and EPSRC. As with the previous case, one MPI process was started on each core.

The flow results for this test case calculated with 512 cores are shown in Fig. 7.6. Here, the surface pressure contours over the fuselage and wings are drawn, and flow streamlines show the wing tip vortices being captured well by the calculation. Figures 7.7(a) to 7.7(c) show the surface pressure coefficient at three different locations along the span of the wing (0.25%, 0.50% and 0.75%). The results from PML are compared to data from [112] which uses an unstructured, finite-volume flow solver for this same test case, calculating inviscid flow at the same conditions. The PML solution agrees well with the published data, with very similar locations for the shocks towards the trailing edge. Small discrepancies are found however, with the magnitudes of the shocks calculated by PML being slightly smaller than those in the reference all along the span of the wing.

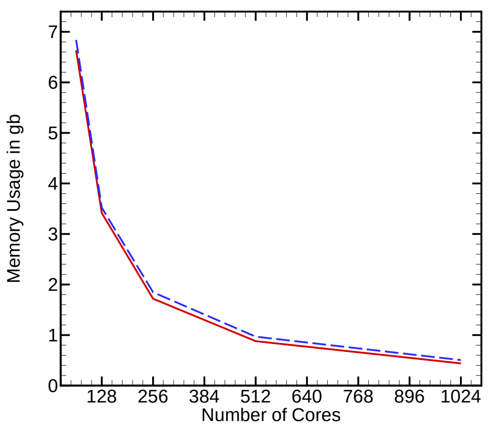


(a) Flow solver speed-up.

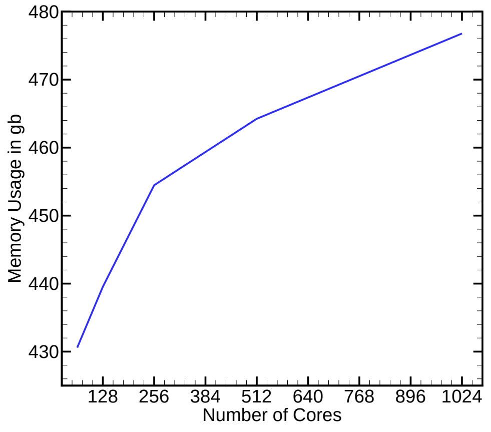


(b) Time reduction.

Figure 7.8: Performance of parallel solver for DLR-F6 case running one MPI process per core.



(a) Minimum and maximum per core.



(b) Total memory usage.

Figure 7.9: Memory usage of the parallel flow solver for DLR-F6 case.

Finally, the parallel performance of the meshless solver is presented. The parallel speed-up is shown in Fig. 7.8 while the memory usage is shown in Fig. 7.9. The parallel flow solver shows very strong scalability in terms of speed, with an almost linear speed-up up to 512 cores and a parallel efficiency of 95% on 1024 cores (compared to 64 CPUs). As with the previous case, the effects of using a local preconditioner are small, with the flow solver converging five orders of magnitude in a similar number of iterations for all the runs on different number of processes, as is shown in Fig. 7.7(d). The memory usage per core shown in Fig. 7.9(a) decreases as expected, but it plateaus at around 1024 cores. This is a result of having to store more communication points, as more processes are used. Even so, the total memory usage increases at a lower rate than the increase in numbers of processes, resulting in a strong scalability in terms of memory, with the total usage increasing by only 11% when going from 64 to 1024 processes.

Chapter 8

Stencil Selection and its Parallel Implementation

8.1 Introduction to the Preprocessor

For multi-body systems, including moving-body simulations, the meshless method in this work needs to select the stencils used by the flow solver to discretise the governing equations. A preprocessing tool has been developed and documented by Kennett et al. in [46,113]. We introduce the stencil selection method here from the original source for the sake of being self-contained. This preprocessor selects stencils automatically from overlapping point distributions associated with bodies, which may be moving relative to one another. The point distributions are obtained from structured or unstructured meshes and the original connectivity of the input grids is used as an aid to select the best stencils from the points available. This way the tool works well for isotropic as well as anisotropic regions, which are usually found close to solid boundaries in CFD problems. Even though the stencils selected by the preprocessor provide a direct input to the flow solver, the preprocessor and the solver are designed as two separate tools that can work in coupled mode for transient moving-body problems.

For brevity, the method described here is for geometry in two dimensions, for details of the implementation in three dimensions refer to [46].

The preprocessor method can be divided into four stages:

1. Check if the surface elements belonging to the solid boundaries of the different input grids intersect in any way. If so, the boundaries need to be redefined accordingly.

2. Detect points falling inside solid surfaces as a result of the point distributions overlap, and exclude (blank) them from the calculation in the flow solver.

3. Select the meshless stencils for all active points.

4. Check that the selected stencils respect the boundaries i.e. to make sure that no stencil contains points that lie on the opposite side of a solid boundary.

More details are given in the following.

Detecting Boundary Overlaps and Redefining Boundaries

A method of detecting solid boundary overlaps, and procedures to redefine the boundaries have been developed and is described in Ref. [113], but this stage has not yet been implemented in the parallel version of this Chapter. In this work, Stage 1 from above is skipped as there is no boundary overlap in the test cases presented.

Blanking Points and Checking Final Stencils

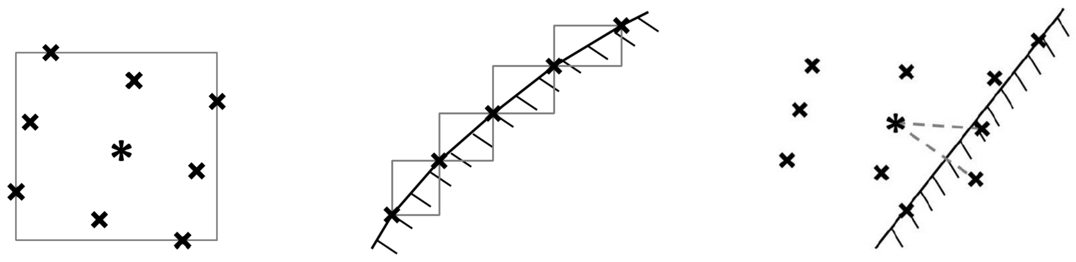
Stages 2 and 4, which correspond to blanking the points internal to solid walls and to checking that the selected stencils respect the boundaries are similar in their implementation. They rely on searching for intersections between the boundary elements and the initial stencils of all the points in the domain. These operations are based on the use of higher-dimensional search trees as discussed in [114]. These search trees allow for fast geometry searches by focusing only on regions that will be of interest.

The procedure works by forming bounding boxes around all initial stencils from the input domains, as in Fig. 8.1(a), and around all boundary elements, as in Fig. 8.1(b). Then, a search tree containing all stencil bounding boxes is formed and traversed with the bounding boxes of boundary elements used as a search region. If intersections between bounding boxes are found, then some of the points in the stencil may lie inside solid walls, or in the case of the final checks in stage 4, some points in the final stencils might not respect the boundaries. To test for this, intersection algorithms are used. Rays are formed between the star point and all neighbouring points in the stencil. The code then looks for intersections between the rays and the boundary elements as in Fig. 8.1(c) to define which of the points in the stencil, if any, are identified to be blanked. In the case of the final checks, these points are removed from the final stencils.

In the case of the blanking operation, this procedure would only identify the points whose stencils cross the solid boundary elements. Points inside solid bodies still need to be removed from the calculation. This is done by a so-called “flood” operation, in which the method blanks the neighbouring points of an already blanked one, as long as they do not cross any solid boundary elements again. The process is repeated until there are no more points interior to solid boundaries to be blanked.

Selecting Candidates

In stage 3, the method selects the stencils for all the points. The operation works by performing a search through each of the points in the domain, looking for intersections


(a) Bounding box for a grid stencil

(b) Bounding box for an internal point stencil.

(c) Blanking points inside boundary elements.

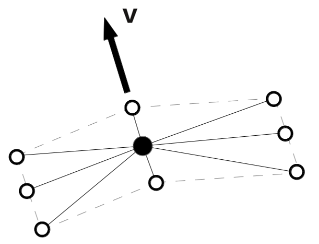
Figure 8.1: Examples of bounding boxes over stencils and boundary elements.

with the initial stencils of other points. The stencil bounding boxes in Fig. 8.1(a) are used as before. All of the points intersecting the stencil of the star point for which the search is being performed are included in a list of possible candidates for the final meshless stencil of that star point.

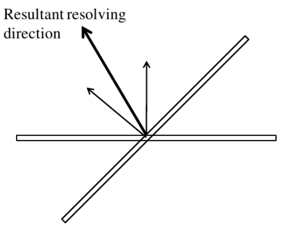
Most CFD grids contain anisotropic regions to capture the rapidly changing flow behaviour without having to greatly increase the total number of grid points. The preprocessor method needs to take into account this arrangement of points when selecting the meshless stencils. To account for the anisotropy of the original stencils, the method defines a resolving vector **v** for each point in the domain. This vector points in the direction where the original stencil is the finest. An example can be seen in Fig. 8.2(a). For each point i, a resultant resolving direction is then formed, using the resolving vectors of the candidate points, as well as the vector of point i, as shown in Figs. 8.2(b) and 8.2(c). Using this resultant direction, a new coordinate system is defined, as shown in Fig. 8.3(a). The basis η is chosen so that the basis vector η_1 lies parallel to the resultant resolving direction, and η_2 lies orthogonal. Setting the origin of the coordinate system to be the star point, the algorithm calculates the quantities a and b as the projections of the stencils onto the newly created coordinate system, as shown in Fig. 8.3(b). It also defines the coefficients ξ_1 and ξ_2 as the coordinates of each of the candidate points in basis η. With these quantities, a merit function ψ is defined, which rates each candidate point in terms of the direction and refinement by balancing the orthogonality of the points chosen (for refinement) and distance. The merit function is given by:

$$\psi = \frac{{\xi_1}^2}{a^2} + \frac{{\xi_2}^2}{b^2} \tag{8.1}$$

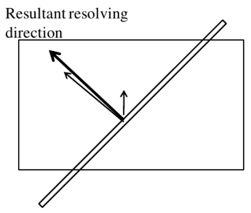
Finally, the method uses this merit function to rate the candidates, and to select the most appropriate ones to form the final stencil by locating them across the quadrants shown in Fig. 8.3(c). In three dimensions the process is similar, with three-dimensional bounding boxes surrounding the stencils and boundary elements.



(a) Resolving vector for a grid stencil

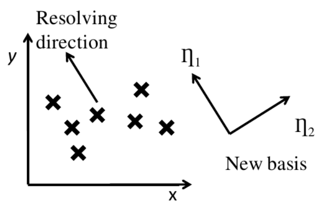


(b) Two anisotropic stencils overlap

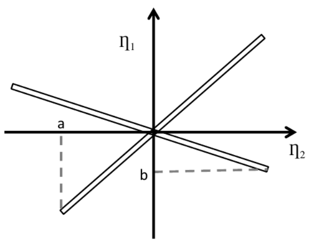


(c) An anisotropic and a regular stencil overlap

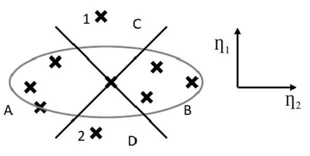
Figure 8.2: Definition of resolving vectors.



(a) Defining new coordinate system



(b) Choosing a and b using stencil projections



(c) Quadrants around resolving direction

Figure 8.3: Definition of new local coordinate system for merit function.

Sorting of Candidates

After the list of candidates is formed for each point in the domain, the stencils are selected according to the merit function ψ described before in Eq. (8.1). For each point in the domain, the method assigns a value of ψ to each of its candidates. It is then required that the list of candidates is ordered according to these values so that the two candidates for each quadrant with the lowest ψ value are selected for the final stencil.

The problem of sorting lists is well known in the field of computer science. Several algorithms have been developed to tackle the problem but a major issue with all of them is that their efficiency depends on the size of the lists to be sorted and how they are arranged initially. For all sorting problems, there is no a-priori information on how expensive the sorting operation will be for a particular sequence of numbers. To illustrate, we use an example in which we use the simplest of the algorithms to sort the following sequence of eight numbers: $\{12, 5, 95, 1, 7, 16, 26, 11\}$. The algorithm finds the smallest number in the list and swaps it with the element stored in the first position. It then scans the list again for the second smallest and swaps it with the element in the second position. It then goes on to find the third element and so on until the list is sorted in a non-decreasing fashion. If at any time the algorithm finds that the next smallest number is in the correct position, it leaves it there and moves to the next. This algorithm is applied to the example and it results in the sequence shown in Fig. 8.4.

In this particular example, the algorithm searched the list eight times and swapped numbers four times. Now, if the initial ordering of the list is changed the same algorithm will yield different results. It is easy to see that the same algorithm for the initial sequence {5, 7, 16, 1, 11, 95, 12, 26} will need seven swaps, and for the initial sequence {1, 5, 7, 11, 12, 16, 95, 26}, will only require one swap.

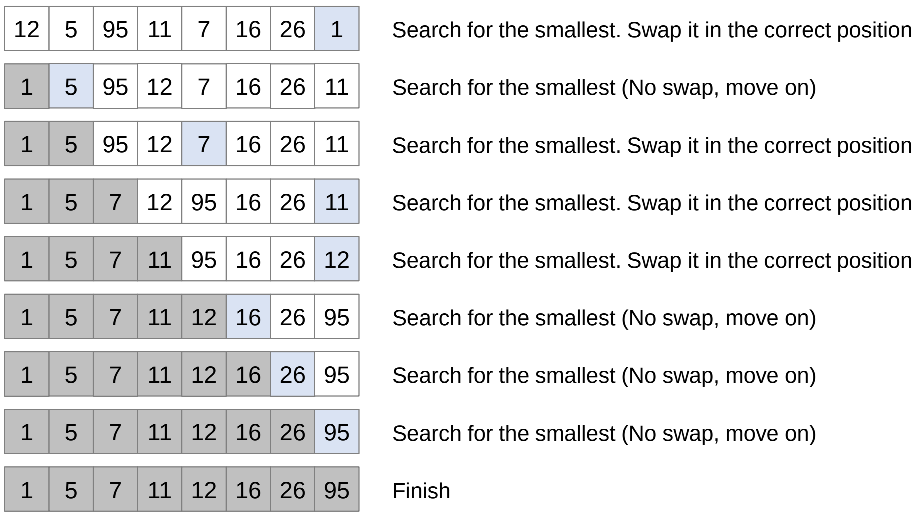

12 5 95 11 7 16 26 1
Search for the smallest. Swap it in the correct position
1 5 95 12 7 16 26 11
Search for the smallest (No swap, move on)
1 5 95 12 7 16 26 11
Search for the smallest. Swap it in the correct position
1 5 7 12 95 16 26 11
Search for the smallest. Swap it in the correct position
1 5 7 11 95 16 26 12
Search for the smallest. Swap it in the correct position
1 5 7 11 12 16 26 95
Search for the smallest (No swap, move on)
1 5 7 11 12 16 26 95
Search for the smallest (No swap, move on)
1 5 7 11 12 16 26 95
Search for the smallest (No swap, move on)
1 5 7 11 12 16 26 95
Finish

Figure 8.4: Example of basic sorting algorithm. Smallest number for each iteration shown in light blue. Numbers already sorted shown in grey.

In the case of the stencil selection, for the sorting problem this means two things: first, different sorting algorithms need to be tested to ensure efficient operation of the software; second, the parallel load balancing is difficult to achieve since we have no information about the cost of the sorting operation. It is worth noting that the performance of a sorting algorithm depends on the particular lists to be sorted. For this reason, several different algorithms are tested in an effort to improve the efficiency of the code. The algorithms tried are all described in detail in Ref. [115]:

- Selection Algorithm: The selection sort algorithm works by successively scanning the array to find the next smallest element and swapping it with the corresponding element in the correct position in the queue. The computational cost of this algorithm is fixed as it needs to scan the array the same number of times, regardless of how the array is originally ordered.

- Insertion Algorithm: While forming the array to be sorted, this algorithm finds the correct position for each element and then shifts all the candidates with bigger values of ψ one step to the right. The cost of this algorithm depends on how many elements need to be shifted to store each candidate.

- Bubble-Sort Algorithm: This algorithm works by comparing pairs of successive elements, swapping them if necessary and repeating the operation until no more swaps are required. In the worst case scenario, this algorithm is as expensive as the selection algorithm, but depending on the initial ordering of the array it can prove more efficient.

- Shell-Sort Algorithm: This algorithm in an extension of the insertion algorithm. It differs in the fact that it starts by comparing and exchanging elements that are far apart and progressively reducing the gap before finishing with neighbouring elements.

- Quick-Sort Algorithm: This algorithm follows a "divide and conquer" strategy. It creates successive partitions of the array by selecting a pivot. All elements smaller than the pivot are moved before it and all greater elements are moved after it. The operation is repeated until the array is sorted.

A summary of the algorithms and its expected computational performance is given in Table 8.1. The different sorting algorithms are evaluated in Chapter 9, in terms of the efficiency of the preprocessor.

Transient Simulations

The process of running transient simulations with movable geometry starts with a call to the preprocessor, which performs all the stencil selection operations and passes this information to the flow solver to solve the governing equations. After this first iteration, a closed loop starts with successive calls to the preprocessor and flow solver. At the beginning of each real time-step, the points are moved in space according to the prescribed or calculated motion, and the preprocessor is called to re-calculate the new stencils. The flow solver then uses these stencils to calculate the flow solution.

Table 8.1: Summary of tested sorting algorithms.

Name	Best case performance	Average case performance	Worst case performance
Selection-Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion-Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble-Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell-Sort	$O(n\,log(n))$	Depends on gap sequence	$O(n^2)$
Quick-Sort	$O(n\,log(n))$	$O(n\,log(n))$	$O(n^2)$

8.2 Parallel Implementation of the Preprocessor

Depending on the case to be studied, the preprocessing method described previously can become expensive in terms of computation time. It was described in Ref. [46] that the time taken by the preprocessor for simple three-dimensional cases ranges from around 10% to 20% of the time taken to converge the flow solution with the solver. This means that for complex cases, the time taken by the serial preprocessor can be counted in hours. This fact justifies the development of parallel implementations of the stencil selection procedures.

The parallel method described here is designed to work as far as possible with the same operations of the serial preprocessor described in Section 8.1. The implementation however is far from trivial. Most of the operations rely on the use of search trees and the main difficulties are finding ways of distributing the search trees among processors while maintaining good load balance and keeping parallel communications to a minimum.

The parallel preprocessor is designed to be used in modern clusters which normally include several interconnected symmetric multi-core (SMC) computing nodes. Each of these SMC nodes contains multiple processing cores with a shared memory space. For this reason, a two-level parallel implementation was devised. The first level is a distributed implementation that uses MPI commands to communicate across SMC nodes, while the second is a shared memory implementation using OpenMP directives to parallelise the work among the cores within each node.

8.2.1 Distributed Implementation

Usually, when using tree search algorithms, a master-slave parallel paradigm can prove beneficial, as the load balance is not assumed a-priori and slave CPUs perform work on small chunks of data at a time, requesting more work from the master as they finish their tasks [116]. In this work however, there was a specific requirement of not using a master-slave type of operation, as by avoiding master-slave algorithms, the number of cores doing work is maximised.

The distributed implementation of the preprocessor subdivides the domain by starting different MPI processes and assigning groups of points to each of the processes, instead of forming one global search tree for the domain and distributing it among CPUs. Each MPI process then forms its own separate search trees that are used for the different operations. Ideally, to increase the performance of the parallel stencil selection, each process should be allowed to work as independently as possible and the work load should be balanced between processes. An immediate drawback of performing tree searches for the stencil selection, is that it is not possible to have prior information about the computational cost of the operations for each individual point. The problem is aggravated as the candidate points need to be sorted according to the merit function

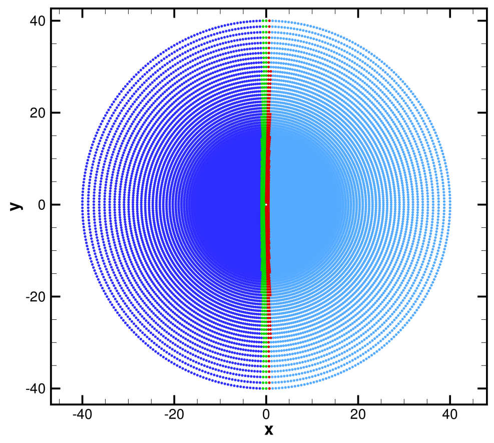
(Eq. 8.1). These considerations make it difficult to correctly load balance the stencil selection problem, as it will be demonstrated in the results section.

There are two modes of operation of the preprocessor. In the first mode, it can be used independent of the flow solver to take different overlapping input grids, calculate the stencils and write an output with the connected domain. In the second mode, the preprocessor is coupled with the flow solver for transient, moving-body simulations. The domain decomposition employed by the preprocessor depends on which of these modes is to be used.

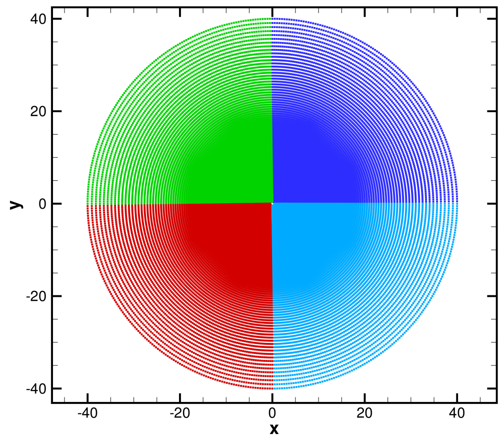
When the preprocessor is used independently, the different input point distributions are overlapped, and then the full computational domain needs to be decomposed among processes. At this stage, the domain is simply a collection of overlapped point distributions with no connection between them. For this reason, the METIS library cannot be used to perform the preprocessor domain decomposition, as it needs the full connectivity of the domain. Instead, two alternative decomposition methods are included with the preprocessor. The performance obtained from these two decomposition techniques can be case-dependent and it is left to the user to decide which is the most appropriate, for each case to be studied.

The first technique is based on slicing the domain according to the original Cartesian coordinates of the points, along a user defined axis. The width of the slices is automatically adjusted so that all the processes have roughly the same number of points. Figure 8.5(a) shows an example where the grid used in the test case of section 5.2 is divided among four processes, along the x-axis. The second type of decomposition is based on polar coordinates in two dimensions and cylindrical coordinates in three dimensions. This technique creates a polar reference system and transforms the coordinates of all the points in the domain into this system, using the transformations found in Appendix A.1. It then divides the polar circle into pieces, aiming to assign roughly the same number of points per piece, to finally assign one of these regions to a each of the processes. The centre of the polar system can either be chosen by the user, or placed at the centroid of the points in the domain. An example of this type of decomposition is found on Fig. 8.5(b), where the same case of section 5.2 is divided among four processes. In three dimensions a similar procedure is used, extruding the polar system along a user-selected axis to form a cylindrical system.

The simulation of transient, moving-body problems is started by the preprocessor selecting the stencils using one of the two decomposition techniques described above. After this initial step, the solver performs its own domain decomposition as described in Sec. 6.6 and from this point on, both the preprocessor and solver use this first decomposition calculated by the parallel flow solver. The decision to use the same domain decomposition in the preprocessor and solver was made to remove the cost of partitioning the domain in each call to the preprocessor and to avoid unnecessary input-output operations. It was demonstrated in Ref. [46] that the flow solver is more expensive



(a) Slices along the x-axis.



(b) Cake pieces using polar coordinates.

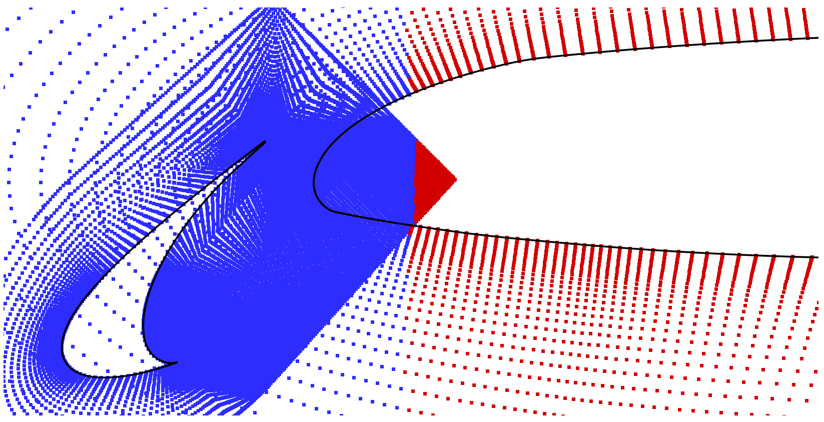
Figure 8.5: Domain decomposition for the preprocessor.

in terms of computing time than the preprocessor. Thus, to favour the flow solver decomposition is an acceptable compromise. For this reason, all the parallel preprocessing operations are designed to work with different types of domain decompositions, including the two described previously, and any type of decomposition given by the solver.

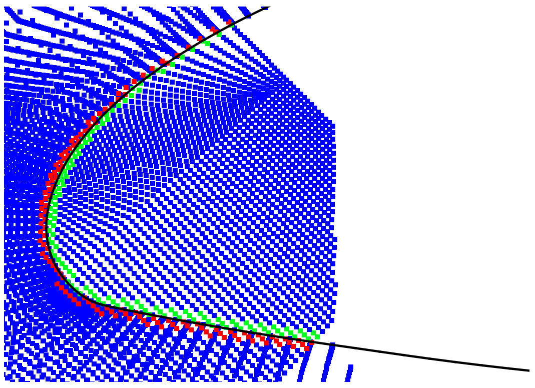
Blanking Points and Checking Final Stencils in Distributed Mode

When running in parallel, each process stores all of the global boundary elements. By doing this, the preprocessor avoids communicating information about the boundaries and most operations regarding boundaries can be performed by each process working independently from others. Hence, each process uses its local trees with the global boundaries to perform the searches, and the same serial operations defined in Section 8.1 to identify points directly behind solid boundaries work well in parallel. Difficulties occur however, when performing the flood operation described in Section 8.1 in parallel. For this operation to work correctly, the code successively blanks neighbours of points that were identified to be blanked by the ray intersection method. In parallel, some processes might not contain any points identified by the ray intersections as being behind solid walls, while still having points that are indeed inside solid bodies.

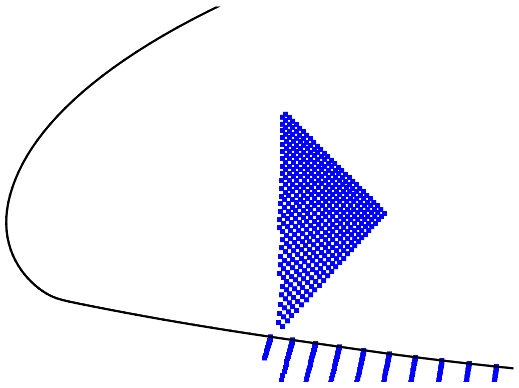
An example is shown in Fig. 8.6(a) where two point distributions belonging to different components of a multi-element aerofoil are overlapped and divided among two processes shown in blue and red. Figure 8.6(b) shows the result of the ray intersection method performed by process 1. Here, the green points are identified to be immediately behind solid elements, thus the points inside solid boundaries located in that process would be flooded as expected. The problem arises when process 2 do not find any



(a) Domain divided among two processes with process 1 shown in grey and process 2 shown in black.



(b) Detail of blanking operation for process 1 with dark grey points identified as being directly behind solid walls



(c) Detail of blanking operation for process 2 with no points identified

Figure 8.6: Example of the blanking of points interior to solid boundaries in parallel.

of its points located behind solid walls. The points forming the triangular shape in Fig. 8.6(c) which should be excluded from the calculation, are not. To solve this problem, processes need to exchange information after the first layer of points behind solid boundaries is found. This way the flood operation that stopped at the limit of an inter-process boundary, will continue to all processes.

The final stage in the parallel preprocessor is checking that the selected stencils are valid and that they respect the boundary elements. Similar to the first part of Stage 2, all processes work independently to check the validity of the selected stencils for their local points.

Selecting Stencils in Distributed Mode

Stage 3, which corresponds to selecting the final stencils, is modified and divided into the following operations:

(a) Form bounding boxes for each of the local points in the domain and create search trees considering these local points only.

(b) Identify the points located in regions of inter-process overlap to make sure only information from relevant points is communicated.

(c) Execute the parallel communication of coordinates, bounding boxes and merit function information for all the points identified before and add the received information to the local search trees.

(d) Perform the search for candidates following the same method described for the serial preprocessor and select the meshless stencils from the candidate list. It is at this stage that the biggest load imbalance is found. Even though all processes are assigned the same number of points, it is likely that the number of candidates per point, as well as the computational cost of the sorting operation, are different.

Details of the second and third operations are given in the following.

Identifying Relevant Points to be Communicated

When running in parallel, each process needs to find candidates for all its assigned points by searching through its local data, but it also needs to find potential candidates that are stored remotely in the other processes. To increase efficiency, before performing the parallel communications each process identifies a list of relevant points to be sent to other processes. This means that all processes will perform a preliminary search to make sure that only relevant data is communicated.

There are several ways this can be done. The simplest is for each process to form one large bounding box surrounding all of its points, communicating this box to all the other processes and letting them search which of their points lie inside the box. Points laying inside the box would be identified as the communication points. This simple method has the drawback of dramatically over-estimating the points to be communicated as can be visualised with the example in Fig. 8.7. Here, a domain formed of two overlapped aerofoils is divided among four processes using a decomposition obtained from the flow solver. In this example, the bounding box for the purple process contains all the points in the blue process and overlaps more than half of the points of the red one. This means that using this method, all of these points would have to be communicated to the purple process. The case of the green process is even worse, as its bounding box (not shown because it exceeds the limits of the image) contains all of the other processes. This would see the green process storing all of the points in the domain, thus resulting in zero parallel scalability in terms of memory. To improve the method, a scheme that uses adaptive sub-division of quadrants in two dimensions, and of octants in three dimensions was developed. The method is explained here for two-dimensional geometry, as the extension to three dimensions is straight-forward.

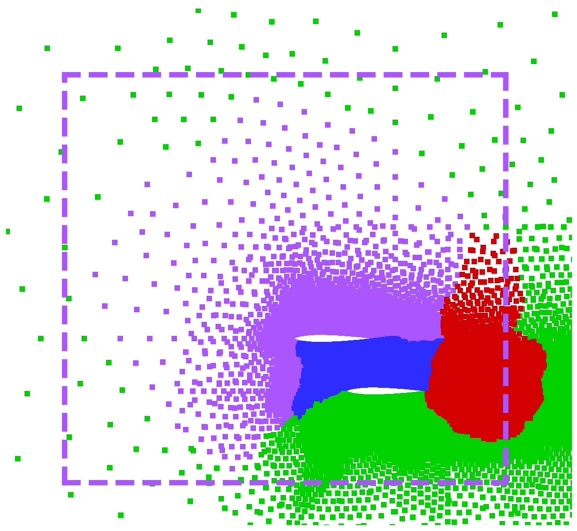


Figure 8.7: Example of a single bounding box surrounding all points assigned to a process.

To help with the description of the method, the same domain shown in Fig. 8.7 is used as an example. The sub-division method is described for process 1. The algorithm starts by forming a bounding box surrounding all the points initially assigned to the process. This box is iteratively sub-divided into four quadrants until a certain stopping criteria is achieved. The method uses a quad-tree data structure with four children branching from each parent to represent the quadrants created at each level. Every time a new sub-division is performed, process 1 searches to see if there are any of its initially assigned points contained within any of the newly created boxes. Boxes containing points are flagged as active, the rest are discarded. At each level, the method also assesses if all four new children of any given parent in the tree are active and if so, the parent is kept as active while the children are discarded. Conversely, if not all children of a given parent are active, the children are kept while the parent is excluded. The process stops if at any given sub-division iteration the number of newly created boxes is higher than the number of points initially assigned to the process.

The final quadrants, corresponding to level six in Fig. 8.8 are used to identify points stored in other processes that need to be sent to process 1. The points marked in Fig. 8.9 are the ones identified by other processes to be communicated to process 1 for the stencil selection.

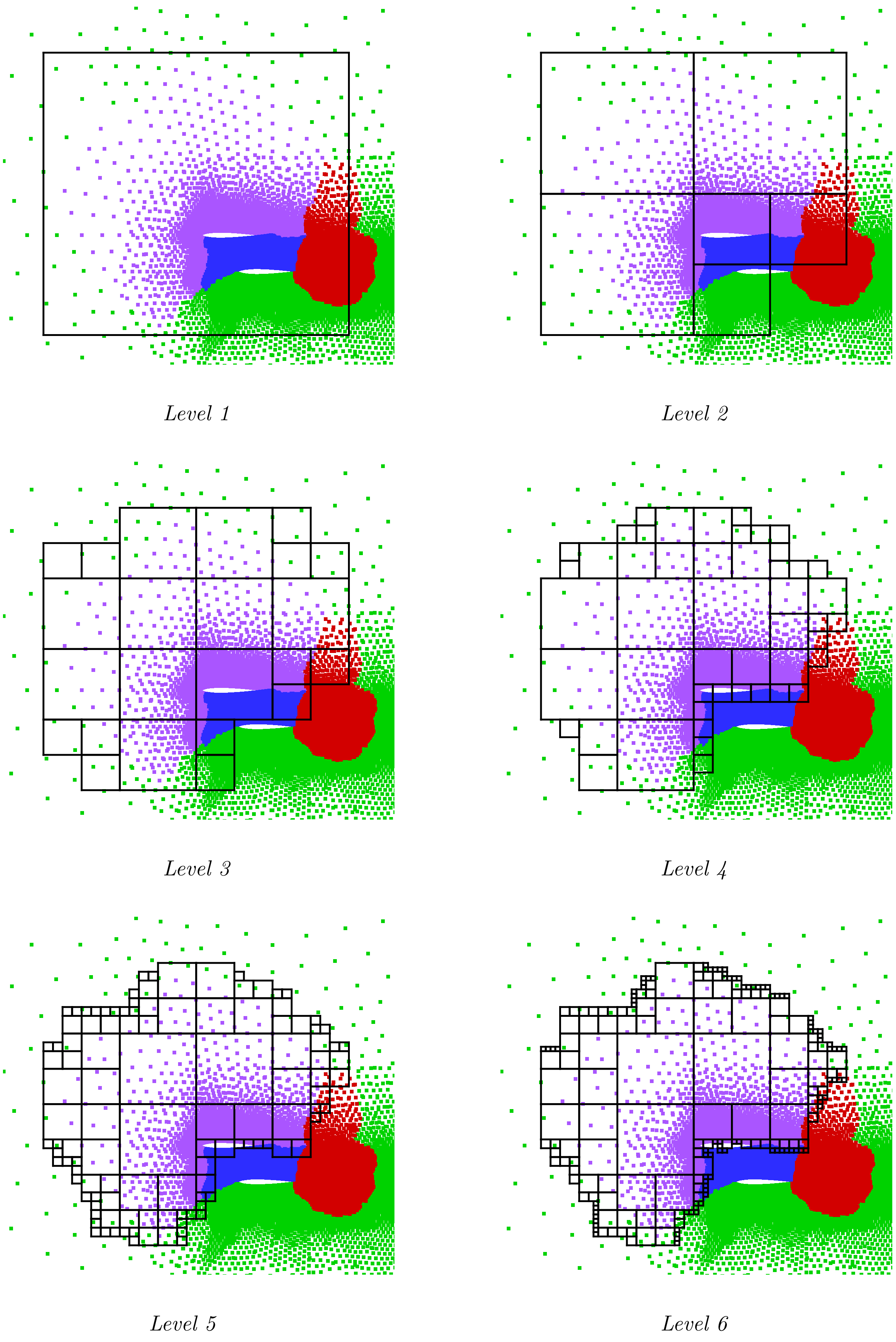


Figure 8.8: Example of adaptive sub-division by quadrants on purple process (The sub-division operation and the exclusion of parents/children are shown as if they were performed together at each step).

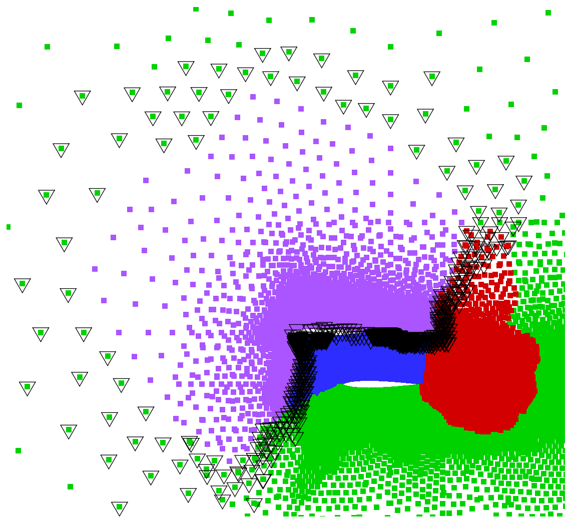


Figure 8.9: Points identified for communication.

Parallel Communication

In the third step, information is exchanged among processes. The data to be communicated includes the coordinates, bounding boxes and resolving vectors for all the points identified before. Different to the parallel communication for the flow solver, the parallel exchange here needs to be run synchronously. This means that the overhead caused by the latency of the network cannot be masked. After the data is exchanged, the received points are added to the local trees. Adding the extra points to the existing trees makes the searches more efficient, when compared to generating separate search trees containing received points only.

In the original method from Ref. [113], the search region for each point in the domain grows to the maximum size of any overlapping stencil. For example in Fig. 8.10, the blue box which surrounds the original stencil of the star point, intersects the green bounding box that surrounds another overlapping stencil. Hence, the search region for the star point grows to the size of the dashed box in grey. All points inside the grey box become candidates for the final stencil of the star point. Now imagine that all the points to the left of the blue dotted line, including the star point, are located in one process and all the points to the right of the dotted line are located in another. All the points inside the grey box located in the second process, including the green and red points, need to be sent to the first process to be consistent with the serial method. Even though the automatic sub-division method described in the previous section allows for quick identification of the points that need to be communicated, the number of these points can be quite large for some cases. This fact poses a problem that cannot be avoided without changing the method used to select the candidates for each stencil.

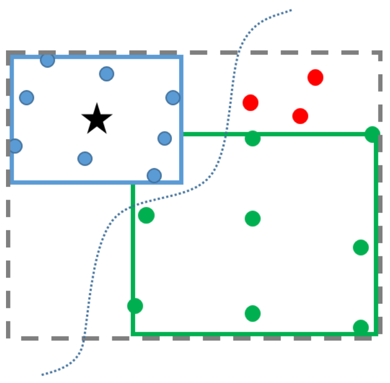


Figure 8.10: Example of the search region for a point. The size of the region grows with any overlapping stencil.

This is especially problematic in cases where big differences are found in the sizes of overlapping stencils, for example when a point distribution for a complex geometry which contains small stencils is overlapped with a background point distribution, which would normally contain bigger stencils to fill the background domain.

In an effort to alleviate the problem, in this work, the size of the search regions grows only to the location of the central point of any overlapping stencil and not to the maximum size of the stencil, as was the case in the original description of the method. This might result in different stencils being calculated with the original method and the one proposed here. In turn, this might have an effect on the meshless approximation of the governing equations that needs to investigated. Two other methods are currently being studied to alleviate this problem but their testing is still ongoing, thus their details are not included in this work. The methods that are being studied are:

- Using a hole-cutting method on any background point distribution to reduce the possibility of having big differences in the sizes of overlapping stencils.
- When running time-dependent simulations, using the data from previous time-steps to limit the size of the search regions.

8.2.2 OpenMP and Hybrid MPI/OpenMP Implementations

Besides load balancing, the main factor that affects the efficiency of the distributed implementation described above are parallel overheads. The main overheads of the distributed method are the preliminary search to identify points to be communicated and the actual parallel communication. One way to eliminate, or at least diminish these overheads, is to use shared memory architectures. A parallel application that only uses shared memory will completely eliminate the parallel overheads mentioned earlier, as all the processes have access to the same data. In this work, the shared memory parallelism is accomplished by starting different threads that run concurrently. The multi-threading algorithms are managed by OpenMP which is an application program interface that is used to explicitly direct multi-threaded, shared memory parallelism. In the preprocessor, OpenMP works by starting different *threads*, each of which is assigned to one processing core, so referring to running on N_p number of threads is the same as using N_p number of cores within an SMC node.

There are two main drawbacks with such an application. First, as the number of cores accessing the memory bank increases, the parallel performance will potentially decrease. More importantly, the cases to be studied would need to fit entirely into memory, making this type of implementation non-scalable in terms of memory. Knowing the advantages and disadvantages of each approach however, the preprocessor can benefit from a hybrid distributed/shared memory implementation. If used carefully, this type of operation can decrease the parallel overheads of the preprocessor while maintaining good scalability.

The hybrid method is developed from the distributed implementation described above. Separate threads are created on each of the SMC nodes and the loops of the different preprocessing operations are parallelised between the threads using OpenMP. The loops to be parallelised with OpenMP are all of the point-based loops. These are loops that go through all of the points in a given node. Most of the loops found in the preprocessor are point-based. As an example we can name the search for candidates, where each core goes through all of its assigned points, finding suitable candidates. All the parallel loops in the code are controlled by dynamic scheduling. This type of scheduling is controlled by the operating system at run-time and works by dynamically assigning groups of points to the separate threads. When a thread finishes its currently assigned group, the operating system assigns a new one and continues.

A simplified visualization of the hybrid code running with two MPI processes with two threads each, is seen in Fig. 8.11. Throughout the code, the multi-threading management follows a "fork/join" procedure, with several threads created at each "fork" juncture. In order to avoid any possible deadlocks when running in hybrid mode, with multiple threads started from each MPI process, only the master threads are responsible for the MPI communications.

In threaded applications, there is a latency overhead associated with actually starting and joining threads. For this reason, the threaded regions of the code were maintained as big as possible. In the shared memory method, threads were started by dividing the main MPI processes into forks at three locations: before the blanking of points, before the operations to select and sort the candidates and again before the checking of the stencils. The code is expected to run in modern computer hardware, designed with NUMA architectures. For this reason, as explained in Section 2.3, care needs to be taken to ensure the data to be processed by each thread is stored locally within the memory space assigned to each core. With this in mind, temporary storage for the required operations is allocated by each thread after they are started.

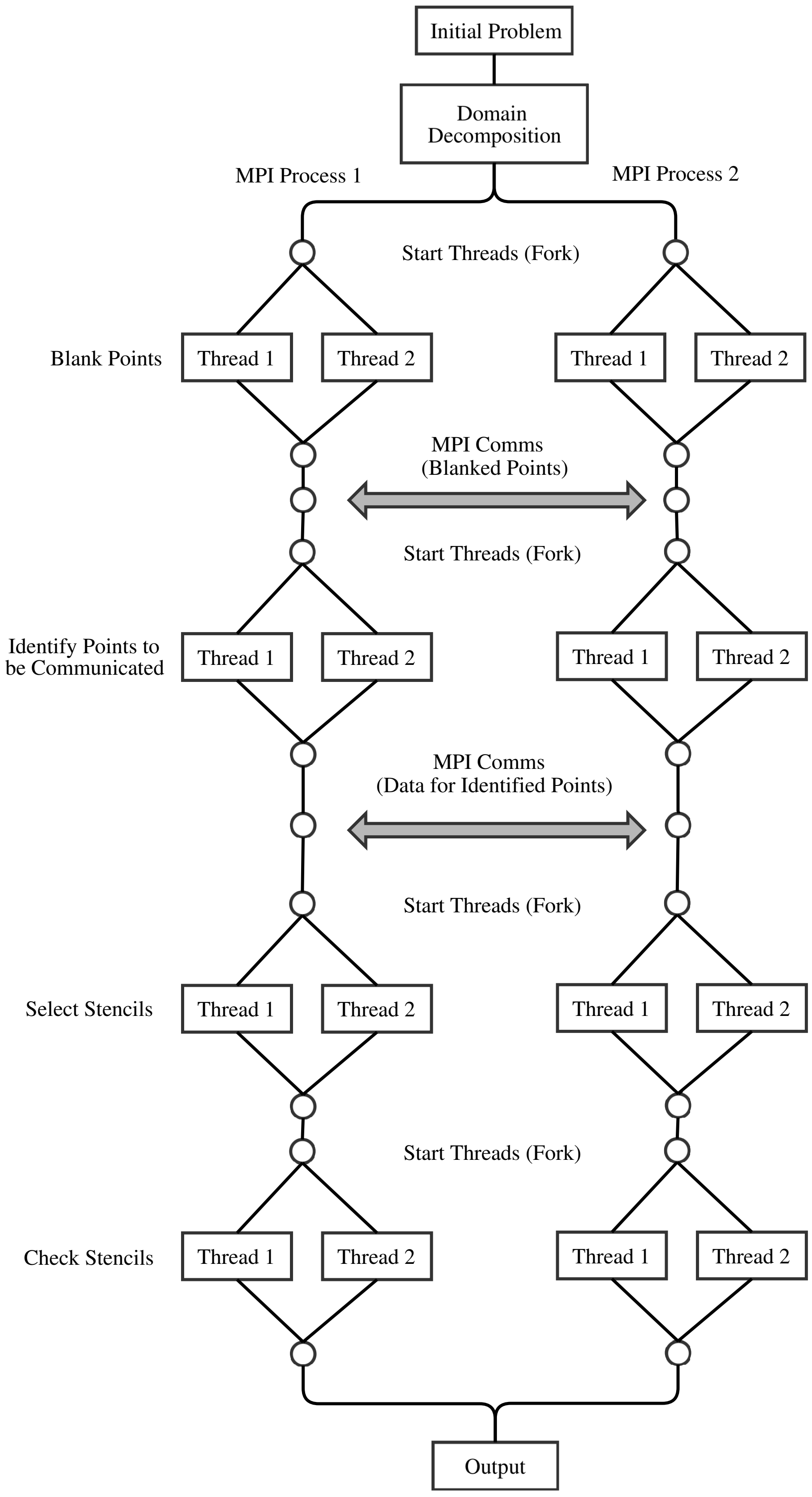


Figure 8.11: Representation of the hybrid MPI/OpenMP algorithm for the preprocessor.

Chapter 9

Parallel Stencil Selection Results

In this chapter, the performance of the parallel stencil selection is evaluated using four test cases, both in two and three dimensions. The purpose of these tests is to assess the parallel methods, not to perform detailed studies of the physics of a particular configuration. The selected test cases are academic, but they still serve the purpose of demonstrating the capabilities of the method.

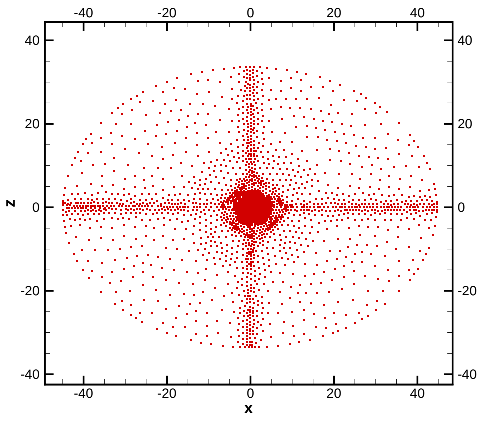
9.1 Presentation of Test Cases

The first test case consists of a two-dimensional aerofoil performing a cyclic pitching motion. The second consists of a two-dimensional multi-element aerofoil in a steady-state simulation. The third case is a simulation of the steady-state flow over the geometry of a generic fighter aircraft with several stores. The final test is a transient three-dimensional case, in which a store is released from a delta-wing geometry and shows the full capabilities of the parallel preprocessor and flow solver.

9.1.1 Test Case 1: NACA0012 Aerofoil in Transient Pitching Motion

This simple two-dimensional case was selected as a validation test for the parallel implementation of the preprocessor. It consists of a NACA0012 aerofoil undergoing a cyclic pitching motion. The case corresponds to the AGARD CT1 experiments documented in Ref. [117].

There are two input point distributions in the case: a background distribution and the aerofoil distribution. The background grid contains 21,585 points and is unstructured in nature. It forms an ellipse that extends 35 chords in the normal direction and 45 chords in the stream-wise direction and is shown in Fig. 9.1(a). The aerofoil point distribution is created from a structured grid and contains 29,798 points with first wall spacing of 1×10^{-3}. The aerofoil grid is shown in Fig. 9.1(b). The PMB grid used to compare the results is the same described for the NACA0012 case in Chapter 5.1.


(a) Background grid.

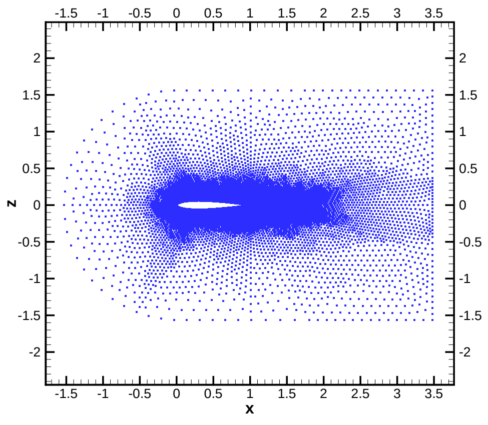
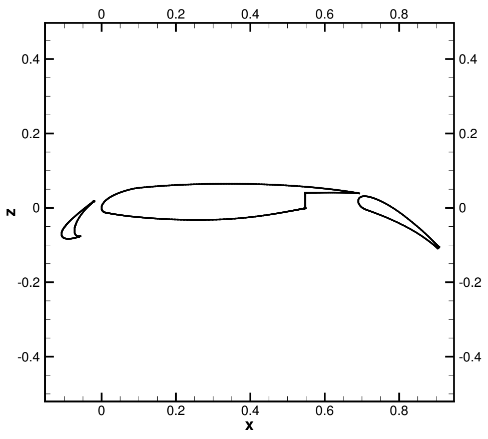

(b) Aerofoil grid.

Figure 9.1: Input point distributions for pitching NACA0012 test case.

9.1.2 Test Case 2: Two-Dimensional Multi-Element Aerofoil

In this two-dimensional case, four point distributions are overlapped to study the flow over a multi-element aerofoil. The test case, which consists of an aerofoil with a slat and a single-slotted flap, is known as 30P-30N and has been extensively studied using both numerical and experimental techniques. The first experimental results of the configuration were documented in [118]. The geometry of the multi-element aerofoil can be seen in Fig. 9.2(a). The four input point distributions are generated from grids for each of the elements of the aerofoil, plus the background grid. The background grid is the same used in the previous case. The input grids for all the elements of the aerofoil are structured in nature. The main element grid contains 34,766 points, the slat is composed of 24,026 and the grid for the flap contains 16,130. The assembly of the multi-element configuration is shown in Fig. 9.2(b).


(a) 30P-30N aerofoil configuration.

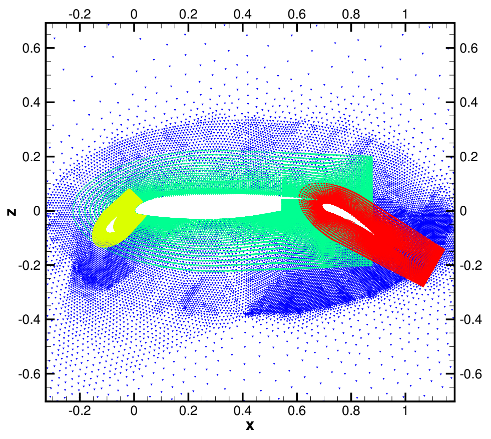

(b) 30P-30N overlapped grids

Figure 9.2: Multi-Element aerofoil.

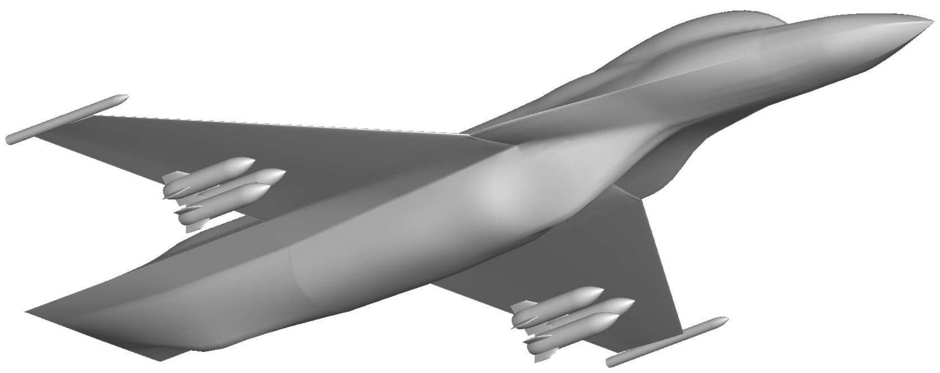


Figure 9.3: OSF test case.

9.1.3 Test Case 3: Open-Source Fighter

The third test case aims to simulate a bigger, more realistic case. It contains a generic fighter aircraft based on publicly available data of an F-16 fighter, combined with eight stores located under the wings. There are ten input grids in total, a background, the aircraft and one for each of the stores. The aircraft, which is called Open-Source-Fighter (OSF) is 14.47m in length, with a wingspan of 10.12m, formed of a NACA 64A204 profile. Details of the aircraft geometry are found in Ref. [119]. There are two types of store used in the study. The first one is a generic, fin-less store of 2.0m in length and a diameter of 0.125m. Two stores of this type are used, located one at the tip of each of the wings of the OSF. The second type of store, which was documented in [120], is ogive-shaped with a length of 0.17m and a diameter of 0.025m. Six of these stores are included in the model, three under each wing. The case geometry is visualised in Fig. 9.3.

The point distribution for the background was generated from an unstructured grid, while all the other distributions were generated from blocked-structured grids. The total number of points in the domain is 11.44 million points. The sizes of the grids for each of the bodies included in the test case are summarised in Table 9.1.

Table 9.1: Grid sizes for the OSF test case.

Body	Instances	Grid type	Number of points
Background	1	Unstructured	600k
Open-Source-Fighter	1	Structured	7000k
Store 1	2	Structured	330k
Store 2	6	Structured	530k
Total			11.44m

9.1.4 Test Case 4: Delta Wing with Store in Unsteady Mode

The fourth to be studied, assesses the capabilities of the parallel preprocessor and flow solver of performing time-accurate simulations with bodies in relative motion. The case consists of a delta wing with a store located beneath it. The wing is a full-span, clipped delta wing with a NACA64A010 aerofoil section, leading edge sweep angle of 45 degrees, with a root chord length of 7.62 meters and a semi-span of 6.6m. The test case was first described in references [120] and [121]. There are three bodies in the simulation, namely the far-field, the wing and the store. All of the point distributions used were generated from unstructured grids in ICEM-CFD. The point distribution for the far-field contains 725,000 points, while the distribution for the wing contains 700,000 points. The store used here is the same as the ogive-shaped store described for the second test case which contained 530,000 points. The first wall spacing for both wing and store is set to $1 \times 10^{-3}L$, where L is the diameter of the store. The initial configuration of the test case is shown in Fig. 9.4.

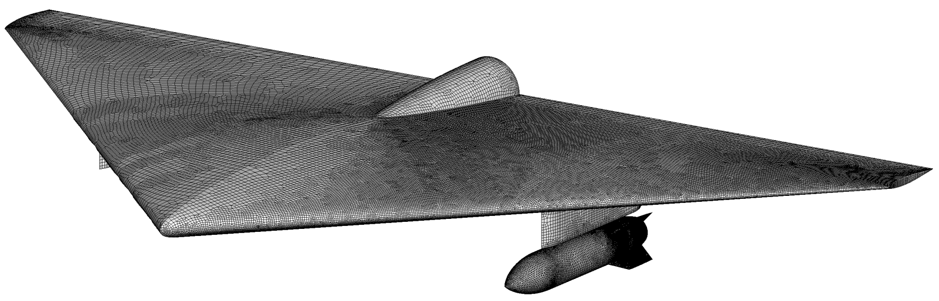


Figure 9.4: Store-drop test case geometry.

At the beginning of the motion, the store is forced away from its carriage position by two ejectors, which are modelled as constant vertical forces acting directly on the store. The trajectory and attitude of the store is calculated by a six degree-of-freedom (6-DOF) routine which is coupled with the flow solver. This way, at each time step of the unsteady simulation, the code calculates the aerodynamic loads by integrating the pressure along the store surface. Then, the 6-DOF module calculates the new location of the store and finally the preprocessor calculates the new stencils. The 6-DOF module is described in detail in Appendix A.2 and the characteristics of the store used in this study are summarized in Table 9.2.

Table 9.2: Full-scale store characteristics.

Variable	Value
Mass	$907kg$
Store Centre of Gravity	$1.416m$ (Aft of store nose)
Store Diameter	$0.508m$
Roll Inertia	$27.12kg \cdot m^2$
Pitch Inertia	$488.1kg \cdot m^2$
Yaw Inertia	$488.1kg \cdot m^2$
Forward Ejector Location	$1.24m$ (Aft of store nose)
Forward Ejector Force	$10675.7N$
Aft Ejector Location	$1.75m$ (Aft of store nose)
Aft Ejector Force	$42702.9N$
Ejector Stroke Length	$0.1m$

9.2 Profiling the Code and Sorting Algorithms

Before proceeding with the evaluation of the parallel preprocessor, the performance of the different operations was studied by profiling the code for the cases one and three described above. The CPU time per operation was measured and the average of five runs is taken. Table 9.3 shows the relative computational cost of the preprocessing operations when running two MPI processes. From the results it is clear that the most expensive operation is the sorting of the candidates according to the merit function.

The performance of a sorting algorithm depends on the particular data to be sorted. For this reason, the different sorting algorithms described in Section 8.1 were tested to find the most efficient one for the type of arrays found on the preprocessor. Table 9.4 shows the speed-up compared to the most expensive algorithm (selection sort) for test cases 2 and 4 (initial position only). The bubble sort and the insertion algorithms proved to be of similar efficiency, with both of them being about 3 times faster than the selection algorithm. There is an increase in speed of more than 7 times using the shell sort algorithm in respect to the least efficient one, while the Quick-Sort method

Table 9.3: Profiling of the preprocessing code.

Parallel Preprocessor Operation	Percentage of total CPU time for case 1	Percentage of total CPU time for case 3
Blank points	0.7 %	2.5 %
Local candidate search	4.9 %	2.5 %
Preliminary search	3.9 %	3.2 %
Parallel communication	0.2 %	0.9 %
Remote candidate search	2.0 %	5.3 %
Sort candidates and select stencils	85.3 %	79.4 %
Check final stencils	3.0 %	6.2 %

Table 9.4: Speed-up of different sorting algorithms.

Test case	Selection-Sort	Insertion-Sort	Bubble-Sort	Shell-Sort	Quick-Sort
Case 1	1.0	2.8	2.8	7.1	9.6
Case 2	1.0	2.8	2.9	7.6	9.0

is more than 9 times faster than the selection algorithm. Based on these results, the Quick-Sort algorithm is now being used in the code and all the following results were obtained using this sorting method.

9.3 Results for the NACA0012 Aerofoil in Transient Motion Test Case

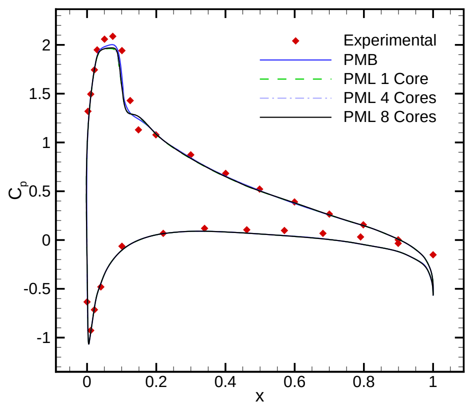
The simulation of the NACA0012 aerofoil in pitching motion serves as an introductory case to demonstrate the capabilities of the method to tackle flow calculations over moving geometries. It is also used to validate the results obtained from the parallel implementation. Because the purpose of the method is to assess the parallel implementation and not to perform a detailed study of the flow physics, the flow is assumed to be inviscid. Results from PML are compared to simulations in PMB and to the original experimental results documented in Ref. [117]. The Mach number is set at 0.6 and the predefined aerofoil motion is a sinusoidal pitch angle change around the quarter chord of the aerofoil. The pitch angle follows the relation

$$\alpha(t) = \alpha_0 + \alpha_a \sin(2kt) \tag{9.1}$$

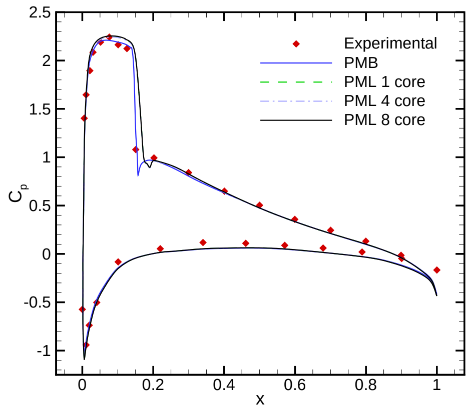
with $\alpha_0 = 2.89°$, $\alpha_a = 2.41°$ and $k = 0.0808$ representing the mean incidence, the pitch amplitude and the reduced frequency, respectively.

During the transient simulation in PML, the background grid is kept static, while the entire aerofoil grid is moved according to the function in Eq. (9.1). Three motion cycles, each consisting of 64 real-time steps were simulated to solve the flow. At each of the real time-steps the aerofoil was moved, the preprocessor was called to calculate the new stencils, and then solver was run until a reduction in the residuals of three orders of magnitude was achieved. Since PMB does not allow for the simulation of movable grids, the pitching movement of the aerofoil was simulated by changing the far-field boundary conditions. PMB was run in serial mode, while PML was run using the distributed implementation using one, four and eight cores.

The pressure coefficient over the aerofoil at two different instants of the simulation are shown in Fig. 9.5. In the figure, the differences in the results obtained with different number of cores are negligible, thus validating the parallel implementation. The results



(a) $\alpha = 5.0°$(upstroke).



(b) $\alpha = 4.82°$(downstroke)

Figure 9.5: Surface pressure coefficients for the NACA0012 in pitching motion.

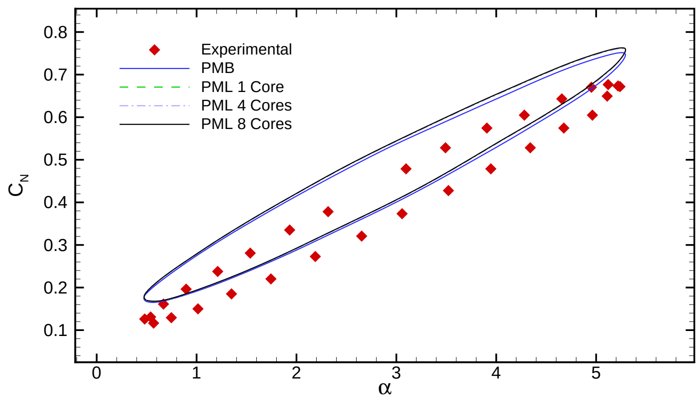
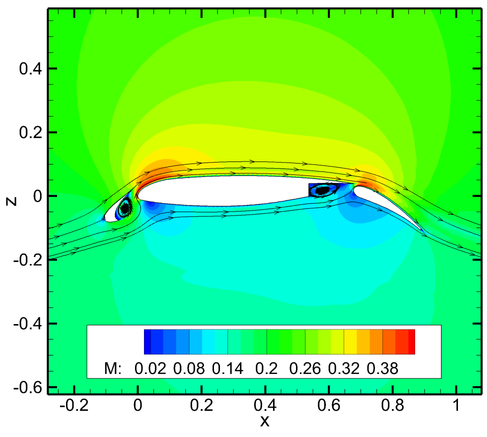


Figure 9.6: Normal force coefficient for NACA0012 in pitching motion.

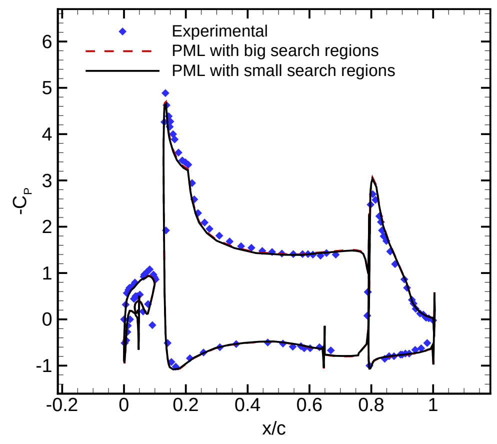
from PMB and PML are in good general agreement, even though small differences are found between the solvers, especially in the location of the shock over the aerofoil on the downstroke. Finally, Fig. 9.6 shows the normal force coefficient on the aerofoil. Even though both solvers show relatively large differences with the data obtained from the experiments, the results from PMB and PML running on different number of cores are again in good agreement. It is important to note that the discrepancies with the experimental data are not a cause for concern, as they are explained by the fact that the simulations were performed assuming inviscid flow.

9.4 Results for the Multi-Element Aerofoil Test Case

For the multi-element aerofoil test case, the preprocessor and flow solver were run in both serial and parallel modes with different numbers of MPI processes. Fully turbulent flow is assumed at a Reynolds number of 9 million, a free stream Mach number of 0.2,



(a) Mach number contours.



(b) Surface pressure distribution.

Figure 9.7: Flow solution for multi-element aerofoil test case.

and zero degrees angle of attack. To obtain the turbulent flow solution, the flow solver was started from a converged Euler calculation on a coarser grid by using interpolation. To achieve this, the same framework used to select stencils for points is used to find possible interpolation candidates from the coarser grid. This way, the code forms bounding boxes for all the points in the coarse and fine grids and searches for box intersections. All the intersecting points in the coarse grid are interpolation candidates for the points in the finer grid. The variables on all points in the finer grid are started by making an average of the previous solution of all its candidates from the coarser grid, weighted with the distance to the point in the finer grid.

The converged flow solution shown in Fig. 9.7(a) was obtained after the solver converged five orders of magnitude for the mean flow variables. In the figure, the big separation regions behind the forward slat and the flap cavity are clearly appreciated. This case was used to test the modification to the method described in Section 8.2.1, which makes the search regions of all points grow to the location of the central point of an overlapping stencil, instead of growing the regions to the maximum dimension of the overlapping stencil. Figure 9.7(b) shows virtually identical results from stencils calculated using the small or big search regions, and both solutions show good agreement for the surface pressure coefficient when compared with the experimental results from [122].

The performance of the parallel preprocessor is measured by two metrics, memory usage and speed-up. The case was run on a system of desktop machines and the preprocessor was run separately in distributed (MPI only), shared (OpenMP only) and hybrid (MPI/OpenMP) modes to test the performance of each implementation. The domain decomposition was performed using the polar coordinates method aiming to assign the same number of points to each process.

The parallel speed-up using the distributed implementation with one MPI process per core is shown in Fig. 9.8. The performance of the parallel preprocessor is good when using two MPI processes, but decreases as the number of processes is increased. This can be explained by Figs. 9.9 and 9.10, where the run times for the preprocessing operations for two and twelve processes are shown. These figures give an indication of the load balance for the preprocessor. When using two MPI processes for instance, the work load is well balanced and both processes finish their operations in about the same time. When running in twelve processes on the other hand, the load balance is not as good, resulting in some processes finishing much faster than the rest. In the figures, the operations for blanking points and the final checking of the stencils are omitted as their contribution to the total computation time is very small for this case. The imbalance in the work load comes from the fact that the domain decomposition cannot predict the cost of the operations per point as it was explained in Section 8.2. Figure 9.11 shows the memory usage for each process when running the preprocessor on one to twelve cores. As expected, for the higher number of processes the total memory consumption increases. This is due to the extra data stored in regions of inter-process overlap. Figure 9.12 shows the number of points initially assigned to each sub-domain as well as the number of extra points received from other sub-domains for the runs with two and twelve processes. To be consistent with the serial method, when running twelve processors almost all of the processes have to store a bigger number of extra points than the amount that was initially assigned to them. For this test case, the large number of points to be exchanged is not a particular problem as the memory overhead is manageable. For bigger cases however, this might constitute a problem as it will be seen in the next test case. In Fig. 9.12 it is also clear that the smaller search regions result in a lower number of points being communicated, which in turn results in lower memory consumption. These results show that the change to smaller search regions provides a positive effect, and they are used in the rest of this thesis.

The tests using the shared memory implementation were run on a workstation with six cores using one thread per core. Figure 9.13(a) shows good speed-up, with an efficiency of approximately 85% when using the maximum number of cores in the machine. The efficiency of the preprocessor decreases as the number of cores is increased, as memory access becomes the bottleneck. When using a shared memory approach, the memory usage of the preprocessor changes only by a small amount as more processes are used. For this reason, the parallel speed-up is the only metric used to measure the performance.

The hybrid implementation was tested with the preprocessor running on a distributed system using two, three and four computing nodes, with one MPI process started on each of the nodes. Each of the MPI processes were parallelised on a different number of threads using OpenMP. Separate tests were performed using two, three and four threads per node, running one thread per core. Table 9.5 summarises the parallel

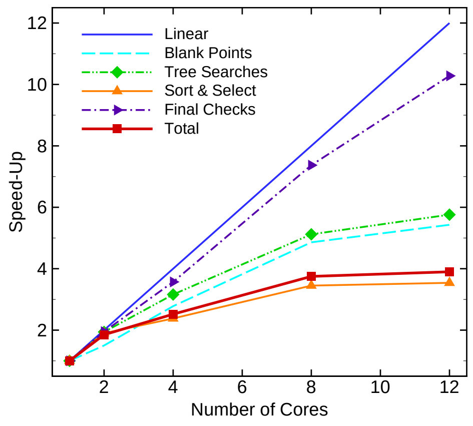


Figure 9.8: Speed-up of preprocessor for multi-element aerofoil case using distributed implementation with polar decomposition, running one MPI process per core

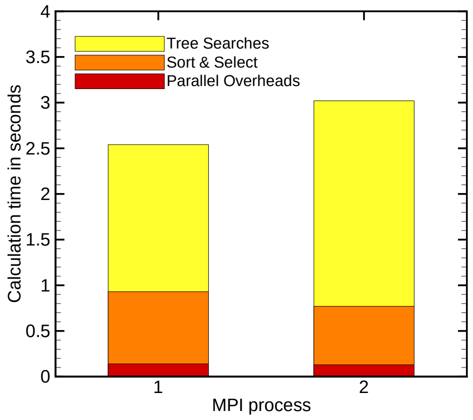


Figure 9.9: Load balance for two processes for multi-element aerofoil case using distributed implementation with polar decomposition

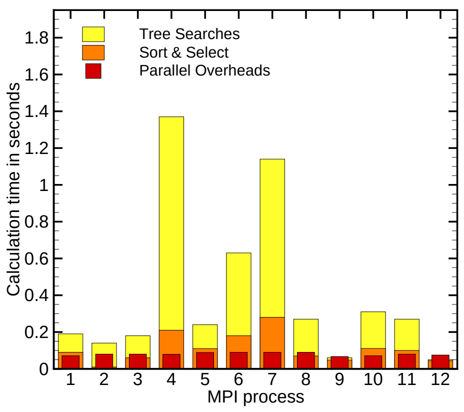


Figure 9.10: Load balance for twelve MPI processes for multi-element aerofoil case using distributed implementation with polar decomposition

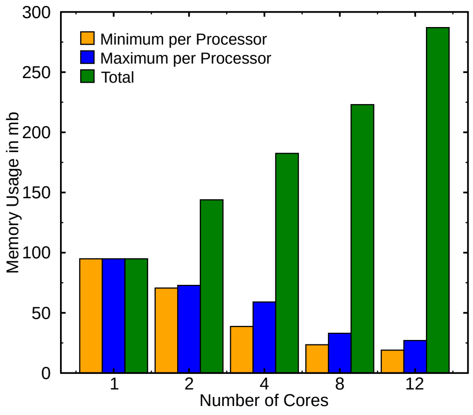
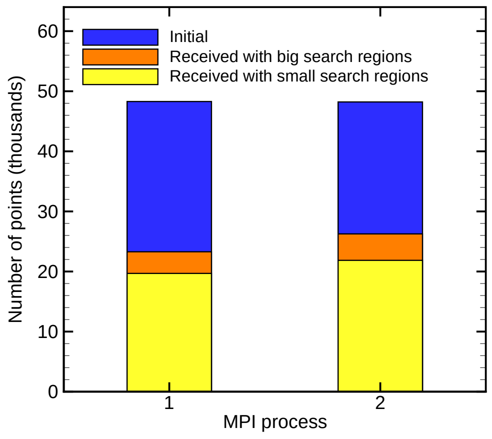


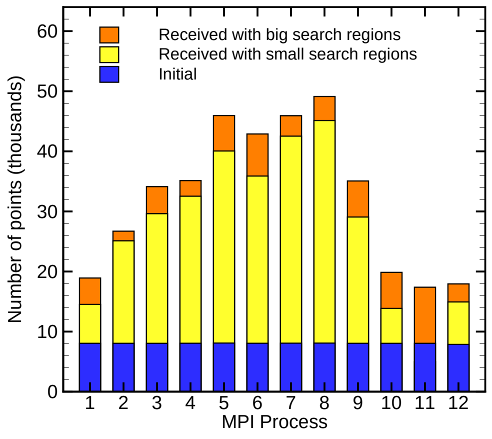
Figure 9.11: Preprocessor memory usage for multi-element aerofoil case using distributed implementation with polar decomposition

configurations that were tested. The parallel speed-up on hybrid mode is shown in Fig. 9.13(b), along with the parallel speed-up for the MPI-only implementation, for comparison. The figure shows an important increase in the performance of the parallel preprocessor when running in hybrid mode. The reasons for this increase are that running on a lower number of MPI processes means an increase in the load balance as well as a reduction of the relative cost of the parallel overheads. Combining this with the performance increase of the OpenMP-only implementation results in a much better overall efficiency, with speed-up gains of 88% when using three nodes and four threads, compared to twelve individual MPI processes (shown in red in Fig. 9.8).

As it was previously established, it is not possible to have an a-priori estimation of the cost per point of the preprocessing operations. It is possible however, to calculate an



(a) 2 processes



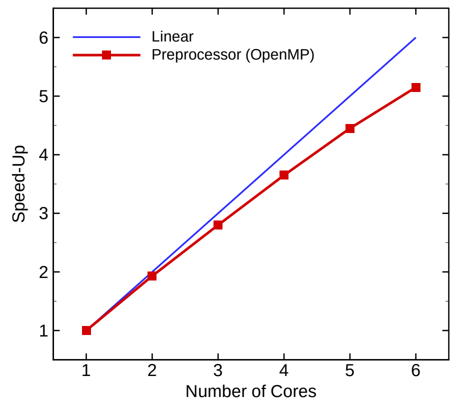
(b) 12 processes

Figure 9.12: Number of initial and communicated points per MPI process for multi-element aerofoil case using distributed implementation with polar decomposition. Comparison of number of communication points between small and big search regions.

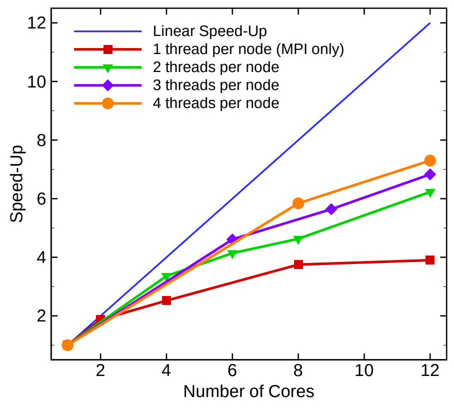
Table 9.5: Processor combinations in hybrid mode for multi-element aerofoil case.

MPI Processes	Threads	Total number of CPUs
2	2	4
2	3	6
2	4	8
3	2	6
3	3	9
3	4	12
4	2	8
4	3	12

estimate of this cost after the preprocessor finished calculating the stencils. This would make it feasible to rebalance the partitions on subsequent calls to the preprocessor, as it would be the case for transient simulations with moving bodies. To test this idea, the preprocessor was run in serial mode and the CPU time needed to perform the tree searches and the sorting and selecting of candidates was used as an estimation of the cost per point of the operations. The parallel preprocessor was then started using a domain decomposition based on the polar coordinates method, aiming to balance the load, based on the cost estimation obtained previously. The parallel preprocessor was run in hybrid mode on an increasing number of MPI processes (nodes) from two to four and always using three threads per node. Comparing Fig. 9.14(a) to Fig. 9.10 it is clear that the load is much better balanced when using four MPI processes and three threads than when using twelve MPI processes. This results in the speed-up shown in Fig. 9.14(b). The figure gives a clear increase in performance after readjusting the

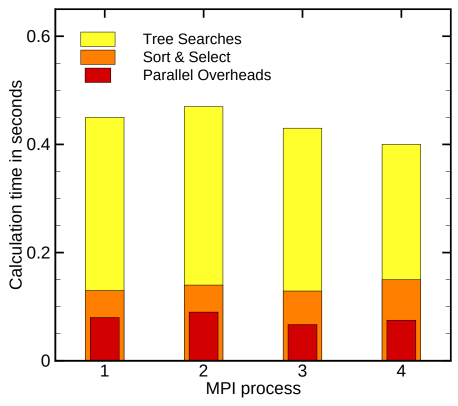


(a) Shared memory implementation.

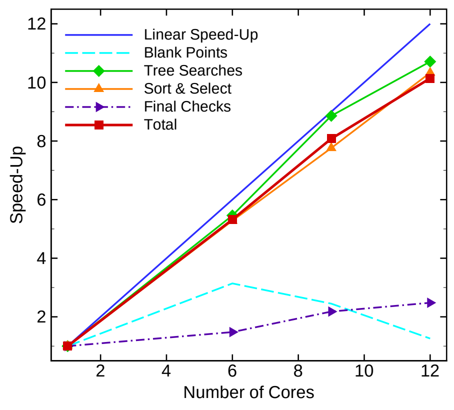


(b) Hybrid implementation.

Figure 9.13: Performance of parallel preprocessor for multi-element aerofoil using shared and hybrid implementations.



(a) Load balance with 12 cores (4 MPI processes and 3 threads each).



(b) Speed-up with increasing number of MPI processes and 3 threads each

Figure 9.14: Performance of parallel preprocessor for multi-element aerofoil using hybrid implementation after rebalancing the partitions.

partitions, resulting for example in close to 50% decrease in the total calculation time when running four MPI processes and three threads, compared to the same combination before rebalancing the partitions (shown in blue with diamonds in Fig. 9.13(b)). It is worth noting that after the cost assessment, the efficiency of the blanking and final checks operations decreased. This is because the estimation of the cost was taken only from the tree searches and the sorting and selecting of candidates. The cost of the blanking and final checking of points was not included in the estimation, as their contribution to the total cost is small.

9.5 Results for the Open-Source Fighter Test Case

For this test case the preprocessor and solver were run on a different number of cores ranging from 12 to 96. The Polaris HPC cluster was again used for the calculations. The flow solver simulated inviscid flow at an angle-of-attack of 3 degrees and a Mach number of 0.5. The flow solution obtained from 96 cores is shown in Fig. 9.15. Here, the surface pressure is plotted along with streamlines coloured by the stream-wise velocity.

The preprocessor was run in distributed mode using both the slice-based and polar-based decompositions, on a different number of MPI processes ranging from 12 to 96, using one process per core. Both decompositions aimed to balance the number of points per process. The Polaris N8 cluster was again used for the calculations. As with the previous case, memory usage and speed-up are the metrics used to measure the performance of the parallel preprocessor.

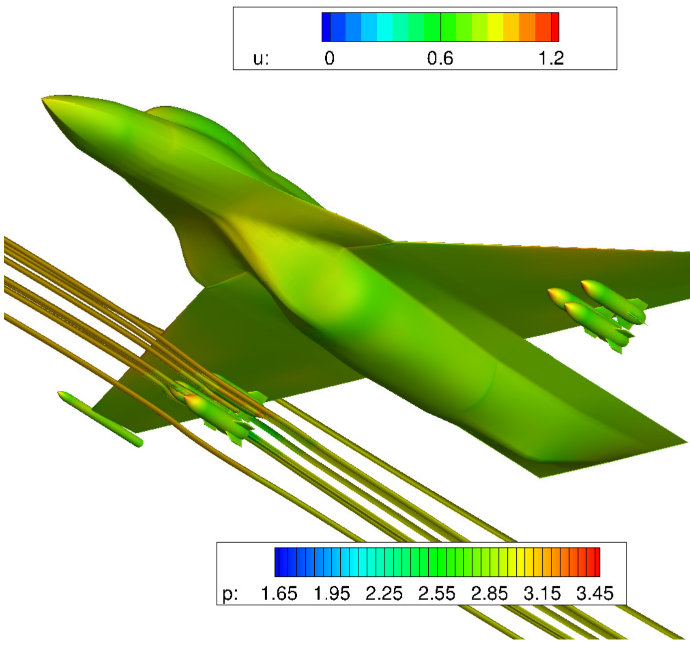
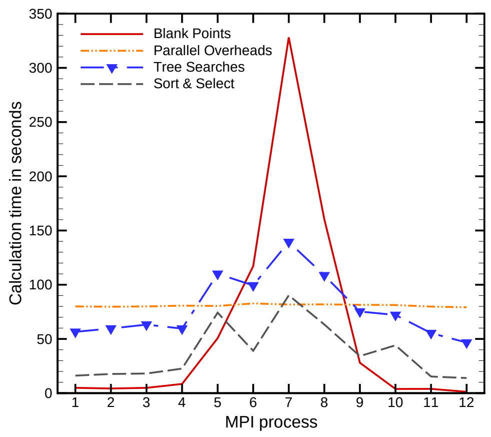
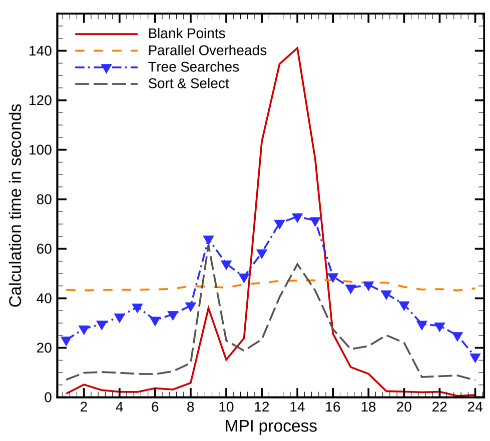


Figure 9.15: OSF test case flow solution showing surface pressure and streamlines coloured by stream-wise velocity.

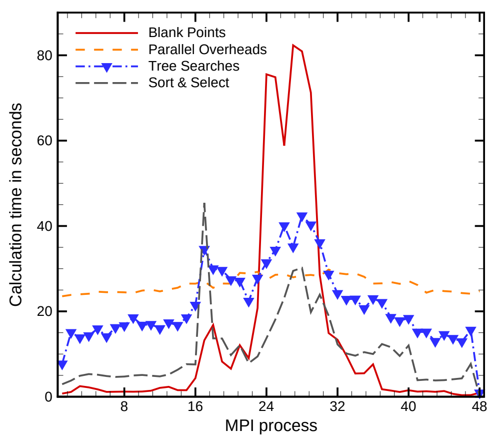
When using the slice-based decomposition, and contrary to previously tested cases, the most expensive operation of the preprocessor is the blanking of points. This can be appreciated in figs. 9.16(a) to 9.16(d), which show the calculation times per operation, per core for this type of decomposition. From the figures it is clear that the slice-based decomposition does not result in a good load balance. The processes which were assigned to the central points in the domain do most of the work, while the rest are left idle, waiting for the others to finish. This behaviour is due to the fact that the points that intersect solid boundaries are found on the centre of the domain.



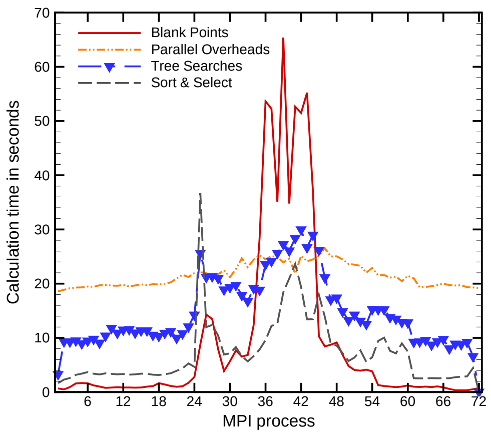
(a) 12 processes



(b) 24 processes



(c) 48 processes



(d) 72 processes

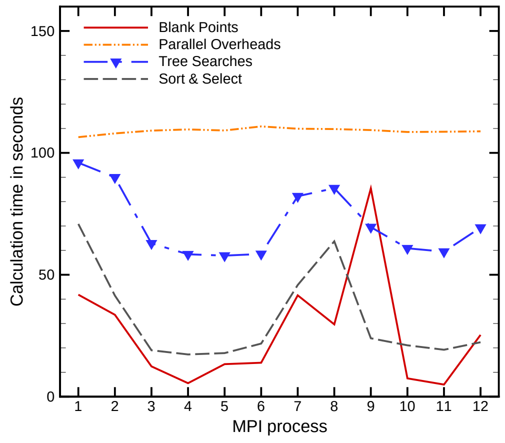
Figure 9.16: Calculation times per operation, per process for OSF test case with slice-based domain decomposition

When using the polar-based decomposition, the load balance is in general much better compared to using the slice-based partitioning. The times per operation, per process for the polar-based decomposition are shown in figures 9.17(a) to 9.17(d). This improvement in load balance of the polar decomposition compared to the sliced-based partitioning translates into a much faster preprocessing time as it can be seen in Fig. 9.18(c). For this case, the polar decomposition proves to be in average 40% faster than the sliced partitioning.

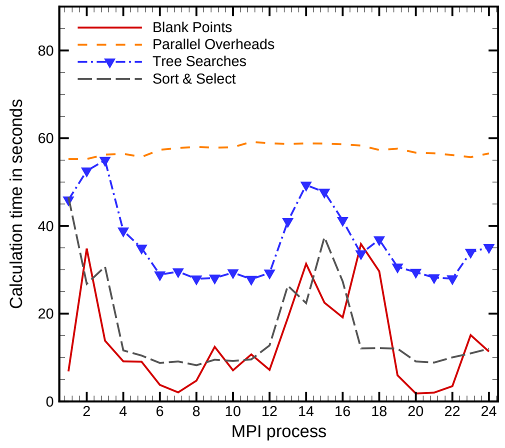
To give an indication of the load balance of the preprocessor, the following formula is used:

$$\varphi = 100\left(1 - \frac{(\check{M} - \hat{m})}{\check{M}}\right) \tag{9.2}$$

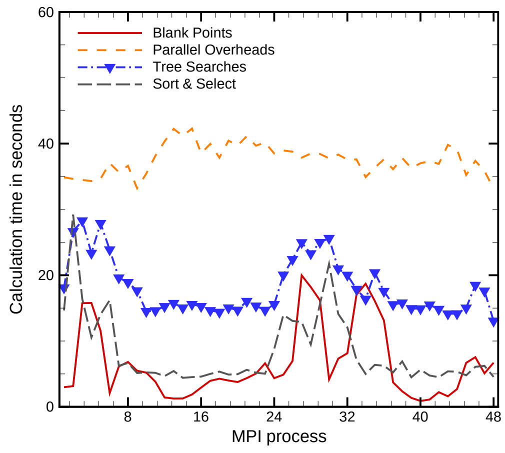
where φ is the load balance percentage, $\check{M}$ is the maximum time taken by a process, and



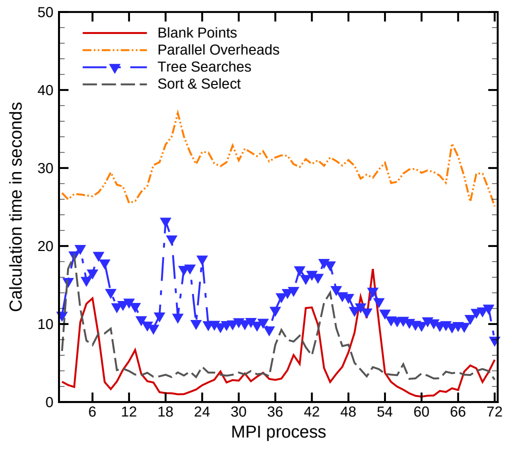
(a) 12 processes



(b) 24 processes



(c) 48 processes

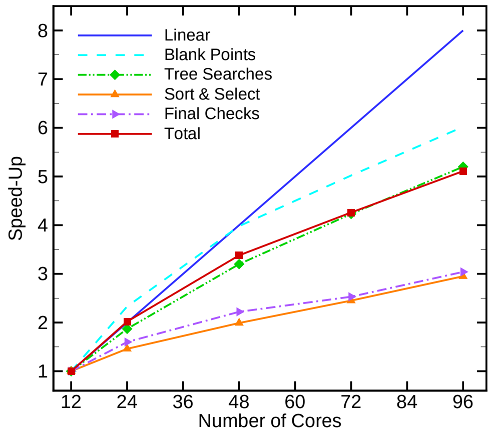


(d) 72 processes

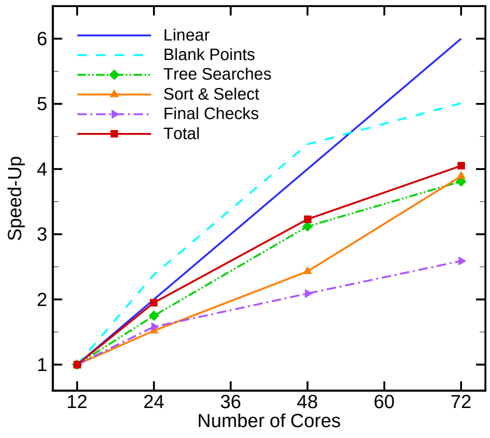
Figure 9.17: Calculation times per operation, per process for OSF test case using polar-based domain decomposition

$\hat{m}$ is the minimum. This equation was used to calculate the load balance percentage for each of the separate operations, and then a weighted average of these percentages was used to obtain the values shown in Table 9.6. These values give an indication of the overall load balance of the preprocessor.

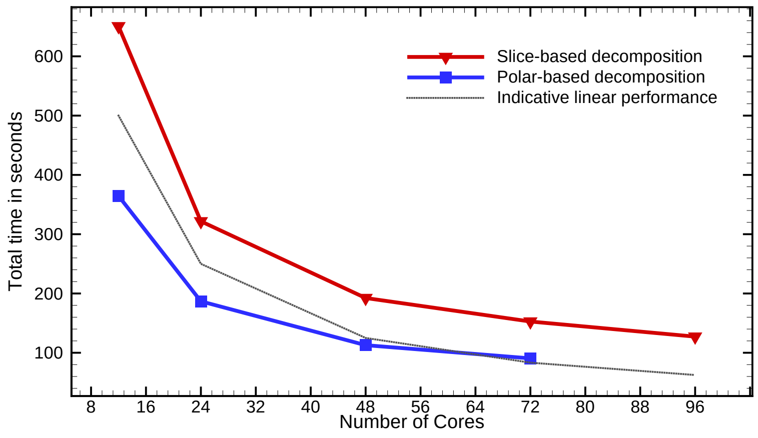
The parallel speed-up achieved by the preprocessor using both decomposition techniques can be seen in figures 9.18(a) and 9.18(b). Even though neither of the two types of decomposition can guarantee an even load balance, an important reduction in the preprocessing times is obtained with both of them. The reduction in the rate of increase of the speed-up when using more processes is due to two reasons. The first is that as shown in Table 9.6, the load balance decreases as more processes are added. The second, is that the parallel overheads become more expensive in relation to the other operations when using more processes, a fact which is clear when looking at figure 9.17.



(a) Slice-based domain decomposition



(b) Polar-based domain decomposition



(c) Calculation times

Figure 9.18: Parallel preprocessor speed-up for OSF case using distributed implementation with two types of domain decomposition

In terms of memory usage, the parallel method yields mixed results. On one hand, as expected, the memory usage per process decreases with more processes. On the other hand, the rate of descent is not as high as expected and the total memory usage increases dramatically when using more processes. This behaviour is due to the large number of points that need to be exchanged for the method to be consistent with the serial implementation, as explained in Chapter 8. The large number of points to be communicated can be visualised in Figs. 9.19 and 9.20. From here, it is clear that the polar-based partitioning results in considerably more points being exchanged than with the slice-based decomposition. As with the previous test case, the use of smaller search regions results in a lower number of points being communicated.

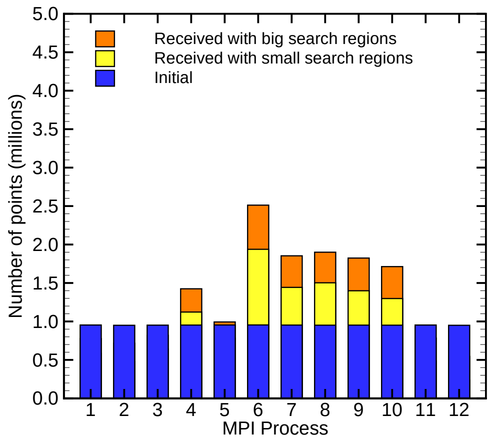
The memory usage with the small search regions is shown in figures 9.21(a) and 9.21(b) for both types of domain decomposition. The number of cores per node

Table 9.6: Preprocessor load balance for OSF test case.

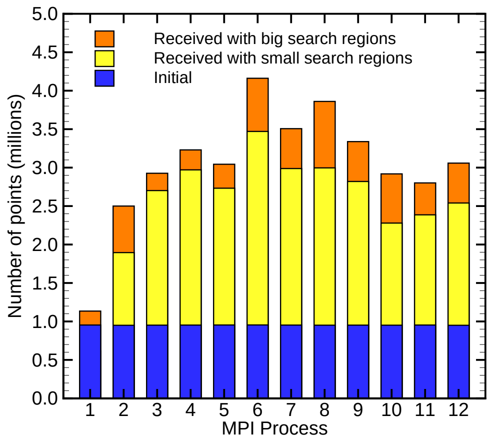
Slice-based decomposition		Polar-based decomposition	
Number of processes	Overall load balance	Number of processes	Overall load balance
12	19%	12	45%
24	16%	24	42%
48	8%	48	36%
72	6%	72	30%
96	7%	-	-

used for the calculations was fixed to twelve (out of 16 available per node) to decrease the total memory usage per node. Using the slice-based domain decomposition the total memory was manageable on this configuration running on 96 processes, but using the polar-based decomposition the total memory exceeded the maximum per node and the calculations failed. The solution to the problem would be to decrease the number of cores per node used, but due to the fact that the number of cores per node used can have a effect the efficiency of the calculations, no attempt was made to decrease this number in an effort to keep the parallel comparisons fair. These are the reasons why the calculation time data shown above increases only to 72 processes when using the polar-based domain decomposition.

In an effort to alleviate some of the previously described difficulties, this test case was also calculated using the hybrid MPI/OpenMP method. The case was run using the polar-based decomposition on 12, 24 and 48 MPI processes with one, two and four OpenMP threads each. The obtained parallel speed-up is shown in fig. 9.22(a) and the calculation times are shown in fig. 9.22(b). Here, it is clear that the hybrid method outperforms the distributed-only technique. Running on 48 cores for example, the MPI-only method yields 82% parallel efficiency when compared to 12 processes, while the hybrid method yields 89%. When using 96 cores the differences are even more noticeable with the MPI-only method yielding 64% efficiency and the hybrid method with 24 MPI processes and four threads resulting in 78% efficiency (when compared to 12 processes). Another important benefit from using the hybrid method is that the memory usage per cores is much lower than the MPI-only method. It was previously described that with our current computing hardware, the maximum number of processes that could be tested when using the MPI-only implementation with the polar-based domain decomposition, was 72. By using the hybrid method it was possible to run up to 192 cores with important gains in terms of speed-up and with memory consumption maintained at the level of 48 MPI processes shown in fig. 9.21.

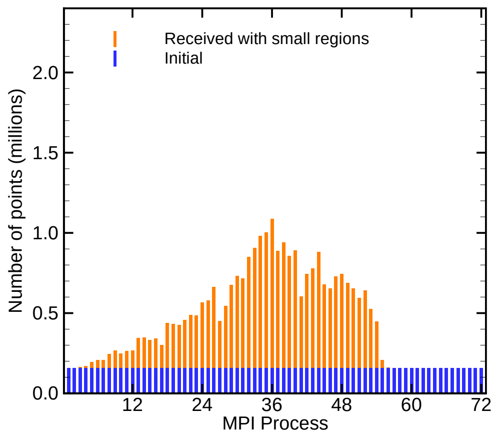


(a) 12 processes using slice-based domain decomposition

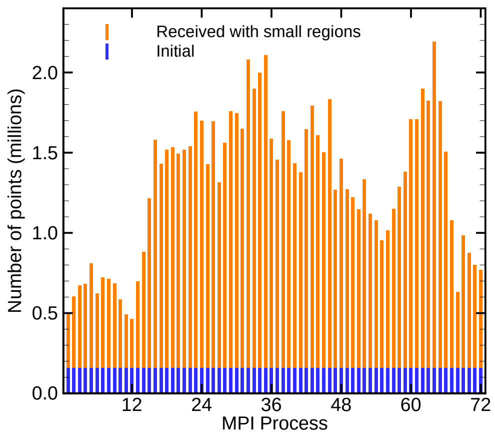


(b) 12 processes using polar-based domain decomposition

Figure 9.19: Number of points received per domain for OSF case running on 12 processes.

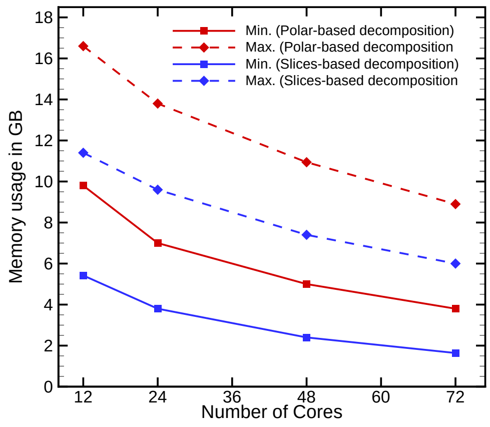


(a) Slice-based domain decomposition

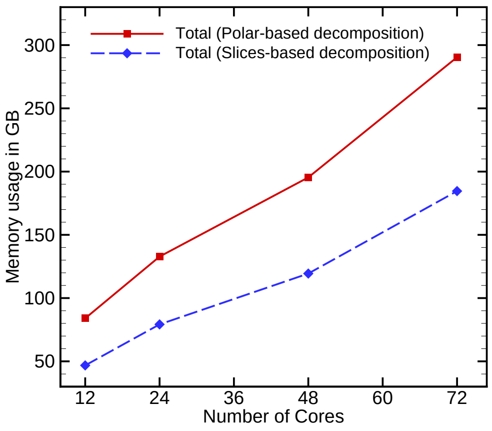


(b) Polar-based domain decomposition

Figure 9.20: Points per domain running on 72 processes for OSF case with distributed implementation (points received with big regions were omitted as they showed a similar trend)

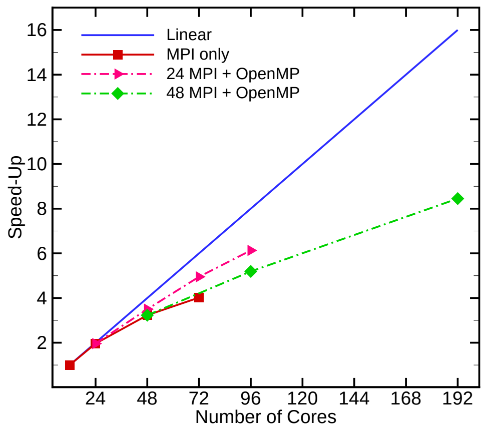


(a) Memory usage per process

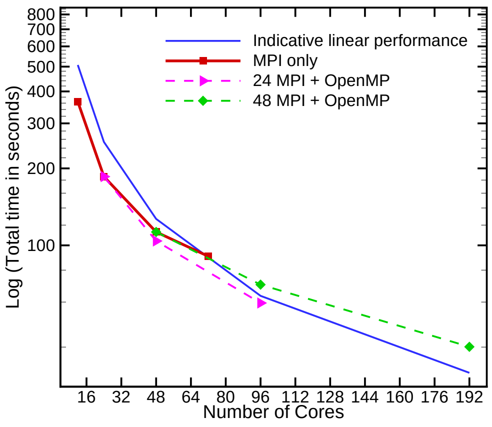


(b) Total memory usage

Figure 9.21: Memory usage for the OSF case using two types of domain decomposition



(a) Hybrid method speed-up



(b) Hybrid method calculation time (Log scale)

Figure 9.22: Parallel performance of the hybrid (MPI/OpenMP) method for OSF case.

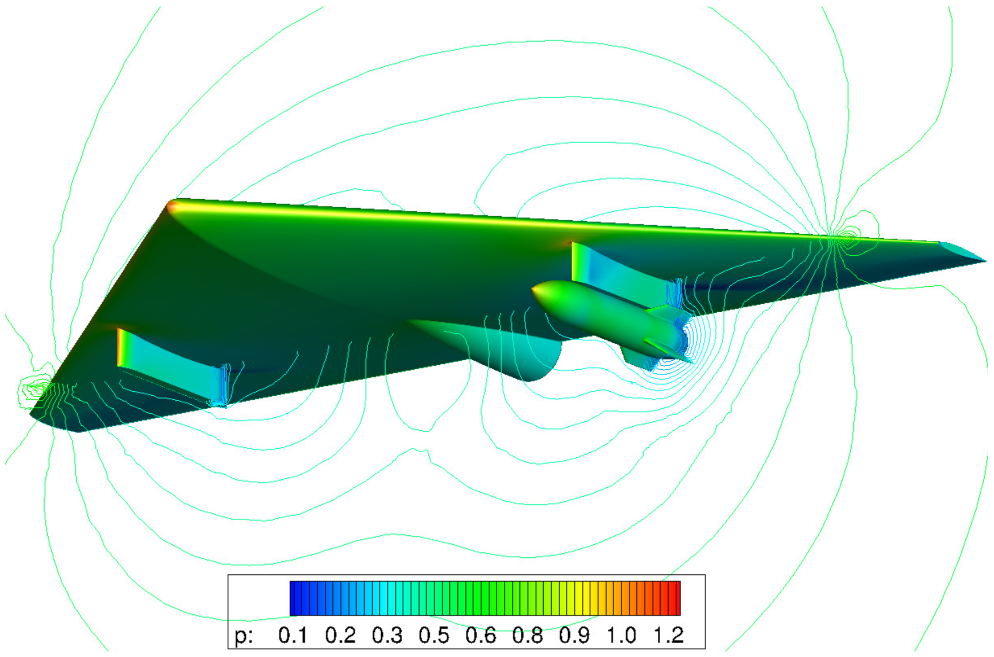
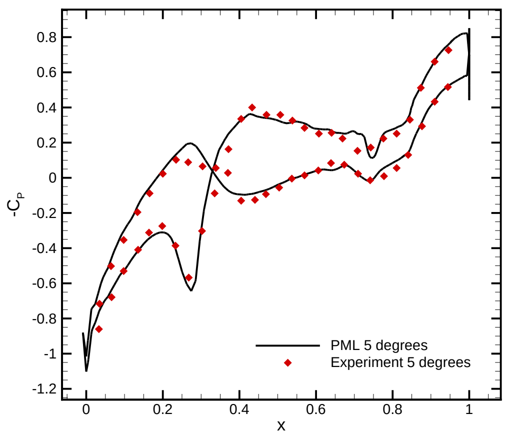


Figure 9.23: Store-drop test case flow solution at t=0.

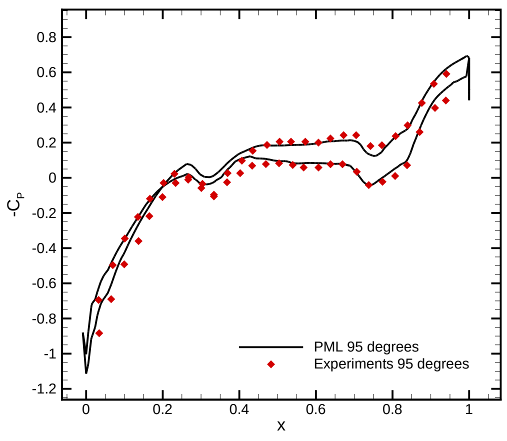
9.6 Results for the Transient Store-Drop Case

The store-drop test case was calculated using the preprocessor and solver in coupled mode, running on 1 to 32 processes. The flow conditions for the test case are inviscid flow at a Mach number of 1.2 and an angle of attack of zero degrees. To start the simulation, the steady-state flow was calculated at the carriage position. The convergence criteria for the steady calculation is a reduction of the residuals of five orders of magnitude. The calculated flow-field and surface pressures at the initial position are shown in Fig. 9.23. Figures 9.24(a) and 9.24(b) show the surface pressure coefficient on the store at *t=0s* compared to the experimental data from [121]. The figures correspond to the data at two cross-section planes on the surface of the store. Both planes cut the store longitudinally and are rotated along the store axis. The first one is rotated by 5 degrees, while the second is rotated 95 degrees. Table 9.7 summarizes the total forces and moments acting on the store compared to the experimental data from [120]. These results show good agreement between PML and the experimental data. The main differences observed are most likely due to the fact that the PML values correspond to an inviscid simulation, and the fact that the store sting used in the wind tunnel experiments was not included in the PML model.

After the steady-state solution has converged, the 6-DOF routine started calculating the successive movement of the store using the loads calculated by the flow solver. The time step used for the simulation is maintained constant at $\Delta t = 0.002$ s and the final time for the simulations was $t = 0.35$ s.



(a) Slice 1. 5° from XZ plane



(b) Slice 2. 95° from XZ plane

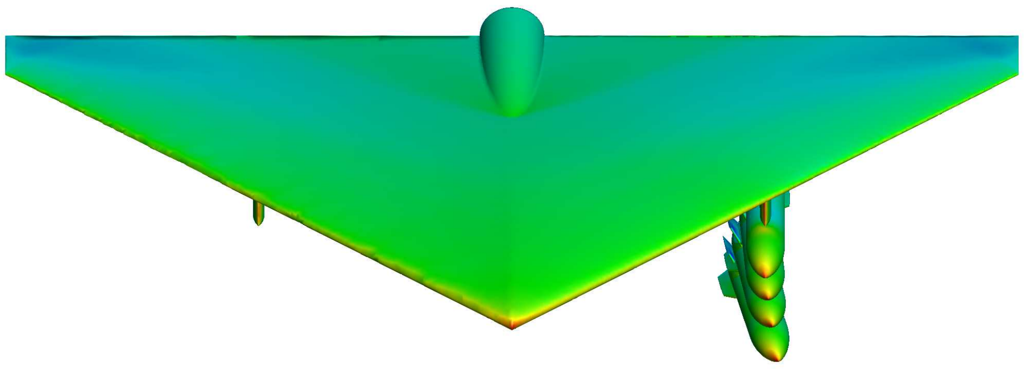
Figure 9.24: Surface pressure coefficient on the store at $t = 0s$, measured through two different planes

Table 9.7: Aerodynamic forces and moments at carriage position (t=0s) for store-drop test case. (All forces in N and moments in $N \cdot m$)

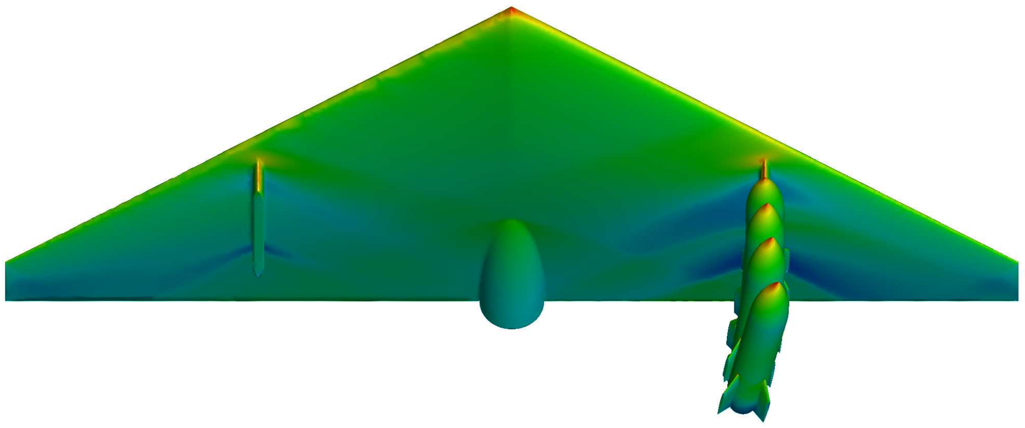
	Fx	Fy	Fz	Mx	My	Mz
Wind Tunnel	4893.0	2504.8	-2451.9	107.2	-3570.0	2896.7
PML	4693.7	2609.4	-2412.5	122.4	-3703.2	3081.8

Figure 9.25 shows the movement of the store as it falls from its carriage position, at four different times during the simulation. The computed results for the relative displacement and velocities of the store centre of gravity are compared to the experimental data in Fig. 9.26. Similar plots for the angular position and angular rates are shown in Fig. 9.27. From these results it is clear that the correlation between the calculated and experimental data for the displacement of the centre of gravity (CG) is excellent. The vertical movement was expected to correlate well, as it is dominated first by the ejector forces and then by the gravitational force acting on the store. The lateral and longitudinal movements on the other hand, are very much dependent on aerodynamic loads and the numerical method provides good predictions for these values. Initially, the store moves inwards towards the centre of the wing and then, it progressively moves outwards. The longitudinal displacement increases steadily as the store falls and this is well predicted by PML.

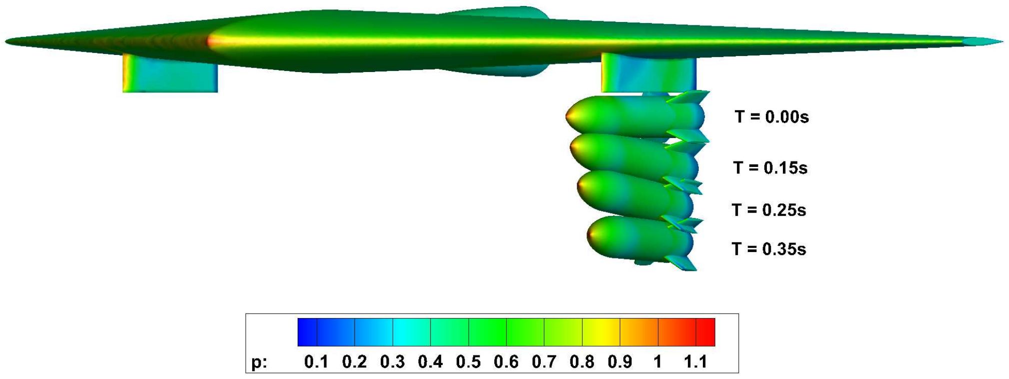
In terms of the angular attitude of the store, it can be seen that the yaw angle is well predicted throughout the simulation. On the other hand, both the roll and pitch angles are slightly over-predicted by PML. The maximum discrepancy in the roll angle is of 0.6 degrees at $t = 0.35$ s. The maximum pitch angle calculated by PML is 5.85 degrees, compared to an experimental value of 5.45 degrees. Even so, PML correctly



(a)

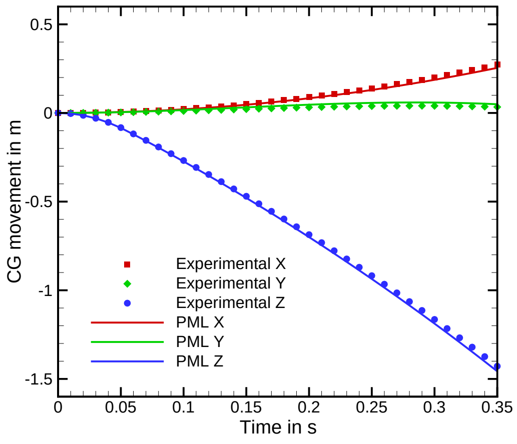


(b)

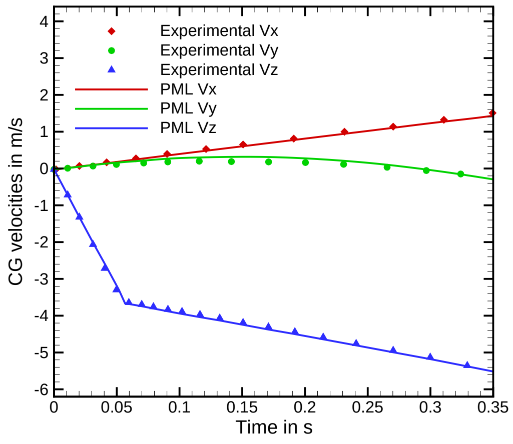


(c)

Figure 9.25: Trajectory of the store during 6-DOF simulation.

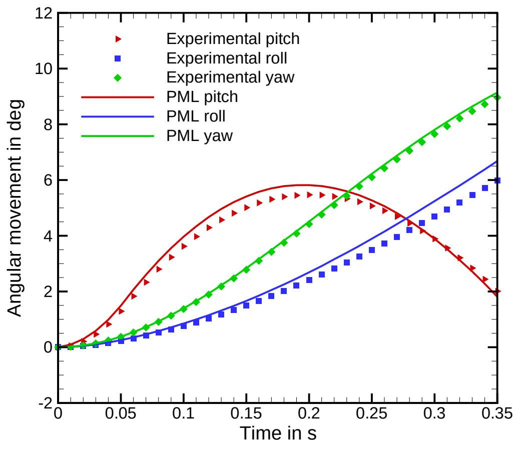


(a) Relative displacement of the CG

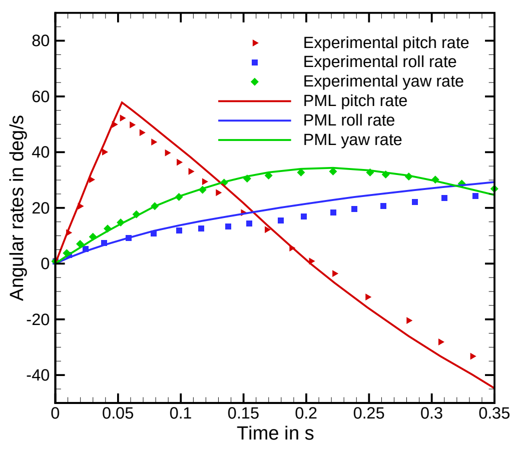


(b) Velocities of the CG

Figure 9.26: Calculated trajectory and velocities of the centre of gravity.



(a) Angular movement of the store

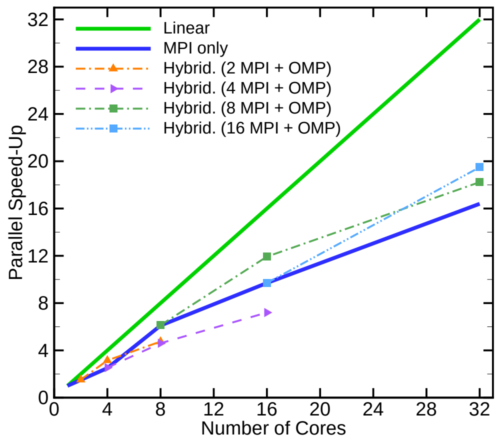


(b) Angular rates of the store

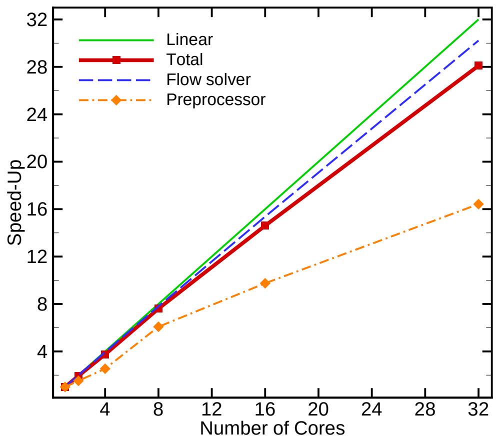
Figure 9.27: Calculated angular movements and rates.

predicts the time of this peak value to be around $t = 0.2$ s and the trend of both the pitch and roll angles follow the experimental values correctly. There are two main reasons that are likely to be behind the discrepancies on the angular movement of the store. The first is the previously mentioned fact that the calculation was inviscid and that the PML model did not include the store sting. The other reason is the Euler integration method used to predict the velocities and positions in the 6-DOF method. It is well known that the error of this first-order method is proportional to the step size used. Because of time constraints, a study of sensitivity of the calculations to the time-step was not performed.

The parallel efficiency of the method was also evaluated. Figure 9.28(a) shows the parallel speed-up obtained for the preprocessor after the first iteration of the flow solver.



(a) Preprocessor speed-up for the store-drop test case



(b) Full time-step speed-up for the store-drop test case

Figure 9.28: Parallel performance of the method for store-drop case.

This means that the preprocessor is using the domain decomposition given by the flow solver. In the figure, the rate of increase of the speed-up for the parallel preprocessor varies, as a different number of processes are used. This is due to the change in load balance given by the domain decomposition. Using four processes for example, the decompositions provided by the solver yields poor load balancing, resulting in a parallel efficiency 64%. When increasing the number of processes to eight however, the load balance proves to be much better and the efficiency increases to 76%. When using 16 and 32 processes similar efficiencies are obtained and the parallel method proves to be scalable as the speed-up continues to climb when the number of processes is increased. The figure also shows the parallel performance when the hybrid method is used. As with the previous test cases, the hybrid method improves the performance of the preprocessor. It is interesting to note however, that this is true only for cases when the limiting factor for the MPI-only implementation is the load balance. In this particular case, going from four to eight MPI processes results in a better load balance as it was noted previously. For this reason, starting with four MPI processes and adding hybrid threads results in lower performance than using eight MPI processes.

Finally, figure 9.28(a) shows the average parallel speed-up for a full time-step of the simulation. This is the total time needed to calculate one iteration of the preprocessor-solver loop as explained in Section 8.1. A hybrid implementation of the flow solver had not being completed at the time of writing, hence the transient simulation of the store release case was performed using the MPI-only method. The flow solver again shows strong parallel performance with a minimum efficiency of 95% on 32 processes. The computational cost per time-step of the flow solver is on average about 7.3 times higher than the cost of the preprocessor for this test case. For this reason, the overall

Table 9.8: Average calculation per real time-step for store-drop case. (All times in seconds)

	1 CPU	2 CPU	4 CPU	8 CPU	16 CPU	32 CPU
Flow Solver	3483.8	1751.2	884.2	446.6	226.7	115.2
Preprocessor	346.5	228.3	136.4	57.0	35.5	21.1

parallel performance of the method is quite good. The average values of the calculation time per iteration are shown in Table 9.8. The fact that the flow solver proves to be this much more expensive than the preprocessor justifies the decision of making the preprocessor work with the domain decomposition given by the flow solver.

In the future, however, different domain decomposition methods for the preprocessor should be investigated, as well as methods for changing the load balance of the flow solver as the transient simulations progress.

Chapter 10

Conclusions and Future Work

In this thesis, the implementation of parallel methods for the simulation of turbulent flows using a meshless scheme was investigated. The research started by implementing the full Navier-Stokes equations onto an existing Euler meshless solver. The extension of the original Euler meshless scheme was carried out by using central differences to include the second derivatives (diffusive terms) of the governing equations. Turbulence modelling was introduced by using Reynolds averaging with the Spalart-Allmaras closure equation. The extension to the original solver was tested with four different test cases and validated by comparing the meshless scheme to computational results from an established finite-volume solver and experimental data. Results showed good agreement for flows in both laminar and turbulent regimes. Future work should include the implementation of Large-Eddy Simulation(LES) and Detached-Eddy Simulation (DES) models into PML to assess the capabilities of the meshless method to tackle more advanced problems.

The second part of the thesis was dedicated at implementing parallel algorithms into the meshless flow solver to improve the computational efficiency. The method uses graph partitioning techniques to perform the domain decomposition and non-blocking MPI commands to perform the parallel communication. In an effort to reduce the parallel overheads, two techniques were used. The first is that the list of points that need to be communicated are stored first. This way, their data is communicated as soon as their variables are updated and processors are free to update the rest of their points while the parallel exchange takes place. The second technique is that only local data is used to form the preconditioner needed to solve the linear system of equations that arises from the implicit integration method. The parallel meshless flow solver was tested on different machines, ranging from a small cluster of workstations to dedicated HPC machines. Two test cases that resemble real industrial applications were studied, a wing in transonic flow conditions and a generic aircraft configuration with a grid size of more than 40 million points. In both cases, the parallel flow solver shows good scalability with the second test case in particular yielding over 90% parallel efficiency on 1024 processors when compared to 64. Results also showed that the asynchronous mode of operation yields an average of 10% speed improvement over synchronous communication techniques. During the tests, it was seen that when running in parallel, the use of an incomplete preconditioner affects the convergence characteristics of the linear solver. This resulted in the number of iterations needed to achieve convergence being different when using a different number of processors. Even so, the variations of the total number of iterations were small and the total reduction in calculation time was significant for both test cases, showcasing the capabilities of the method to perform big calculations.

Finally, the development of a parallel automatic preprocessing tool for the selection of stencils was presented. The preprocessor takes different overlapping grids that can be stationary, or moving relative to one another, and selects meshless stencils which are used by the flow solver to solve the governing equations. Three types of implementation were described. The first one follows a distributed memory approach using MPI commands, the second one uses a shared memory approach with OpenMP directives and the third is a hybrid method that combines the two previous ones.

The distributed method first decomposes the domain according to the geometry of the problem, by using slices along the x coordinates or by using polar coordinates. It then blanks the points internal to solid boundaries, and identifies which of the locally stored points are to be sent to other processors for their stencil search. After the parallel communication takes place, the method finds candidates for each of the points in the domain, sorts these candidates according to a “merit” function and finally selects the stencils that are sent to the flow solver. The shared memory method assumes that all processors have access to the same data and parallelises the operations that contain loops by dynamically assigning chunks of data to each of the processors at run-time. The hybrid method combines the two previous approaches by dividing all the available processors into groups. The method follows the distributed approach described before in between these groups, but internally, it follows the shared memory idea by parallelising the work in between the processors that form the group. When performing parallel unsteady simulations where the preprocessor and flow solver are coupled together, both of them use the same domain decomposition. This is done in an effort to avoid introducing the overheads from performing the decomposition.

The first set of tests carried out with the preprocessor showed that the sorting of the candidates was the most expensive operation of the method. Several different sorting algorithms were tested in an effort to reduce this cost and it was found that the Quicksort algorithm proves the fastest, reducing the sorting times by almost an order of magnitude when compared to the original “brute-force” sorting algorithm.

Four test cases were then successfully computed, showcasing the method as a powerful tool for applications where different component grids can move in relation to one another. Of particular relevance, the fourth test case was the simulation of a store being dropped from an aircraft. In this case, the parallel preprocessor and flow solver were coupled with a 6-DOF module that calculates the motion of the store from the aerodynamic loads. The method correctly predicted the movement of the store when compared to experimental data. The proposed implementations of the parallel preprocessor were successful in accelerating the calculation of meshless stencils and allowed for the study of big cases resembling real industrial applications, that would otherwise be impossible to compute using serial computers.

There were a few difficulties that were found in each of the implementations. When testing the distributed memory method, three main problems were found. The first one is the fact that the load balancing techniques included in the proposed implementation proved far from perfect, thus hampering the parallel speed-up. This is due to the fact that it is not possible to predict the computational cost per point of the different operations performed by the preprocessor. The test cases proved that in fact, the computational cost of the operations can vary from point to point by as much as three orders of magnitude. The second problem found is that the communication overheads can be significant, depending on the case to be tested and the number of processors used. The final one is that the memory usage grows quickly when using more processors as they need store data that was initially assigned to other processors. The main difficulty found with the shared memory method is that the scheme is limited by the physical memory of the machine to be used. The second is the fact that due to the design of current computing hardware the access to the memory becomes slower when using more processors.

By using the hybrid method however, some of the difficulties of both the distributed and shared memory approaches are alleviated. Modern (and most likely all future) hardware is designed around the use of chips that contain several processors sharing a common memory bank. Most HPC machines use several of these chips connected together. For this reason, it makes sense to use the shared memory method within one chip and use the distributed approach among several chips. It was found that using the hybrid method and combining several distributed processes with several threads each, yields the best results both in terms of speed-up and memory usage.

When using the distributed or hybrid implementations, the polar-based decomposition proves to be considerably faster than the slice-based partitioning on the cases tested, but the memory usage of the polar based method is much higher than using the slices technique. The recommendation then is to use the polar based decomposition unless the problems are limited by memory usage.

As a final conclusion to this work, we can safely state that the initial objectives were met. Furthermore, to the best of the authors knowledge, this is so far the only published implementation of parallel methods applied to the problem of selecting stencils and solving the flow governing equations using a meshless scheme aimed at aerodynamic problems with movable geometry.

Future Work

Future work needs to include the parallel implementation of boundary re-definitions when different bodies intersect. At the moment this capability is not included in the parallel method, but if this scheme is to become valuable for real-life industrial applications this problem needs to be addressed. Future work should also include research on how to optimize the size of the search regions used to select the candidates for the stencil of any given point. This can have a major impact on the efficiency of the parallel preprocessor. In the original method, the search regions for points located in one processor can grow too big into the domain of other processors, especially in cases where small stencils overlap big stencils close to inter-process boundaries. This results in processors having to exchange big amounts of data with one another, thus hampering memory scalability. In an effort to reduce this problem, in this work the method was slightly changed, with the search regions growing only to the location of the central point of an overlapping stencil, instead of growing the regions to the maximum dimension of the overlapping stencil. This simple change had a positive effect of the method, with the parallel preprocessor needing to communicate less data to provide the solver with stencils that produce virtually identical results to the ones calculated with stencils selected using the original method. Other possible solutions that should be tested in the future include using hole-cutting techniques on background grids to reduce the number of stencils of detailed component grids (which are usually small) that overlap the background stencils (which are normally big). One final suggestion is to use data from previous time-steps during a transient simulation to limit the size of the search regions.

The final suggestion for future work is that a shared memory implementation and in turn, a hybrid MPI/OpenMP implementation of the flow solver needs to be completed, in order for the hybrid preprocessing method to work in transient simulations when the preprocessor is coupled with the flow solver. Progress has been made towards this goal, but further testing is needed to provide conclusive results of the capabilities of the hybrid flow solver.

Bibliography

[1] Thompson, J. F., Soni, B. K., and Weatherill, N. P., *Handbook of Grid Generation*, CRC Press, 1999.

[2] Mondal, P., Munikrishna, N., and Balakrishnan, N., "Cartesian-Like Grids Using a Novel Grid-Stitching Algorithm for Viscous Flow Computations," *Journal of Aircraft*, Vol. 44, 2007.

[3] Baker, T. J., "Mesh Generation: Art or Science?" *Progress in Aerospace Sciences*, Vol. 41, 2005, pp. 29–63.

[4] Murman, S. M., Aftosmis, F. J., and Berger, M., "Simulations of Store Separation from an F/A-18 with a Cartesian Method," *Journal of Aircraft*, Vol. 41, 2004.

[5] Zeeuw, D. D. and Powell, K. G., "An Adaptively Refined Cartesian Mesh Solver for the Euler Equations," *Journal of Computational Physics*, Vol. 104, 1993, pp. 56–68.

[6] Clarke, D. K., Salas, M. D., and Hassan, H. A., "Euler Calculations for Multi Element Airfoils Using Cartesian Grids," *AIAA Journal*, Vol. 24, 1986, pp. 353–358.

[7] Meyer, M., *Simulation of Complex Turbulent Flows on Cartesian Adaptive Grids*, Ph.D. thesis, Technical University of Munich, 2013.

[8] Lee, J. D. and Ruffin, S. M., "Solution of Turbulent Flow using a Cartesian Grid Based Numerical Scheme," *International Conference on computational and Information Sciences Aerospace Systems Engineering*, 2009, pp. 612–472.

[9] Lee, J. D., *Development of an Efficient Viscous Approach in a Cartesian Grid Framework and Application to Rotor-Fuselage Interaction*, Ph.D. thesis, Georgia Institute of Technology, 2006.

[10] Coirier, W. J. and Powell, K. G., "Solution-Adaptive Cartesian Cell Approach for Viscous and Inviscid Flows," *AIAA Journal*, Vol. 34, 1996, pp. 938–945.

[11] Wang, Z. J. and Chen, R. F., "Anisotropic Solution-Adaptive Viscous Cartesian Grid Method for Turbulent Flow Simulation," *AIAA Journal*, Vol. 4, 2002.

[12] Frymier, P. D., Hassan, H. A., and Salas, M. D., "Navier-Stokes Calculations Using Cartesian Grids, 1: Laminar Flows," *AIAA Journal*, Vol. 26, 1988, pp. 1181–1188.

[13] Mavriplis, D. J., "Adaptive Mesh Generation for Viscous Flows Using Delaunay Triangulation," *Journal of Computational Physics*, Vol. 90, 1990, pp. 271–291.

[14] Peraire, J., Peiro, J., and Morgan, K., "Adaptive Remeshing for Three-Dimensional Compressible Flow Computations," *Journal of Computational Physics*, Vol. 103, 1992, pp. 269–285.

[15] Hassan, O., Probert, E. J., Morgan, K., and Peraire, J., "Mesh Generation and Adaptivity for the Solution of Compressible Viscous High Speed Flows," *International Journal for Numerical Methods in Engineering*, Vol. 148, 1995, pp. 1123–1148.

[16] Zagaris, G., Campbell, M., Bodony, D. J., Shaffer, E., and Brandyberry, M., "A Toolkit for Parallel Overset Grid Assembly Targeting Large-Scale Moving Body Aerodynamic Simulations," *19th International Meshing Roundtable*, 2010.

[17] Alleaume, A., Francez, L., Loriot, M., and Maman, N., "Large Out-of-Core Tetrahedral Meshing," *Proc. 16th International Meshing Roundtable*, Vol. Sandia National Laboratory, 2007.

[18] Andrae, H., Ivanov, E., Gluchshenko, O., and Kudryavtsev, A., "Automatic Parallel Generation of Tetrahedral Grids by Using a Domain Decomposition Approach," *Journal of Computational Mathematics and Mathematical Physics*, Vol. 48, 2008, pp. 1448–1457.

[19] Chrisochoides, N., *Parallel Mesh Generation*, Springer, 2005.

[20] de Cougny, H. L., Shephard, M. S., and Ozturan, C., "Parallel Three-Dimensional Mesh Generation," *Computing Systems in Engineering*, Vol. 5, 1994, pp. 311–323.

[21] Ivanov, E. G., Andrae, H., and Kudryavtsev, A., "Domain Decomposition Approach for Automatic Parallel Generation of Tetrahedral Grids," *International Mathematical Journal Computational Methods in Applied Mathematics*, Vol. 6, 2006, pp. 178–193.

[22] Rai, M. H., "Navier-Stokes Simulations of Rotor/Stator Interaction using Patched and Overlaid Grids," *Journal of Propulsion and Power*, Vol. 3, 1987, pp. 387–396.

[23] Mathur, S., "Unsteady Flow Simulations using Unstructured Sliding Meshes," *AIAA Journal*, 1994.

[24] Wang, M. H., Calabrese, R. V., and Bakker, A., "Effect of Reynolds Number on the Flow Generated by a Pitched Blade Turbine," *45th CSChE Conference, Qubec City*, 1995.

[25] Nam, H. J., Park, Y., and Kwon, O. J., "Simulation of Unsteady Rotor-Fuselage Aerodynamic Interaction Using Unstructured Adaptive Meshes," *Journal of the American Helicopter Society*, Vol. 51, 2006, pp. 141–149.

[26] Steijl, R. and Barakos, G., "Computational Investigation of Rotor-Fuselage Interactional Aerodynamics using Sliding-Plane CFD Method," *AIAA Journal*, Vol. 47, 2009, pp. 2143–2157.

[27] Murthy, J. Y., Mathur, S. R., and Choudhury, D., "CFD Simulation of Flows in Stirred Tank Reactors Using a Sliding Mesh Technique," *Mixing 8, Proceedings of The Eighth European Conference on Mixing, Institution of Chemical Engineers*, Vol. Symposium Series No. 136, 1994, pp. 341–348.

[28] Adami, P. and Martelli, F., "Three-Dimensional Unsteady Investigation of HP Turbine Stages," *Journal of Power and Energy*, Vol. 220, 2006, pp. 155–167.

[29] Rivera, C. A., Heniche, M., Bertrand, F., Glowinski, R., and Tanguy, P., "A Parallel Finite Element Sliding Mesh Technique for the Simulation of Viscous Flows in Agitated Tanks," *International Journal for Numerical Methods in Fluids*, Vol. 69, 2012, pp. 653–670.

[30] Gomez-Iradi, S., *CFD for Horizontal Axis Wind Turbines*, Ph.D. thesis, School Of Engineering. University of Liverpool, 2009.

[31] McNaughton, J., Afgan, I., Apsley, D. D., Rolfo, S., Stallard, T., and Stansby, K., "A Simple Sliding-Mesh Interface Procedure and its Application to the CFD Simulation of a Tidal-Stream Turbine," *International Journal for Numerical Methods in Fluids*, Vol. 74, 2014, pp. 250–269.

[32] Steijl, R. and Barakos, G., "Sliding Mesh Algorithm for CFD Analysis of Helicopter Rotor-Fuselage Aerodynamics," *International Journal for Numerical Methods in Fluids*, Vol. 58, 2008, pp. 527–549.

[33] Steger, J. L., Dougherty, F. C., and Benek, J. A., "A Chimera Grid Scheme," *Advances in Grid Generation ASME FED*, Vol. 5, 1983, pp. 59–69.

[34] Suhs, N. E., Rogers, S. E., and Dietz, W. E., "PEGASUS 5: An Automatic Pre-Processor for Overset-Grid CFD," *AIAA Paper 2002-3186. AIAA 32ND Fluid Dynamics Conference, St. Louis*, 2002.

[35] Meakin, R. L., *Composite Overset Structured Grids*, CRC Press, 1999.

[36] Benoit, C., Jeanfaivre, G., and Canonne, E., "Synthesis of ONERA Chimera Method Developed in the Frame of CHANCE Program," *31st European Rotorcraft Forum. Florence, Italy*, 2005.

[37] Boger, D. and Dreyer, J., "Prediction of Hydrodynamic Forces and Moments for Underwater Vehicles Using Overset Grids," *AIAA Paper 2006-1148. 44th AIAA Aerospace Sciences Meeting And Exhibit*, 2006.

[38] Boger, D., Noack, R. W., and Amar, R. W., "Overset Grid Applications in Hypersonic Flow Using the DPLR Flow Solver," *AIAA Paper 2008-921. 46th AIAA Aerospace Sciences Meeting And Exhibit*, 2008.

[39] Tarhan, E. and Kavsaoglu, M. S., "Parallel Overset-Grid Euler Solution of Generic Wing Pylon and Finned Store," *Journal of Aircraft*, Vol. 42, 2005.

[40] Kao, K. H. and Liou, M. S., "Advance in Overset Grid Schemes: From Chimera to DRAGON Grid," *AIAA JOURNAL*, Vol. 33, 1995.

[41] Wissink, A. M. and Meakin, R. L., "Computational Fluid Dynamics with Adaptive Overset Grids on Parallel and Distributed Computer Platforms," *International Conference on Parallel and Distributed Computing*, 1998.

[42] Prewitt, N. C., Belk, D. M., and Shyy, W., "Parallel Computing of Overset Grids for Aerodynamic Problems with Moving Objects," *Progress in Aerospace Sciences*, Vol. 36, 2000, pp. 117–172.

[43] Zagaris, G., Campbell, M., Bodony, D. J., Shaffer, E., and Brandyberry, M. D., "A Toolkit for Parallel Overset Grid Assembly Targeting Large-Scale Moving Body Aerodynamic Simulations," *Proceedings of the 19th International Meshing Roundtable*, 2010, pp. 385–401.

[44] Landmann, B. and Montagnac, M., "A Highly Automated Parallel Chimera Method for Overset Grids Based on the Implicit Hole Cutting Technique," *International Journal for Numerical Methods in Fluids*, Vol. 66, 2010, pp. 778–804.

[45] Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "An Implicit Meshless Method for Application in Computational Fluid Dynamics," *International Journal for Numerical Methods in Fluids*, Vol. 71, 2012, pp. 1007–1028.

[46] Kennett, D. J., Angulo, J., Timme, S., and Badcock, K. J., "Semi-Meshless Stencil Selection on Three-Dimensional Anisotropic Point Distributions with Parallel Implementation," *AIAA Paper 2013–0867. Presented at the 51st AIAA Aerospace Sciences Meeting, Grapevine, Texas*, 2013.

[47] Lucy, L. B., "A Numerical Approach to the Testing of the Fission Hypothesis," *Astronomical Journal*, Vol. 93, 1977, pp. 1013–1024.

[48] Monaghan, J. J. and Gingold, R. A., "Shock Simulation by the Particle Method SPHs," *Journal of Computational Physics*, Vol. 52, 1983, pp. 374–389.

[49] Swegle, J. W., Hicks, D. L., and Attaway, S. W., "Smoothed Particle Hydrodynamics Stability Analysis," *Journal of Computational Physics*, Vol. 16, 1995, pp. 123–134.

[50] Dyka, C. T., Randles, P. W., and Ingel, P., "Stress Points for Tension Instability in SPH," *International Journal for Numerical Methods in Engineering*, Vol. 40, 1997, pp. 2325–2341.

[51] Liu, G. R., *Meshfree Methods: Moving Beyond the Finite Element Method*, CRC Press, 2003.

[52] Liu, G. R. and Gu, Y. T., *An Introduction to Meshfree Methods and Their Programming*, SPRINGER, 2005.

[53] Liu, M. B. and Liu, G. R., "Smoothed Particle Hydrodynamics:A Meshfree Particle Method," *World Scientific Publishing*, Vol. 620, 2003, pp. 89–119.

[54] Li, S. and Liu, W. K., "Meshless and Particle Methods and their Applications," *Applied Mechanics Review*, Vol. 55, 2002, pp. 1–34.

[55] Bernal, F. M., *Meshless Methods for Elliptic and Free-Boundary Problems*, Ph.D. thesis, Universidad Carlos III de Madrid, 2008.

[56] Nayroles, B., Touzot, G., and Villon, P., "Generalizing the Finite Element Method: Diffuse Approximation and Diffuse Elements," *Computational Mechanics*, Vol. 10, 1992, pp. 307–318.

[57] Breitkopf, P., Rassineux, A., Savignat, J. M., and Villon, P., "Integration Constraint in Diffuse Element Method," *Computer Methods in Applied Mechanics and Engineering*, Vol. 193, 2004, pp. 1203–1220.

[58] Belytschko, T., Lu, Y., and Gu, L., "Element-Free Galerkin Methods," *International Journal for Numerical Methods in Engineering*, Vol. 37, 1994, pp. 229–256.

[59] Viana, S. A., Rodger, D., and Lai, H. C., "Overview of Meshless Methods," *International Compumag Society Newsletter*, Vol. 14, 2007.

[60] Atluri, S. N. and Zhu, T., "A new Meshless Local Petrov-Galerkin (MLPG) Approach In Computational Mechanics," *Computational Mechanics*, Vol. 22, 1998, pp. 117–127.

[61] Katz, A., *Meshless Methods for Computational Fluid Dynamics*, Ph.D. thesis, Stanford University, 1999.

[62] Atluri, S. N., Kim, H. G., and Cho, J. Y., "A Critical Assessment of the Truly Mesh-less Local Petrov-Galerkin (MLPG) and Local Boundary Integral Equation (LBIE) Methods," *Computational Mechanics*, Vol. 24, 1999, pp. 348–372.

[63] Lin, H. and Atluri, S. N., "The Meshless Local Petrov-Galerkin (MLPG) Method for Solving Incompressible Navier-Stokes Equations," *Computer Modeling in Engineering and Sciences*, Vol. 2, 2001, pp. 117–142.

[64] Batina, J. T., "A Gridless Euler/Navier-Stokes Solution Algorithm for Complex Aircraft Applications," *AIAA Paper 1993-0333. AIAA 31ST Aerospace Sciences Meeting and Exhibit, Reno, NV,*, 1993.

[65] Oñate, E., Idelsohn, S., Zienkiewicz, O. C., Taylor, R. L., and Sacco, C., "A Stabilized Finite Point Method For Analysis of Fluid Mechanics Problems," *Computational Methods in Applied Mechanical Engineering*, Vol. 139, 1996, pp. 315–346.

[66] Oñate, E., Idelsohn, S., Zienkiewicz, O. C., and Taylor, R. L., "A Finite Point Method in Computational Mechanics. Applications to Convective Transport and Fluid Flow," *International Journal for Numerical Methods In Engineering*, Vol. 39, 1996, pp. 3839–3866.

[67] Oñate, E. and Idelsohn, S., "A mesh-free finite point method for advective-diffusive transport and fluid flow problems," *Computational Mechanics*, Vol. 21, No. 4–5, 1998, pp. 283–292.

[68] Katz, A. and Jameson, A., "A Meshless Volume Scheme," *AIAA Paper 2009-3534. 19th AIAA Computational Fluid Dynamics. 22 - 25 June 2009, San Antonio, Texas*, 2009.

[69] Shirazaki, M. and Yagawa, G., "Large-Scale Parallel Flow Analysis Based on Free Mesh Method: a Virtually Meshless Method," *Computational Methods in Applied Mechanical Engineering*, Vol. 174, 1999, pp. 419–431.

[70] Gunther, F., Liu, W., Diachin, D., and Christon, M., "Multi-Scale Meshfree Parallel Computations for Viscous, Compressible Flows," *Computational Methods in Applied Mechanical Engineering*, Vol. 190, 2000, pp. 279–303.

[71] Li, J. and Hon, Y., "Domain Decomposition for Radial Basis Meshless Methods," *Numerical Methods for Partial Differential Equations*, Vol. 20, 2004, pp. 450–462.

[72] Kosec, G., Depolli, M., Rashkovska, A., and Trobec, R., "Super Linear Speedup in a Local Parallel Meshless Solution of Thermo-Fluid Problems," *Computers and structures*, Vol. 133, 2013, pp. 30–38.

[73] Yagawa, G. and Shirazaki, M., "Parallel Computing for Incompressible Flow Using a Nodal-Based Method," *Computational Mechanics*, Vol. 23, 1999, pp. 209–217.

[74] Fujisawa, T., Inaba, M., and Yagawa, G., "Parallel Computing of High-Speed Compressible Flows Using a Node-Based Finite-Element Method," *International Journal for Numerical Methods in Engineering*, Vol. 58, 2003, pp. 481–511.

[75] Flynn, M., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. 21, 1972, pp. 948–960.

[76] Guerrero, M. S., *Parallel Multigrid Algorithms for Computational Fluid Dynamics and Heat Transfer*, Ph.D. thesis, Department of Machines and Thermal Engines. Universitat Politecnica de Catalunya, 2000.

[77] Lewis, T. G. and El-Rewini, H., *Introduction to Parallel Computing*, Prentice Hall, 1992.

[78] Davis, D. E., *A Parallel Computational Fluid Dynamics Unstructured Grid Generator*, Ph.D. thesis, Graduate School of Engineering. Air Force Institute of Technology, 1993.

[79] Amdahl, G., "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of AFIPS Conference*, 1967, pp. 483–485.

[80] Gustafson, J. L., "Reevaluating Amdahl's Law," *Communications of the ACM*, Vol. 31, 1988, pp. 532–533.

[81] Baldwin, B. S. and Barth, T. J., "A One-Equation Turbulence Transport Model for High Reynolds Number Wall-Bounded Flows," *NASA Tech. Memo*, Vol. 102847, 1990.

[82] Yakhot, V., Orszag, S. A., Gatski, T. B., and Speziale, C. G., "Development of turbulence models for shear flows by a double expansion technique," *Physics of Fluids A*, Vol. 4, 1992, pp. 1510–1520.

[83] Bertin, J. J., Periaux, J., and Ballmann, J., *Advances in Hypersonics: Modeling Hypersonic Flows*, Birkhauser Boston, 1992, ISBN: 0817636633.

[84] Chen, C. J., *Fundamentals Of Turbulence Modelling*, CRC Press, 1997.

[85] Libby, P. A., *An Introduction To Turbulence*, CRC Press, 1996.

[86] Barakos, G., *Study of Unsteady Aerodynamics Phenomena Using Advanced Turbulence Closures*, Ph.D. thesis, Faculty of Technology. University of Manchester, 2009.

[87] Spalart, P. R. and Allmaras, S. R., "A One-Equation Turbulence Model for Aerodynamic Flows," *AIAA Paper 92-0439*, 1992.

[88] Belytschko, T., Krongauz, Y., Organ, D., Fleming, M., and Krysl, P., "Meshless Methods: an Overview and Recent Developments," *Computational Methods in Applied Mechanical Engineering*, Vol. 139, 1996, pp. 3–47.

[89] Roe, P. L., "Approximate Riemann Solvers, Parameter Vectors and Difference Schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372.

[90] Michalak, K. and Ollivier-Gooch, C., "Limiters for Unstructured Higher-Order Accurate Solutions of the Euler Equations," *AIAA Forty-Sixth Aerospace Sciences Meeting*, 2008.

[91] Barth, T. J. and Jespersen, D. C., "The Design and Application of Upwind Schemes on Unstructured Meshes," *AIAA paper*, Vol. 89, 1989.

[92] Mavriplis, D. J., "Unstructured Mesh Discretisations and Solvers for Computational Aerodynamics," *AIAA Paper 2007-3955, 18th AIAA Computational Fluid Dynamics Conference, Miami, Florida*, 2007.

[93] Axelsson, O., *Iterative Solution Methods*, Cambridge University Press, 1994.

[94] Eisenstat, S., Elman, H., and Schultz, M., "Variational Iterative Methods for Nonsymmetric Systems of Linear Equations," *SIAM Journal of Numerical Analysis*, Vol. 20, 1983, pp. 345–357.

[95] Saad, Y., *Iterative Methods for Sparse Linear Systems. 2nd Edition*, Society for Industrial and Applied Mathematics, 2003.

[96] Jameson, A., "Time Dependent Calculations Using Multigrid, with Applications to Unsteady Flows Past Airfoils and Wings," *AIAA Paper 91-1596, AIAA 10TH Computational Fluid Dynamics Conference, Honolulu*, 1991.

[97] Badcock, K. J., Richards, B. E., and Woodgate, M. A., "Elements of Computational Fluid Dynamics on Block Structured Grids Using Implicit Solvers," *Progress in Aerospace Sciences*, Vol. 36, 2000, pp. 351–392.

[98] Barakos, G., Steijl, R., Badcock, K. J., and Brocklehurst, A., "Development of CFD Capability for Full Helicopter Engineering Analysis," *31st European Rotorcraft Forum*, Vol. Florence, 2005.

[99] Spentzos, A., Barakos, G., Badcock, K. J., Richards, B. E., Wernert, P., Schreck, S., and Raffel, M., "CFD Investigation of 2D and 3D Dynamic Stall," *AIAA Journal*, Vol. 35, 2005, pp. 10231033.

[100] Vallespin, D., Da Ronch, A., Boelens, O., and Badcock, K. J., "Vortical Flow Prediction Validation for an Unmanned Combat Air Vehicle Model," *Journal of Aircraft*, Vol. 48, 2011, pp. 1948–1959.

[101] Tritton, T. J., "Experiments on the Flow Past a Circular Cylinder at Low Reynolds Numbers," *Journal of Fluid Mechanics*, Vol. 6, 1959, pp. 547–567.

[102] Taneda, S., "Experimental Investigation of the Wakes behind Cylinders and Plates at Low Reynolds Numbers," *Journal of the Physical Society of Japan*, Vol. 11, 1956, pp. 302–307.

[103] Williamson, C. H. K. and Roshko, A., "Measurements of Base Pressure in the Wake of a Cylinder at Low Reynolds Numbers," *Zeitschrift Fuer Flugwissenschaften und Weltraumforschung*, Vol. 14, 1990, pp. 38–46.

[104] Subhankar, S., Sanjay, M., and Biswas, B., "Steady Separated Flow Past a Circular Cylinder at Low Reynolds Numbers," *Journal of Fluid Mechanics*, Vol. 620, 2009, pp. 89–119.

[105] Cook, P. H., McDonald, M. A., and Firmin, M. C. P., *Aerofoil RAE 2822 Pressure distributions, and Boundary Layer and Wake Measurements*, AGARD AR 138, 1979.

[106] Timme, S., *Transonic Aeroelastic Instability Searches Using a Hierarchy of Aerodynamic Models*, Ph.D. thesis, University of Liverpool, 2010.

[107] Karypis, G. and Kumar, V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM. Journal on Scientic Computing*, Vol. 20, 1999, pp. 359–392.

[108] Woodgate, M., Badcock, K. J., and Richards, B. E., "A Parallel 3D Fully Implicit Unsteady Multiblock CFD Code Implemented on a Beowulf Cluster," *Parallel CFD*, Vol. 20, 1999, pp. 359–392.

[109] Schmitt, V. and Charpin, F., "Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers," *AGARD, TR AR138*, 1979.

[110] Lain, K. R., Klausmeyer, S. M., Zickuhr, T., Vassberg, J. C., Wahls, R. A., Morrison, J. H., Brodersen, O., Rakowitz, E., Tinoco, E. N., and Godard, J. L., "Data Summary from Second AIAA Computational Fluid Dynamics Drag Prediction Workshop," *Journal of Aircraft*, Vol. 42, 2005, pp. 1165–1178.

[111] Vassberg, J., Tinoco, E., Mani, M., Rider, B., Zickuhr, T., Levy, D., Broderson, O., Eisfeld, B., Crippa, S., Wahls, R., Morrison, J., Mavriplis, D., and Murayama, M., "Summary of the Fourth AIAA CFD Drag Prediction Workshop,"

AIAA 2010-4547, 28th AIAA Applied Aerodynamics Conference, Chicago, IL, June 2010.

[112] Martin, M., Andres, E., Widhalm, M., Bitrian, P., and Lozano, C., "CAD-Based Aerodynamic Shape Design Optimization with the DLR Tau Code," *Paper ICAS 2010-2.6.1 27th Congress of International Council of the Aeronautical Sciences. Nice, France*, 2010.

[113] Kennett, D. J., Timme, S., Angulo, J. J., and Badcock, K. J., "Semi-Meshless Stencil Selection for Anisotropic Point Distributions," *International Journal of Computational Fluid Dynamics*, Vol. 26, 2012, pp. 463–487.

[114] Bonet, J. and Peraire, J., "An Alternating Digital Tree (ADT) Algorithm for 3D Geometric Searching and Intersection Problems," *International Journal of Numerical Methods in Engineering*, Vol. 31, 1991, pp. 1–17.

[115] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, The MIT Press, 2009, ISBN: 0262033844.

[116] J'aJ'a, J., *An Introduction to Parallel Algorithms*, Addison-Wesley Pub Co, 1992.

[117] Landon, R. H., "NACA 0012. Oscillatory and Transient Pitching," *Technical Report*, Vol. AGARDR702, 1982.

[118] Valarezo, W. O., McGhee, R. J., Goodman, W. L., and Paschal, K. B., "Multi-Element Airfoil Optimization for Maximum Lift at High Reynolds Numbers," *In Proceedings of the AIAA 9th Applied Aerodynamics Conference*, Vol. Washington, DC, 1991, pp. 969–976.

[119] Marques, S., Badcock, K. J., Khodaparast, H. H., and Mottershead, J. E., "Transonic Aeroelastic Stability Predictions Under the Inuence of Structural Variability," *Journal of Aircraft*, Vol. 47, 2010, pp. 1229–1239.

[120] Carman, J. B., Hill, D. W., and Christopher, J. P., "Store Separation Testing Techniques at the Arnold Engineering Development Center, Volume II: Description of Captive Trajectory Store Separation Testing in the Aerodynamic Wind Tunnel (4T)," *Arnold Engineering Development Centre-TR-79-1*, Vol. 2, 2000.

[121] Fox, J. H., "Generic Wing, Pylon, and Moving Finned Store," *Verification and Validation Data for Computational Unsteady Aerodynamics*, Vol. RTO Technical Report, 2000.

[122] Chin, V. D., Peter, D. W., Spaid, F. W., and McGhee, R. J., "Floweld Measurements About a Multi-Element Airfoil at High Reynolds Numbers," *AIAA paper 93-3137*, Vol. 31, July 1993.

Appendix A

Appendices

A.1 Decomposition by Polar Coordinates

One of the domain decomposition methods used in this work uses polar coordinates as a basis for two-dimensional problems and cylindrical coordinates for three-dimensional ones. In two dimensions, the original cartesian coordinates (X and Z) of the points are transformed. In three-dimensional problems, the user selects in which plane (XZ or YZ) to perform the polar transformation and the third axis is extruded from this plane. In any of the cases, the transformation from Cartesian to Polar coordinates is done using the following equations:

$$r = \sqrt{x'^2 + y'^2} \tag{A.1}$$

$$\varphi = \text{atan2}(y',\ x') \tag{A.2}$$

where x' and y' are the selected horizontal and vertical coordinates, r and φ are the radius and angle in the polar coordinates respectively and atan2 is a variation of the arc-tangent function defined as:

$$\text{atan2}(a,\ b) = \begin{cases} \arctan(\frac{b}{a}), & \text{if } a > 0 \\ \arctan(\frac{b}{a}) + \pi, & \text{if } a < 0 \text{ and } b \geq 0 \\ \arctan(\frac{b}{a}) - \pi, & \text{if } a < 0 \text{ and } b < 0 \\ \frac{\pi}{2}, & \text{if } a = 0 \text{ and } b > 0 \\ -\frac{\pi}{2}, & \text{if } a = 0 \text{ and } b < 0 \\ \text{undefined}, & \text{if } a = 0 \text{ and } b = 0 \end{cases} \tag{A.3}$$

A.2 6-DOF Motion Simulation

This appendix describes the implementation of the motion simulation of unconstrained rigid-bodies onto PML. A rigid body in space has six degrees of freedom (6-DOF), three translations and three rotations. Because of this, the kinematics of any body in space are fully described by the position of one point in the body (usually taken at the centre of mass) and the angular orientation (attitude) of the body in respect with an external reference system. The external reference system used in this work is an inertial (and hence, fixed) reference system, called ground. In order to define the angular motion of the body, a second reference system called body-fixed is introduced. This body-fixed system is the unique frame defined by the principal axes of the moments of inertia of the body. The ground and body-fixed systems can be seen in Fig. A.1.

The attitude of the body-fixed system relative to the ground is tracked by using quaternions, commonly known as Euler Parameters:

$$\mathbf{p} = [e_0 \;\; e_1 \;\; e_2 \;\; e_3]^T \tag{A.4}$$

In simple terms, the Euler parameters can be seen as describing the attitude of the object by defining one axis of rotation ($\mathbf{a}$) and the angle of rotation (ϕ) about that axis.

$$\begin{aligned} e_0 &= \cos\frac{\phi}{2} \\ e_1 &= a_x \sin\frac{\phi}{2} \\ e_2 &= a_y \sin\frac{\phi}{2} \\ e_3 &= a_z \sin\frac{\phi}{2} \end{aligned}$$

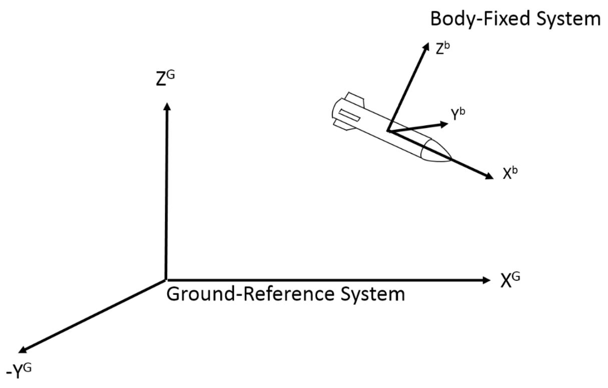


Figure A.1: Ground reference system, denoted with the superscript G and body-fixed system, denoted by the superscript b.

The vector-transformations between the two reference systems are performed with the help of the transformation matrix **A**, which is composed of the direction cosines. Using this transformation, a vector in the body-fixed system is transformed from the ground system by:

$$\overline{V^b} = \mathbf{A}^T \overline{V^G} \tag{A.5}$$

where **A** can be expressed in terms of the Euler parameters as:

$$\mathbf{A} = 2 \begin{bmatrix} {e_0}^2 + {e_1}^2 - \frac{1}{2} & e_1 e_2 - e_0 e_3 & e_1 e_3 + e_0 e_2 \\ e_1 e_2 + e_0 e_3 & {e_0}^2 + {e_2}^2 - \frac{1}{2} & e_2 e_3 - e_0 e_1 \\ e_1 e_3 - e_0 e_2 & e_2 e_3 + e_0 e_1 & {e_0}^2 + {e_3}^2 - \frac{1}{2} \end{bmatrix} \tag{A.6}$$

The 6-DOF motion is calculated by solving the Newton-Euler equations for rigid-body motion. The motion calculation is separated into two components, namely the translation of the centre of mass (CG) and the angular motion around this CG. The translation of the CG is governed by the Newton laws of motion which can be written in the ground frame as:

$$\mathbf{F}^G = m\ddot{\mathbf{r}}^G \tag{A.7}$$

where $\mathbf{F}^G$ is the total sum of forces, m is the mass of the body and $\ddot{\mathbf{r}}^G$ is the acceleration of the CG in the ground reference frame. The forces acting on the body are the sum of the aerodynamic forces, the external forces (eg: thrust, ejectors) and the gravitational force.

The angular motion is governed by the Euler equations of motion. These equations written in the body-fixed reference frame are:

$$\mathbf{M}^b = \mathbf{I}\dot{\boldsymbol{\omega}}^b + \boldsymbol{\omega}^b \times \mathbf{I}\boldsymbol{\omega}^b \tag{A.8}$$

where $\mathbf{M}^b$ is the sum of moments acting on the CG of the body, $\mathbf{I}$ is the inertia matrix, $\dot{\boldsymbol{\omega}}^b$ is the angular acceleration and $\boldsymbol{\omega}^b$ is the angular rate vector. Assuming a constant inertia matrix and that the inertia cross-terms (eg: I_{xy}, I_{yz}, etc) are zero, equation A.8 can be written as the system:

$$\begin{aligned} {M_x}^b &= {I_x}^b {\dot{\omega}_x}^b - ({I_y}^b - {I_z}^b){\omega_y}^b {\omega_z}^b \\ {M_y}^b &= {I_y}^b {\dot{\omega}_y}^b - ({I_z}^b - {I_x}^b){\omega_z}^b {\omega_x}^b \\ {M_z}^b &= {I_z}^b {\dot{\omega}_z}^b - ({I_x}^b - {I_y}^b){\omega_x}^b {\omega_y}^b \end{aligned} \tag{A.9}$$

Solving equations A.7 and A.8 results in the second derivatives of the CG position and the first derivatives of the angular rates, $\ddot{\mathbf{r}}^G$ and $\dot{\boldsymbol{\omega}}^b$ respectively. Euler's theory of motion defines that the Euler parameters are the same in both ground and body-fixed reference systems. Using this, we can find the first and second derivatives of the Euler parameters from the body-fixed angular rates and accelerations as follows:

$$\begin{aligned} \dot{\mathbf{p}} &= \tfrac{1}{2}\mathbf{L}^T \boldsymbol{\omega}^b \\ \ddot{\mathbf{p}} &= \tfrac{1}{2}\mathbf{L}^T \dot{\boldsymbol{\omega}}^b - \tfrac{1}{4}(\boldsymbol{\omega}^b)^2 \mathbf{p} \end{aligned} \tag{A.10}$$

where:

$$\mathbf{L}^T = \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \tag{A.11}$$

After the derivatives are found, a system of second-order ordinary differential equations of the form $\frac{d^2x}{dt^2} = f(t, x, \dot{x})$ is obtained. These equations are solved in time using a modified Euler scheme:

$$\begin{aligned} x_{n+1} &= x_n + \dot{x}_n \Delta t + \tfrac{1}{2}\ddot{x}_n \Delta t^2 \\ \dot{x}_{n+1} &= \dot{x}_n + \ddot{x}_n \Delta t \end{aligned} \tag{A.12}$$

where n and $n+1$ denote the current and next time-steps and Δt is the size of the step.

Finally, the Euler parameters can be transformed into the roll (ϕ), pitch (θ) and yaw (ψ) angles described by the rigid body with:

$$\begin{aligned} \tan(\phi) &= \tfrac{2.0(e_0e_1 + e_2e_3)}{(e_0{}^2 - e_1{}^2 - e_2{}^2 + e_3{}^2)} \\ \sin(\theta) &= -2.0(e_1e_3 - e_0e_2) \\ \tan(\psi) &= \tfrac{2.0(e_1e_2 + e_0e_3)}{(e_0{}^2 + e_1{}^2 - e_2{}^2 - e_3{}^2)} \end{aligned} \tag{A.13}$$