UPC **CTTC**

Parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Centre Tecnològic de Transferència de Calor
Departament de Màquines i Motors Tèrmics
Universitat Politècnica de Catalunya

Ricard Borrell Pol

Doctoral Thesis

Parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Ricard Borrell Pol

TESI DOCTORAL

presentada al

Departament de Màquines i Motors Tèrmics
E.T.S.E.I.A.T.
Universitat Politècnica de Catalunya (UPC)

per a l'obtenció del grau de

Doctor per la UPC

Terrassa, September 2012

Parallel algorithms for Computational Fluid Dynamics on unstructured meshes

Ricard Borrell Pol

Directors de la Tesi

Dr. Assensi Oliva Llena

Dr. Guillem Colomer Rey

Dra. Ivette Rodríguez Pérez

Tribunal Qualificador

Dr. Antonio Pascau Benito
Universidad de Zaragoza

Dr. Joaquim Rigola Serrano
Universitat Politècnica de Catalunya

Dr. Roel Verstappen
University of Groningen

A la Romina i la Mila

Agraïments

Crec fermament que les coses que aconseguim en aquesta vida tenen molt a veure amb les persones que ens envolten i la sinèrgia que ens hi uneix. En aquest sentit, durant aquesta tesi he tingut molta sort. És per això que, en primera instància, vull agrair a tots els companys i amics del CTTC la seva col·laboració i els bons moments que hem viscut junts. En especial, vull mostrar el meu agraïment...

A l'Assensi, el cap del nostre departament, per la seva amistat i confiança, així com l'entusiasme que ens transmet cada dia per seguir progressant. També li vull agrair que m'hagi permès enfocar la meva tasca d'investigació en els temes que més m'han interessat.

A l'Oriol Lehmkuhl, amb qui he treballat molt intensament durant tots aquests anys de tesi. Especialment en el desenvolupament del codi TermoFluids. El seu empeny i la seva capacitat, juntament amb la Ivette Rodríguez, per abordar la simulació numèrica de fluxos cada cop més complexes, ha estat un estímul fonamental per seguir progressant en la computació paral·lela.

A en Xavi Trias i l'Andrey Gorobets, per ajudar-me en incomptables ocasions, per totes les converses enriquidores i estimulants que hem tingut. El segon capítol d'aquesta tesi té com a *background* el seu treball, juntament amb en Manel Soria, en el desenvolupament d'algoritmes paral·lels per codis DNS en malles estructurades.

A en Guillem Colomer, amb qui hem treballat conjuntament en el desenvolupament dels algoritmes paral·lels per resoldre l'equació de transport de Boltzmann exposats en el tercer capítol d'aquesta tesi. Ha estat un gran plaer treballar amb ell i conèixer els seus punts de vista.

Volia mostrar el meu agraïment també a l'equip d'informàtics del laboratori, en Ramiro, l'Octavi i anteriorment en Daniel Fernàndez, per la paciència que han tingut en ajudar-me infinites vegades. Per sort, en els darrers temps m'he dedicat més a escriure i no els he molestat tant sovint.

Als meus companys de sala, l'Angel i en Carles pels bons moments que hem viscut junts. A en Guillermo, en Lluís i l'Olga, amb qui he treballat en els darrers temps. Als meus companys matemàtics amb qui vaig començar aquesta aventura, en Ferran i la Gemma.

Finalment, al professor Roel Verstappen, per acollir-me en l'estada que vaig realitzar al departament de Matemàtiques i Ciències Computacionals de la Universitat de Groningen.

En el terreny més personal, volia mostrar el meu agraïment als meus amics, pel seu suport, per fer veure que s'ho creien cada cop que, en els últims anys, els deia que que en pocs mesos tindria la tesi enllestida.

Molt especialment als meus avis, per tots els anys que vam conviure a Barcelona mentre estudiava la carrera.

Als meus germans i especialment als meus pares, per la seva estima i el seu suport incondicionals durant tota la meva vida.

Finalment, al meva estimada dona, la Michelle, i les meves filles, la Romina i la Mila, que em fan feliç cada dia.

A tots vosaltres, amb molta estima, gràcies. Ja la tenim aquí! us la dedico.

Abstract

Direct Numerical Simulation (DNS) of complex flows is currently an utopia for most of industrial applications because computational requirements are too high. For a given flow, the gap between the required and the available computing resources is covered by modeling/simplifying of some terms of the original equations, in order to make the problem easier to be numerically solved. For example, turbulence models attempt to mimic the effects of the unresolved scales of motion in the flow or, in the simulation of flames, the effects of the radiation are usually introduced using empirical correlations with the temperature. These are essential strategies to make possible the numerical simulation of most of industrial flows. On the other hand, the continuous growth of the computing power of modern supercomputers contributes to reduce the gap between the required and the available computational resources for DNS, reducing hence the unresolved physics that need to be attempted with approximate models. This growth, consisting on doubling the supercomputing capabilities roughly every year, widely relies on parallel computing technologies. However, getting the expected performance from new complex computing systems is becoming more and more difficult, and therefore part of the CFD research is focused on this goal. Regarding to it, some contributions are presented in this thesis[1].

The first objective was to contribute to the development of a general purpose multi-physics finite-volume CFD code. This code, referred to as TermoFluids (TF), is programmed following the object oriented paradigm and designed to run in modern parallel computing systems. Among its functionalities there is the possibility to solve multi-physics problems with high-level methods including radiation effects, reactive flows, multiphase flows, fluid-structure interactions, problems with dynamic meshes or multi-scale systems. The code is intensively involved in many different projects ranging from basic research to industry applications. Besides the capability to properly simulate all these complex physical phenomena, one of the strengths of TF is its good parallel performance demonstrated in several supercomputers, and explicitly tested with up to 8192 CPUs.

In the context of this thesis, the work was focused on the development of two of the most basic libraries that compose TF. The first one is the Basic Objects Library (BOL), which is a parallel unstructured CFD application programming interface (API), on the top of which the rest of libraries that compose TF are written. The BOL frees its users of the important software development effort required to support the basic discrete operations necessary to perform parallel unstructured CFD sim-

[1]This work comprises three main chapters, the second and the third chapters are based on the contents of two papers published in international journals. Hence, they are written to be self-contained and only minor changes have been introduced with respect to the originals. A complete list of the publications carried out within the framework of this thesis is presented in Appendix C.

ulations. Afterwards, the Linear Solvers Library (LSL) was developed containing many different algorithms to solve the linear systems arising from the discretization of the equations. The solution of these systems is one of the most time-consuming parts of CFD simulations of incompressible flows and, therefore, it has an important influence on the overall performance.

The first chapter of this thesis contains the main ideas underlying the design and the implementation of the BOL and LSL libraries, together with some examples and the description of some industrial applications. A detailed description of some application-specific linear solvers included in the LSL is carried out in the following chapters.

In the second chapter, a parallel direct Poisson solver restricted to problems with one uniform periodic direction is presented. It can be applied to configurations such as the flow around an airfoil or around a sphere, in which the uniformity and periodicity imposed by the method fit with the isotropic nature of the flow in the span-wise and azimuthal direction, respectively. The Poisson equation needs to be solved, at least, once per time-step when modeling incompressible flows, to project the velocity field onto a divergence-free space. Due to the non-local nature of its solution, this elliptic system is one of the most time consuming and difficult to parallelize parts of the code. The solver here proposed is a combination of a direct Schur-complement based decomposition (DSD) and a Fourier diagonalization. The latter decomposes the original system into a set of mutually independent 2D subsystems which are solved by means of the DSD algorithm. Since no restrictions are imposed in the non-periodic directions, the overall algorithm is well-suited for solving problems discretized on extruded 2D unstructured meshes. The load balancing between parallel processes and the parallelization strategy are also presented and discussed in this chapter. The scalability of the solver is successfully tested using up to 8192 CPU cores for meshes with up to 10^9 grid points. Finally, the performance of the DSD algorithm as 2D solver is analyzed by direct comparison with two preconditioned conjugate gradient methods. For this purpose, the turbulent flow around a circular cylinder at Reynolds numbers 3900 and $10,000$ are used as problem models.

In the last chapter, a solver for the Boltzmann Transport Equation (BTE) is presented. It can be used to solve radiation phenomena interacting with flows. The solver is based on the Discrete Ordinates Method and can be applied to unstructured discretizations. In the solution process, the flux for each angular ordinate is swept across the discretization mesh, within a source iteration loop that accounts for the coupling between the different ordinates. A spatial domain decomposition strategy is used to divide the work among the available CPUs. The sequential nature of the sweep process makes the parallelization of the overall algorithm the most challenging aspect. Several parallel sweep algorithms, which represent different options of interleaving communications and calculations in the solution process, are analyzed. The option of grouping messages by means of buffering is also considered. One of the heuristics proposed consistently stands out as the best option in all the situations analyzed, which include different geometries and different sizes of the ordinate set. With this algorithm, good scalability results have been achieved regarding both weak and strong speedup tests with up to 2560 CPUs.

Contents

List of Figures

List of Tables

1

Design and implementation of a CFD code

Abstract. One of the main objectives of this thesis has been to contribute to the development of a general purpose multi-physics CFD code. This code, referred to as TermoFluids, has been programmed following the object oriented paradigm and designed to run in modern supercomputers. Its functionalities and performance are in constant evolution due to the increasing number of developers and users. In the context of this thesis, two of the most basic libraries that compose the code were developed. These are the Basic Objects Library, which is an unstructured CFD application programming interface, on the top of which the rest of libraries that compose TermoFluids are written; and the Linear Solvers Library, which contains several algorithms to solve the linear systems of equations, derived from the discretization of the equations that model the physics under study. In this chapter, the main ideas underlying the design and the implementation of these two libraries are presented, together with some examples and the description of some industrial applications.

An important part of this research has been developed in Termo Fluids S.L. company, which is a spin-off of the Centre Tecnològic de Transferència de Calor (CTTC) of the Universitat Politècnica de Catalunya.

1.1 Introduction

Fluids are ubiquitous (air, water, blood, lava...), therefore the knowledge, understanding and predictability of fluid flows is of great importance for our development as a species. The Computational Fluid Dynamics (CFD) field is a scientific branch focused on the numerical simulation of fluid flows. It includes, among other aspects, the development of numerical methods and algorithms, the design and verification of turbulence models, the physical analysis and understanding of the simulation results, and also more practical issues such as the development of software and the knowledge of the hardware to be used for the computations. It is therefore a multidisciplinary knowledge which involves scientists from different areas, such as physics, mathematics, engineering or computer science. Many industries and areas of research can benefit from the advances in CFD. For example, to name a few, the marine and off-shore engineering (resistance of ships or wave impact on vessels), the automotive and aerospace industry (combustion in engines, climate control in the passenger compartment, aerodynamics) and also biomedical applications (blood flow through arteries, respiratory flow in lungs).

The basic model to predict a single-phase fluid flow are the Navier-Stokes equations, that describe the conservation of momentum, together with the continuity equation, which expresses the conservation of mass. Other physical processes like radiation transport, chemical reactions (for example the ones that occur in combustion phenomena), thermal processes, interaction between different fluids and interaction with solids, or dynamic geometries, can also be included in the simulations by means of simplified source terms or the addition of more differential equations to the system. Thus, the flow simulation becomes a multi-physics problem.

Apart from having proper formulated discrete equations (in some phenomena, for instance the simulation of blood flow inside the human circulatory system, this is one of the most challenging aspects), a major difficulty in order to generalize the use of CFD to "real life" applications are its computational requirements. If we consider the Direct Numerical Simulation (DNS) of basic isotropic turbulent flows, the computational complexity of the simulation scales with $Re^{11/4}$ [1], where the Reynolds number, Re, is a dimensionless variable which measures the ratio between the effects of convection and diffusion. This scaling is even worse for general 3D cases with turbulent boundary layers. For a typical aerodynamic application (airflow around a flying plane airfoil for instance) the Re of the flow is around 10^7. If we consider the existing academic studies of turbulent boundary layers in channel flows, one of the largest DNS simulation ever made, with $Re \sim 10^5$, was carried out by Hoyas et al. [2] in 2006. During an approximate period of half a year they used 2048 CPU, and about 25 TB of data were generated. This means that, using the same software and hardware, we would require at least 10^5 years to carry out a DNS for a real aerodynamic case ($Re \sim 10^7$) in such a simplified parallelepiped domain. As a consequence, for the simulation of most of turbulent industrial flows there is no choice but to work with approximated equations, in which the influence of the smallest scales of motion is modeled in order to decrease the computational requirements. In any case, these approaches may be quite good for many applications. In general, the two main families of turbulence modeling approaches are the Reynolds Averaged Navier-Stokes (RANS) simulations, in which a long-time temporal filter is applied resulting in a steady mean flow, and the Large Eddy Simulation (LES), in which localized spacial filters are applied. In terms of computational effort the LES lies in between the RANS and the DNS.

Together with the development of better turbulence models and numerical methods, one of the basic strategies to expand the capabilities of the CFD is to take advantage of the constant growth of computing power. This growth implies sometimes qualitative changes in the technology, like the appearance of the graphical processing units (GPU) computing in the recent years. In fact, numerical methods are designed to be applied into computing systems, so their efficiency on the state of the art or future computing equipments is of major importance. In accordance with Moore's law, the speed of the new supercomputers has been doubling every 13.2 months during the last 20 years [3]. This means that, if this trend continues, just by adapting the software, each decade we could approximately increase an order of magnitude the Re of the isotropic turbulent flows that can be directly solved. However, the efficient usage of supercomputers is far from being trivial and the maximum possible performance is rarely achieved. Therefore, part of the research on the numerical simulation field has to focus on the development of parallel software to take advantage of the constantly growing computing power.

In the context of this thesis, we have worked on the development of a general purpose multi-physics CFD software named TermoFluids (TF). This is an object oriented software programed in C++ and designed to run in parallel computing systems. TF is composed of several libraries arranged in a hierarchical scheme from the most fundamental and general to the most specific ones (which deal for instance with only a particular physical phenomenon). General unstructured meshes are used for the geometric discretizations, and the basic equations are discretized by means of a "symmetry-preserving"/"energy-conserving" approach [4, 5]. There are also a number of LES [6, 7] and regularization turbulence models [8, 9], and a library with general and application-specific linear solvers [10, 11]. In the last years, TF has been evolving into a multi-physics software incorporating, for example, radiation effects [11,12], reactive flows [13], multiphase flows [14–16], fluid-structure interactions [17], dynamic mesh methods [18] and multi-scale systems [19]. Besides the capability to properly simulate all these complex physical phenomena, one of the strengths of the code is its good parallel performance, demonstrated in several supercomputers (see Appendix B), and explicitly tested with up to 8192 CPUs [10]. TF is actually being used in both industrial and academic applications, ranging between DNS studies of the flow around a sphere [20, 21], in order to understand the structures of turbulence in forced convective flows, and LES simulations of the flow inside the power generator nacelle of a wind turbine [22] or inside a domestic refrigerator [23].

TF has an increasing number of users and developers with different needs and interests. The development of the code is a result of the interaction of between all of them. In the context of this thesis initially the Basic Objects Library (BOL) was developed, which is a parallel unstructured finite-volume CFD application programming interface (API), on the top of which the rest of libraries of TF are written. Afterwards, was developed the Linear Solvers Library (LSL), which has several general purpose and application-specific algorithms to deal with the solution of the linear systems derived from the discretizations.

The rest of the chapter is organized as follows. In Section 1.2 there is a general explanation of the parts that constitute a CFD simulation. Sections 1.3 and 1.4 describe the concepts in which the design and implementation of the BOL and the LSL are based, respectively. Section 1.5 presents, as a demonstrative example of an implementation using the BOL and the LSL libraries, a code for the resolution of the Poisson equation. Finally, some industrial applications are outlined in Section 1.6.

1.2 Anatomy of a CFD simulation

A typical CFD simulation can be divided into three main phases: i) the set-up or pre-processing, ii) the time-integration, and iii) the post-processing. These phases may be done consecutively, but sometimes it is necessary to repeat a previous phase in order to modify some aspect of the simulation. For example, during the post-processing, we may realize that the integration time is not long enough or that the quality of the mesh is poor to capture all the physics. This would force us to return to the time-integration or pre-processing phase, respectively. One key aspect is that the simulation can be restarted at any specific point minimizing the computations that must be repeated. This is achieved by storing intermediate results in binary files.

Usually, the most expensive part in terms of computational time is the iterative process necessary to carry out the time integration. The time step must be small enough to capture all the details of the physics and to keep the simulation stable, and this results into large numbers of time iterations to complete the simulations. Some simulations require several months of computations, while others require only a few minutes. It depends on the complexity of the physics that need to be captured with them.

On the pre-processing phase, a mesh covering the physical domain must be defined, and after that several calculations are carried out before starting the iterative process. Among these, the most important are the calculation of geometric properties, the evaluation of the topological relations between different elements of the discretization, the set-up of the iterators, the evaluation of the communication schemes for the parallelization and the set-up of linear solvers. All these processes are necessary before starting a simulation but, since the results are stored in files, they are only carried out once independently of the number of executions necessary to complete the simulation. The higher the number of iterations required by the time-integration process, the less significant is the time spent in the set-up process in relative terms.

Finally, the cost of post-processing the data generated during the time integration depends on the analysis required. Generating a movie with instantaneous fields, in order to observe the dynamic structure of the fluid flow, may be rather expensive, and in some cases demands parallel computation, while the evaluation of some statistical flow features is mainly carried out on run time during the time integration. Nevertheless, while the computational resources for the simulations grow, the storage and management of the data outputs are increasingly challenging problems for the CFD community [24].

In the design of the code, it is important to identify the classes and methods that form the most intensive part of the computations, to be considered when optimizing the code. In our case these are the methods used in the time-integration process. By contrast, the methods that are seldom used, are not a priority for the optimization unless they represent an important cost respect to the total simulation time.

1.3 Basic objects library

The basic objects library (BOL) is an in-house parallel API used by the rest of libraries that compose TermoFluids code. It has been designed to be of general purpose and to have a good parallel performance in the different existing parallel computational infrastructures. Following the Object Oriented (OO) paradigm, the library is mainly composed by classes, representing types of data, and member functions, which define the core operations to be performed with each data type. The BOL library is divided into four main interrelated areas:

- The *geometric classes* which deal with the geometric elements and concepts of the discretization: nodes, polygons, polyhedra, faces, cells, volumes, areas, intersections, projections, distances, etc.
- The *algebraic classes* representing the algebraic structures and operations: vectors, sparse matrices, norms, dot products, cross products, sparse-matrix vector products, etc.
- The *parallelism classes* which represent the issues related with the parallelization such as the domain decomposition, the communication schemes, the global

and local labeling of elements, and the communication buffers.

- The *utilities,* to basically handle input/output (IO) and profiling operations.

This section describes the main ideas underlying the design of the first three areas. The utilities are a more technical part of the code that will not be addressed in this general explanation.

1.3.1 Geometric classes

Prior to carry out a simulation using the BOL, the mesh where the equations will be discretized must be defined. Apart from some basic cases, like structured meshes or meshes generated by extrusion or revolution of a given 2D mesh, it is not possible to generate geometric discretizations using the BOL. Thus, mesh generators, such as the ANSYS ICEM CFD [25] or GMSH [26] packages, are required.

Figure 1.1 shows different type of meshes that the BOL can deal with. In fact, any type of mesh within the BOL is treated as a general unstructured 3D one. Particular cases with structured patterns are not treated differently. On the other hand, 2D cases are solved by means of 3D meshes with prismatic elements, using Neumann boundary conditions at the front and back planes (see Figure 1.1.d).

The data necessary to define an unstructured mesh consists of: i) The coordinates of the vertices, ii) The ordered sequence of IDs of vertices that defines each face, and iii) The IDs of the faces that define each cell. Note that the ID of an element refers to its relative position on the input file. Some other details like the boundary conditions and the distinction between different zones in the domain can also be defined in the mesh input file.

Each basic element of the geometric discretization (vertex, face, cell or node) has its corresponding class in the library. Moreover, they are all grouped as private members of the class `Mesh`[1] which has a `Container<T>` for each element type, *i.e.* it has a `Container<Vertex>`, `Container<Node>`, `Container<Face>` and a `Container<Cell>`. The `Container<T>` template is derived from the standard class `vector<T>`, defined in the C++ Standard Template Library (STL).

To get an element from a `Mesh` object there are two options: i) If the position of the element in its container, referred to as its `lid` (local identification), is known, then it can be accessed directly with the member function `Container<T>::getL(lid)`, ii) Using a `Container<T>::iterator`: iterating through all the elements of the container the required element is also accessed.

Most of the operations carried out with the BOL are generic, designed to be applied to groups of elements of the same type. For this reason, using iterators is a natural way of programming with it. For example, in order to know the volume of the domain covered by the mesh, it is necessary to iterate through all the cells

[1]The common typography of code editors is used to write the name of the elements of the BOL.

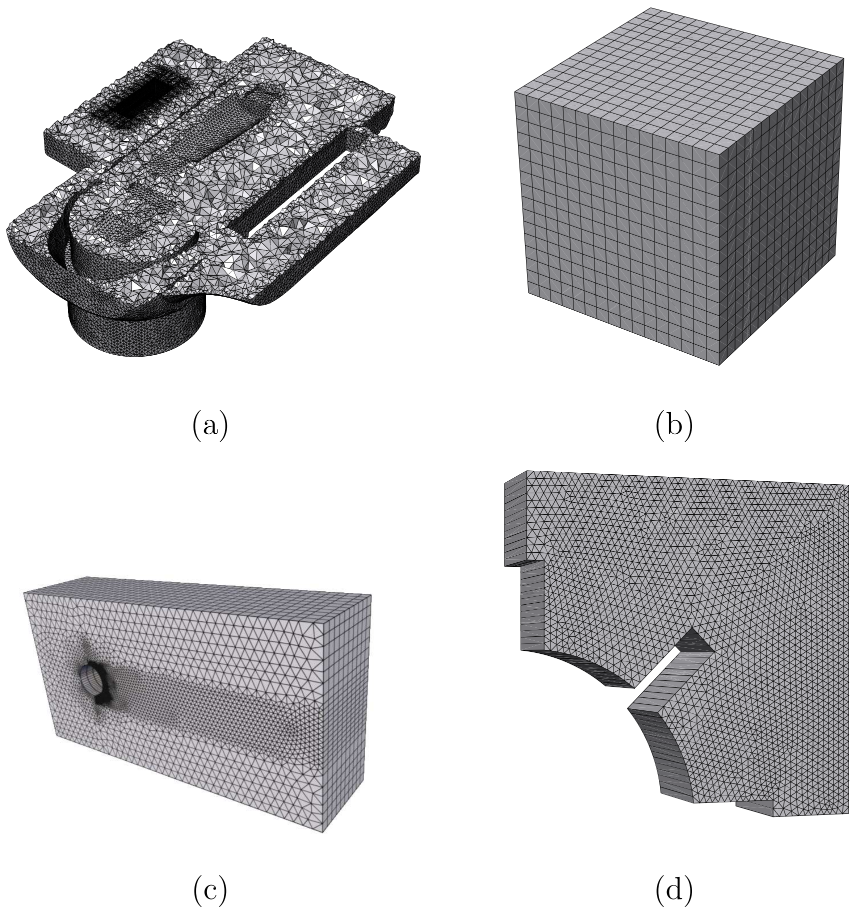


Figure 1.1: Different type of geometric discretizations solved with the BOL. a) General three dimensional mesh composed of polyhedrons, b) Structured mesh, c) Mesh obtained by extrusion of a 2D unstructured grid, d) Mesh for a 2D case.

and add the volume of each one to a global summation variable. The `lid` of each specific cell is not needed, because cells are treated as a group applying the same process to each of its elements. Sometimes we need to iterate through a subset of the elements of a container. For example, we may need to iterate through the boundary faces, through a specific set of nodes forming a stencil, etc. These partial iterators are defined by means of a `vector<int>` containing the specific `lid` of the elements that must be swept. Thus, a `PartialIterator` iterates through the previously defined vector of `lid`s and accesses to the corresponding elements of the container. The class `Mesh` has several partial iterators defined on the pre-processing member function `Mesh::close()`. An example of its usage is:

```
double surface=0;
```

```
Mesh::boundary_face_iterator bfend=mesh.bfEnd();
for(Mesh::boundary_face_iterator i=mesh.bfBegin(); i!=bfend; ++i){
    surface+=i->getSurface();
}
```

In this case, the iterator through the boundary faces of the mesh is used to find the surface of the domain covered by it. Other iterators defined in the class `Mesh` are:

```
Mesh::face_iterator
Mesh::vertex_iterator
Mesh::boundary_cell_iterator
Mesh::inner_node_iterator
Mesh::const_inner_vertex_iterator
...
```

Each iterator `typedef` is composed of words separated by the underscore character "_" . At the last place the word "`iterator`" is always written. This is preceded by the type of item for which it iterates, "`vertex`", "`node`", "`face`" or "`cell`". Previously the words "`boundary`" or "`inner`" can be used to determine if it is restricted to the elements located in the boundary of the domain, or the complementary of the boundary, respectively. And finally the name of the iterator is preceded by the word "`const`" when the elements accessed by the iterator are not modified. Having an exhaustive set of iterators helps a lot in the programing of the routines of a CFD code. Most of discretization operations are carried out by means of iterators through subsets of equivalent elements, therefore, the iterators must be well optimized.

Hereafter the main characteristics of the classes that represent the basic element of the mesh are described:

- `Vertex`. The class `Vertex` is derived from the generic class `D3`, which is a vector with three coordinates[2]. The `Vertex` in the `Mesh` requires the definition of some iterators in order to access the elements of its environment. These iterators can not be defined independently by any `Vertex` object, because the information of the whole set of mesh elements related with it is necessary. Therefore, the set-up of these partial iterators is carried out in the function `Mesh::close()`. Finally, the member function `Vertex::close()` is just called to check that all the data is consistent. Some examples of vertex iterators are:

  ```
  Vertex::neighbor_vertex_iterator
  Vertex::const_neighbor_node_iterator
  Vertex::containing_face_iterator
  Vertex::const_containing_cell_iterator
  ```

[2]Class `D3` has some geometric methods such as the distance to another `D3` object, or the distance to a `Plane` or `Line` object, respectively. The generic classes `D3`, `Line` and `Plane`, do not represent basic elements of the mesh but are used in numerous geometric routines of TF code.

...

- `Node`. The class `Node` is also derived from the class `D3`, however each `Node` is univocally associated to a boundary face or a cell, and generally located in its centroid. The `lid` of the associated face or cell is stored as a private member of the class and can be obtained by means of the member function `Node::getFaceOrCellLid()`. The `Node` has some member functions to carry out basic operations and iterators to access to the other mesh elements related with it.

- `Face`. A face is a polygon and can be defined as an ordered set of integers corresponding to the `lid` of its vertices. On the preprocessing phase (`Face::close()`) some geometric information is calculated to be used on the discretization: the face area, the centroid, a unitary normal vector, the distances between its centroid and each of its vertices, etc. All this information can then be obtained by calling the corresponding member functions (`Face::getSurface(), Face::getCentroid()`, etc). Some checking is also carried out such as the coherence of the iterators, the convexity of the polygon, and coplanarity of the vertices that form it.

- `Cell`. A `Cell` is a polyhedron and is defined with the `lid`s of the `Face` objects that form it. In this case no ordering is needed: a set of faces determines only one possible polyhedron. The `Cell` has also a set-up method (`Cell::close()`) in which all the geometric information required in the discretization operations is evaluated. In particular to evaluate the volume and the centroid we use the algorithms described in [27]. The coherence of all the data including the iterators is also cheeked.

To recap, a `Mesh` object includes all the elements that define a geometric discretization arranged in different containers. The basic elements by themselves are simple but more complex are their interactions. The class `Mesh` and the classes representing the basic elements (`Vertex`, `Node`, `Face`, and `Cell`), gather all this information and a natural way, using its member functions and a comprehensive set of iterators, to obtain both the properties of a particular element and the relationship between them.

1.3.2 Algebraic classes

The algebraic elements used in the BOL are the vector and the matrix. The vectors are introduced in the code by means of the template `Container<T>` parameterized in this case with the type `double` or `D3`. Some specific member functions, such as the maximum and Euclidean norms, are defined, and some basic operators `(+, -, *, +=, *= ...)` are overloaded. When possible, for these algebraic operations the BLAS [28] standard routines are implicitly used. The code needs therefore to be linked with some BLAS implementation installed in the system in which it is executed.

In many cases, the components of a vector are associated to elements of the mesh, becoming then scalar or vectorial fields. In these cases there is a correspondence between the container of geometric elements and the container used to store the components of the vector. This way, the elements of the vector can be accessed by means of mesh iterators, using the function `Iterator::lid()`, which returns the `lid` of the element pointed by the iterator, to access the corresponding element of the vector. In the discretization of the Navier-Stokes equations, for example, the pressure and the velocity are an scalar and a vectorial field, respectively, defined in the nodes of the mesh.

On the other hand, the class `SparseMatrix` is derived from a `Container<SparseRow>` while, at the same time, the `SparseRow` is derived from the STL class `map<int,double>`[3]. Therefore, an `SparseMatrix` is essentially a `vector<map<int,double>>`. This format is very flexible and allows to introduce the matrix elements without following any particular order, or worry about issues related to the dynamic memory. However, this flexibility comes at a price in terms of memory consumption and computational inefficiency. For this reason, it is only used to manipulate the matrix and, before starting the algebraic operations, the entries are moved to a compressed storage format, reducing the memory requirements and improving the cache performance of the operations.

The compressed storage format is generally sparse. However, in some specific cases, when the percentage of entries is high, the option of storing the matrix in a dense pointer is also considered. This increases the memory requirements but allows to use the optimized dense algebraic routines of the BLAS libraries, which are much faster than the sparse routines. On the other hand, sparse storage schemes allocate contiguous positions in memory only for the nonzero elements of the matrix. Thus, they require a scheme to know where the elements fit into the full matrix. Despite there are many sparse matrix formats [29], in the BOL only the standard Compressed Sparse Row (CSR) format is used, because it fits with the parallelization strategy explained in next subsection, and it makes absolutely no assumptions about the sparsity structure of the matrix.

The basic routines for algebraic operations with matrices and vectors are defined by means of member functions or overloaded operators. Both alternatives are maintained but, despite the operators are more elegant and make the code more user-friendly, they become less efficient for some operations. This is illustrated in the next example:

[3]Maps are sorted associative containers that store elements formed by the combination of a key value and a mapped value.

```
b=A*x;
A.prod(x,b);
```

In both cases the result of `A*x` is stored in vector `b`. However, when the member function is used (second line), a pointer to `b` is given, as function argument, in order to store in it the result of the product. On the contrary, using the operator (first line), an intermediate object is automatically created to store the result of the product, and then is copied to `b`. The creation of these intermediate objects produce a degradation of the performance.

Linear solvers are very important in CFD codes because deal with one of the most computationally intensive parts of the simulation. They could also be considered as algebraic objects but in TermoFluids they are all grouped in a separated library explained in Section 1.4.

1.3.3 Parallelization

The BOL has been designed as a parallel library from the beginning. Running the code in parallel is of major importance in order to divide the computational and memory requirements. Up to now, in the explanation of the geometric and algebraic classes, the parallelization issues have been omitted for clarity, however, the library is fully designed to produce parallel objects and methods. In fact, the parallelization is mostly generated from the distribution of the containers, using the class `DistributedContainter<T>` instead of its base class `Container<T>`. This way, the mesh elements, the components of the vectors and the rows of the sparse matrices become automatically distributed. Sequential runs of the code are also possible but, nowadays, when even desktop computers have multi-core architectures, they are almost not used in practice. The main items of the parallelization are explained in the following paragraphs.

Distributed memory model. Current supercomputers are composed of many networked individual nodes, being each node itself essentially an independent computer, generally with a multi-core processor. Moreover, in the last years the idea of using general purpose Graphical Processing Units (GPU) as co-processors is becoming increasingly popular, and several supercomputers are composed of heterogeneous nodes which also incorporate these stream processing units. For example, according to the statistics provided by the TOP500 foundation (http://www.top500.org/), at this moment 3 of the top 10 general purpose supercomputers, and 12% of the top 500, appear as hybrid supercomputers incorporating GPU accelerators.

To make processors located in different nodes working together, the distributed memory paradigm, based in message-passing, becomes indispensable. The BOL is implemented using the Message Passing Interface (MPI) standard, which makes the code portable across all distributed memory systems. The distributed model can be combined with the shared memory multi-threading model to engage the cores inside each node. However, except for some specific cases, this option hasn't major advantages because, generally, the overhead due to transport of data between local caches, produces a similar or even greater degradation than the communications between cores of the same node [30]. As a consequence, the BOL has been implemented following a pure distributed model by means of the MPI standard. Only in some specific routines the threading option is used. Finally the possibility to use GPUs as co-processors is now being studied to accelerate the solution of the linear systems derived from the discretization [31].

Domain decomposition. The parallelization of the BOL is mainly originated from a mesh partition. It can be roughly stated that the computational cost of the numerical solution of a fluid in each cell is proportional to its number of faces. Following this idea, the mesh decomposition is derived from a partition of the graph of cells in which each cell has a weight equal to its number of faces. In many cases, in fact, it is not necessary to use a weighted graph because all the cells have similar number of faces. The graph partitioning is carried out by means of an external tool such as the METIS library [32]. Apart than providing a good load balance, METIS routines minimize the edge cuts, reducing the data exchange requirements in the simulation. If necessary, multilevel mesh partitions can also be generated by calling the graph partition routines recursively. Imbalanced partitions, to be used on heterogeneous clusters with nodes of different computational power, are also possible.

After the mesh partition, each parallel process deals with a subset of cells, nodes, faces and vertices that all together form a subdomain of the mesh. These are referred to as the *owned* elements of each type. Likewise, the partition of a field is derived from the partition of the set of elements on which it is defined.

Halo elements. In the geometric and algebraic parallel operations, each parallel process may need elements owned by others. For example, in the evaluation of the gradient of a scalar field defined in the nodes of a mesh, for each node the position and the field values of its neighbors are required. Thus if the node is on the boundary of a subdomain and has neighbors belonging to other subdomains, the parallel process associated to its subdomain needs information from other parallel processes to evaluate the gradient. To solve this problem, in the `DistributedContainer<T>` objects, for each parallel process, apart from the owned elements, a copy of the required elements owned by other processors is attached. Following with the gradient example, the copies of the neighbor nodes contained in other subdomains and the components of the scalar field associated to them, would be attached to the respective distributed containers. For each parallel process storing part of a `DistributedContainer<T>`, the copies of external elements attached to it are referred to as its *halo*. The number of halo elements attached to the distributed containers depends on the requirements imposed by the algorithms. To this regard, when the order of the numerical schemes used in a parallel discretization is increased, the halos of the involved geometrical and algebraic containers increase as well.

Its important to remark that any element of a halo is a copy. This means that the original element is owned by another parallel process. Thus, if the original element changes in the owner parallel process, the copy stored in the halo must be updated before using it. Otherwise, the results of the sequential and parallel executions would differ. Therefore, if the elements of a distributed container vary during a simulation halo updates may be required. In contrast, if the distributed container elements do not vary, communications are not necessary although elements are distributed. For example, if the mesh does not change, all the halos of all the geometric containers do not require to be updated.

The distinction between owned and halo elements of a distributed container requires the definition of the corresponding iterators:

```
DistributedContainer<T>::owned_iterator
DistributedContainer<T>::halo_iterator
```

And from the combination of these iterators with the ones defined in the previous subsections, we have iterators like:

```
Mesh::const_owned_node_iterator
Mesh::inner_halo_cell_iterator
Mesh::boundary_owned_face_iterator
...
```

The qualifier "`owned` / `halo`" is added just before the type specifier ("`cell`", "`node`" etc). If it is omitted, then the iterator covers all the elements, including both owned and halo ones.

Local and global identification of elements. In a sequential `Container<T>` each element is uniquely determined by its local identifier, `lid`, which, recalling that the container is derived from the class `vector<T>`, refers to its position in the vector. However, in a `DistributedContainer<T>` the `lid` only identifies the element locally, i.e. different elements owned by different parallel processes may have the same `lid`. In order to uniquely determine each element, global identifiers, `gids`, are also used.

For each parallel process, while the `lids` are consecutive the `gids` may be not. As a consequence, the correspondence between the `gids` and `lids` needs to be stored in a `map<int,int>` referred to as `g2l` (`lid=*g2l.find(gid)`), while the correspondence between `lids` and `gids` can be stored in a `vector<int>` referred to as `l2g` (`l2g[lid]=gid`).

Any distributed container iterator has the member function `Iterator::gid()` that returns the `gid` of the element being pointed by it. The `DistributedContainer<T>` class has also the member functions `DistributedContainer<T>::getG(gid)` and `DistributedContainer<T>::setG(gid,T)`, to read and write elements of the container using the global identifier, respectively. In these functions, the map `g2l` is used to obtain the `lid` of the required element, and then it is extracted from the storage vector. Therefore, is preferable to use the functions "`getL(...)`" and "`setL(...)`", and avoid the search operation (`map<T,T>::find(key)`) necessary to obtain the `lid`.

Communication scheme. Once the halos of a distributed container are fixed, it is necessary to determine the communication scheme. This means, knowing which processes need to communicate with which others and what information they must exchange. This scheme is calculated on a pre-processing stage prior to carrying out any halos update.

The update of halos can be performed by all parallel processes simultaneously, at a certain point of the code, or separately, with processes sending or receiving information at different moments of the execution. In the second chapter of this thesis (pag. 53), there is a description of a Poisson solver in which all communications are carried out simultaneously, basically before the matrix vector products. The reason is that all processes involved in the parallel execution work almost synchronously, doing the same operations in different parts of the domain. As a consequence, the halo requirements arrive at the same time for all of them, so they can all update halos in only one communication episode. On the other hand, in the Boltzmann transport equation solver explained in the third chapter (pag. 93), several triangular systems are solved simultaneously, one for each angular ordinate of the discretization. In this case, since different processes can work with different angular directions at the same time, there is no synchronization and, as a consequence, the communications are carried out in an asynchronous mode at any time when the data required is available.

The information to be communicated in a halo update is managed by means of the class `CommBuffer<T>`. If the elements to be sent/received are all of the same type, this is used to parameterize the buffer. Otherwise, the `char` type is adopted and binary copies of elements from/to the buffer are carried out. For this purpose, the function `memcpy` of the C standard library is used.

Simultaneous halo updates are carried out by means of the member function `DistributedContainer<T>::update()`: each parallel process calls the non-blocking MPI routines `MPI_Isend` and `MPI_Irecv`, to initialize all required send and receive operations, and then the function `MPI_Wait` to halt the process until all operations are completed. In some situations (for example when the messages are too large) initialization of all the send and receive operations at once overflows the network. A better alternative, in these cases, is the function `DistributedContainer<T>::updateVizing()`, in which the communications are arranged by pairs of processes, with the routine `MPI_Sendrecv`. In order to maximize the number of simultaneous bilateral communications, an analogy with the edge coloring problem of the graph theory is used: two adjacent edges (bilateral communications involving a same parallel process) must have different colors (must not be performed at the same time). A constructive proof of the Vizing's[4] theorem [33] is used to obtain the optimal solution.

For a given `DistributedContainer<T>`, the owned elements of each parallel process, the halo requirements, the correspondence between global and local identifications and the communication scheme, are managed by means of the class `ParallelTopology`. Each distributed container has a pointer to a `ParallelTopology` object, which can be used to define the parallelism of several `DistributedContainer<T>`. This way, halo updates of multiple containers can be performed simultaneously reducing the latency costs. An example of the parallel topologies usage is shown below:

```
Mesh mesh("meshfile");
mesh.close():

const ParallelTopology       nodtop=mesh.getNodeTopo();
DistributedContainer<double> temp(nodtop);
DistributedContainer<D3>     vel(nodtop);
```

In the function `Mesh::close()` the parallel topology associated to the nodes partition is generated, for each parallel process the halo of the node's container are the boundary nodes owned by others. In this example, this topology is used to define the parallel structure of the temperature and velocity fields, respectively.

In general, halo updates are the most expensive communication episodes in a simulation, although, other types of communications are also needed . For example, a global variable such as the time-step is fixed as the minimum between all the local time-steps. In these cases the MPI functions that fit with our needs are directly used.

1.4 Linear Solvers Library

The linear solvers library (LSL), programmed on the top of the BOL, is an algebraic library composed of different general purpose and application-specific algorithms to solve linear systems of equations. It is mainly used to deal with the linear systems derived from the discretization of the partial differential equations (PDEs) that model the physical phenomena being simulated.

Initially, only the Navier-Stokes equations for incompressible flows were solved

[4]Vizing's theorem states that an undirected graph can be edge-colored in either Δ or $\Delta + 1$ colors, where Δ is the maximum degree of the graph (i.e. the maximum number of edges incident to a vertex).

with TermoFluids code. The fractional-step method [34,35] adopted for its numerical resolution, requires the solution of a Poisson system at each time step to meet the incompressibility constraint. As a consequence, the LSL was initially conceived for the solution of this ill-conditioned elliptic system in simulations of different characteristics: general 3D, 2D, 3D but with one homogeneous periodic direction, executed sequentially, executed in parallel with few number of CPU, executed in parallel with large number of CPU, etc. For each kind of simulation there is a better option in the code. Actually, with the evolution of TF to a multi-physics code, new systems of equations with different characteristics need to be solved. In almost all cases, the solution of them is one of the most time consuming and computationally intensive parts of the execution.

The solution of regular linear systems is a problem with a unique solution for which are known, since hundreds of years ago, solution methods such as the classical Gaussian elimination algorithm. Therefore, if we had computers infinitely fast and with infinite memory capacity, this would not be a problem. However, resources are limited and classical algorithms can only deal with relatively small systems, far from the requirements of problems with industrial interest. Therefore, this is a practical problem in which the goal is, given a computer system, to obtain a sufficiently accurate solution in the fastest possible way. Once we have an enough accurate and fast algorithm for our needs, the problem is finished. In the literature there are many methods, some for general applications, such as the Multigrid or Krylov solvers [29,36,37], and others more specific, like the two methods shown in the following chapters of this thesis. The constant evolution of parallel computing systems, doubling their capacity approximately every thirteen months [3], makes this a very dynamic field because, although qualitative changes in the algorithmic are not so common, at least it is necessary to adjust the strategies to the available computational resources.

In the second chapter of this thesis, there is a description of a Poisson solver for the solution of discretizations in which one of the directions of the domain is uniformly meshed and has periodic boundary conditions. These hypothesis fit with the isotropic nature of some flows, such as the flow around a cylinder [38], the flow around a sphere [20] or the flow around an airfoil [39]. Under these conditions, the system can be decomposed into a set of 2D subsystems by means of a Fourier diagonalization. As a consequence, the memory requirements for the direct solution of these subsystems becomes affordable and a direct Schur complement based method is applied to them. This methodology has been tested on large systems of equations with up to 10^9 unknowns using up to 8192 CPUs [10].

In the third chapter of this thesis, a solver for the solution of the Boltzmann transport equation, discretized with a first order upwind-like numerical scheme, is presented. Upwind-like schemes result in a triangular subsystem for each angular ordinate, which can be solved very efficiently by means of a forward substitution, sweeping the nodes from the upstream to the downstream zones of the domain. In this case, the most challenging aspect is the parallelization because the forward substitution process is sequential in nature. This goal can be achieved by solving all directions at the same time: each angular ordinate has a different sweeping order and the inactivity for some ordinates is compensated with the solution of others. This method has been tested for discretizations with up to 240×10^6 unknowns on up to 2560 CPUs [11].

Apart from these two specific contributions, the rest of solvers that compose the LSL library are standard general purpose methods. In particular Krylov subspace methods with different preconditioners such as the Jacobi diagonal scaling, incomplete factorizations or inverse preconditioners. Their proper implementation in general purpose GPUs is now being investigated [31]. In some cases we also use direct solvers such as the sparse LU and Cholesky factorizations, or the Schur complement based decompositions for parallel executions. The choice of the appropriate method depends on the characteristics of the problem. These are some considerations arising from our experience:

- *Preprocessing cost.* We define the preprocessing stage of a solver as the operations that are carried out with the system matrix before touching the r.h.s. term. For example, in a LU decomposition the preprocessing stage is the evaluation of the L and U factors. The total cost of the solver in a simulation is the cost of the pre-processing plus the solve phases. If the system matrix does not vary during the simulation, then the preprocessing stage is carried out only once. In these cases, if the number of calls of the solve phase is high, then a pre-processing stage with large costs can be accepted because the real cost per iteration becomes small. This is the situation that occurs with the Poisson system in time-accurate simulations with a constant mesh. In this situation, any pre-process that accelerates the solution phase is profitable. On the contrary, when the system matrix varies, the pre-processing stage is repeated each time step so its cost can not be neglected.

- *Direct vs iterative methods.* This is not a real controversy because in general 3D cases the use of direct solvers is limited to small meshes, due to the large pre-processing costs and memory requirements. The limit depends on the hardware and software being used but, in our case, 3D systems with no more than one million of unknowns could be afforded. As a general rule, direct solvers can only be used in special cases in which the system has a favorable structure, like the two methods shown in the following chapters of this thesis.

  When applicable, direct solvers are usually faster and obviously more robust. However they do not benefit from good initial approximations, which can reduce the number of required iterations of the iterative methods. For example, in time-accurate DNS or LES simulations, in which the time-step is usually really small (to capture all the significant temporal scales), good initial guesses for the Poisson system can be obtained from solutions of previous time steps, and few iterations are necessary to reach convergence. In fact, in these kind of simulations, rather than the solution of the linear system, the most important limiting factor, in terms of computational requirements, are the tiny time steps that result in large number of time-iterations. In RANS simulations, this situation changes: the length of the time step grows and, as a consequence, the cost of the iterative solution of the Poisson system.

- *Parallelism.* Parallel computation consists in using multiple parallel processes to solve the same computational problem. Given a linear system of equations, a parallel solver is expected to reduce its execution time when the number of processes involved in the execution grows, i.e. that it has a good *strong speedup* or acceleration. Moreover, it is also expected that, if the size of the linear system and the number of parallel processes involved in the execution grow proportionally, the execution time keeps constant (or grow slowly), i.e. that the algorithm has a good *weak speedup.* The difficulty to achieve the first objective is that, when the number of parallel processes increases, the communications between them become more costly respect to the computations (that get reduced). At the end, it comes a stagnation or even a slowdown. On the other hand, the major difficulty in getting a good week speed up is that, normally, the asymptotic complexity of the algorithms is greater than $\mathcal{O}(N)$ and, therefore, the operations to perform per parallel process grows with the size of the problem. Anyways, with any number of processors the optimal method will be the one that goes faster, not the one that has better speedup respect to one processor. When a particular problem has to be solved, the choice of a solver is a matter of speed not of acceleration. For instance, a method that has already stalled its acceleration could be better than one that is accelerating superlinearly. However, generally, when increasing the number of parallel processes involved in the execution, the most suitable methods become simpler, as more often than not complexity (direct solvers, incomplete factorizations, sparse inverse preconditioners with complex sparsity patterns, multilevel methods. . .) slows down acceleration and at a certain point becomes less efficient.

All the solvers of the LSL library are derived from the abstract class[5] `Solver`, with virtual member functions `Solver::solve(...)` and `Solver::setUp()`. Therefore, any solver needs to have defined these two member functions. The class `Solver`

[5]An abstract class is, conceptually, a class that can not be instantiated. Is, therefore, a class without objects. It has one or more pure virtual member functions (without definition) that must be defined by any derived class.

has, as private elements, a pointer to an object of the class `SparseMatrix` (the system matrix) and to an object of the class `Parameters`. The class `Parameters` is a container of parameters of different type, used to avoid filling the member functions of the solvers with parametric arguments. In fact, each solver has associated a particular class of parameters derived from the base class `Parameters`, in which default values are fixed by construction.

Conceptually, each solver, as a class of the code, incorporates as private elements the most important data generated during the solution process, which is carried out by the member functions. The `SparseLU` class, for instance, incorporates as private elements the distributed sparse matrices L and U and the member functions `LU::reordering()` and `LU::factorization()`, to carry out a fill-reducing ordering of the unknowns and the factorization, respectively .

1.5 Example

The discretization and solution of the Poisson equation with Neumann boundary conditions is shown below, as an example of how is it to program with the BOL and LSL libraries. The derived linear system of equations is solved using a conjugate gradient (CG) Krylov subspace method with a sparse approximate inverse (SAI) preconditioner.

```
#include  "mpi.h"
#include  <bol/BOL.h>
#include  <lsl/LSL.h>

using namespace bol;
using namespace lsl;

int main(int argc, char **argv) {

   mpi_init(argc,argv,true);
   if(argc-1 != 1) {crash<<"a.out <Mesh_file> "<<endLine; return 0;}

   Mesh mesh(argv[1]);
   mesh.close();

   ParallelTopology&            topo= mesh.getInnerNodeTopo(); //boundary nodes discarted
   SparseMatrix                 A(topo);
   DistributedContainer<double> b(topo),x(topo);

   //Definition of the system matrix
   for(Mesh::const_inner_owned_node_iterator i=mesh.ionBegin(); i!=mesh.ionEnd(); ++i){

      double sum=0;

      for(Node::const_neighbor_node_iterator j=i->nnBegin(); j!=i->nnEnd(); ++j)
        if(!j->isBoundary()){
           double coef = j->getFaceSurface()/(*j-*i)*j.getFaceNormVect());
           A.insertL(i.lid(),j.lid(),-coef);
           sum+=coef;
        }

      A.insertL(i.lid(),i.lid(), sum);
   }

   SAI_param param_prec;       //Default parameters are used
   SAI  prec(A, param_prec);
   prec.setUp();

   CG_param param_cg;
   param_cg.setDouble("tol", 1.e-08); //The tolerance is fixed

   CG   solver(A, prec, param_cg);
   solver.setUp();

   b=1;
   solver.solve(b,x);

   x.update();
   double residual= (b-A*x).norm2();

   cout<<"The residual obtained is:"<< residual<<endl;
   cout<<"The number of iterations:"<< param_cg.getInt("niter")<< endl;

   mpi_end();
   return(0);
}
```

1.6 Industrial projects

In the context of collaboration between the CTTC and Termo Fluids SL, many industrial and research projects have been developed. The numerical simulation of fluids is an important tool to better understand the problems under study as well as to improve the industrial designs. In this thesis I have not directly worked in these projects, however, the two basic libraries described above are an important part of the basis on which they stand. Furthermore, the interaction with the scientists who develop these projects and its almost impossible to fulfill requirements, are one of the key aspects that drive us to further improve the code and the numerical methods used in it. In what follows, two examples of industrial applications are summarized.

Numerical simulation of wind turbine dedicated airfoils. The flow around aerodynamic profiles in pre- or full-stall state at high Reynolds numbers is a problem of increasing interest since it is a normal operation state for wind turbine blades. In the past years, it has been subject of many experimental and numerical investigations. Most of the numerical studies performed up to now have been carried out using RANS modeling, but it is well-known that such models fail in predicting the flow at angles of attack (AoA) near or after the stall, mainly due to the highly unsteady nature of the flow. In these situations, large-eddy simulations (LES) can be a good alternative for simulating such complex flows.

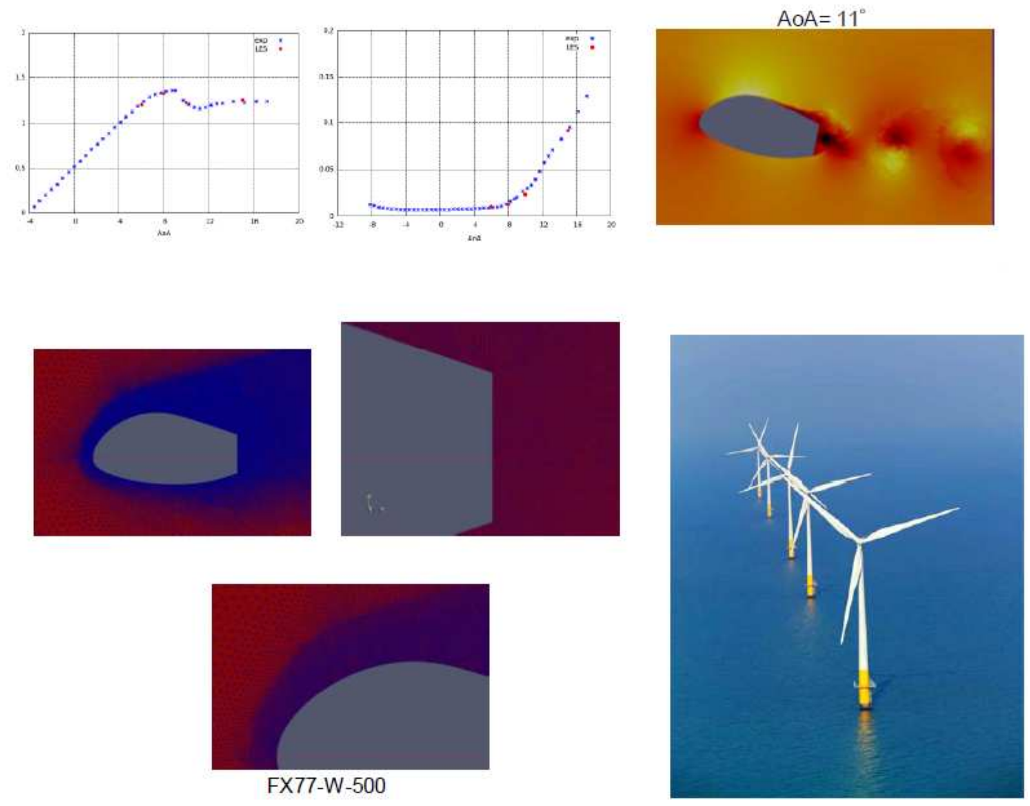


Figure 1.2: Illustrative results for the FX77-W-500 airfoil: mesh design, instantaneous snapshots and C_d and C_l distributions.

This R&D project between the CTTC-TF and other European research centers and universities aims at modeling wind turbine airfoils at high Reynolds numbers and high AoA. Initially three profiles have been selected (DU-93-W-210, DU-91-W2-250 and FX77-W-500) for Reynolds numbers up to $3 \cdot 10^6$ and AoA up to 16°. Numerical results have been compared with experimental ones showing a good agreement [40].

Domestic refrigerator analysis. This R&D project between Fagor and the CTTC-TF was focused on the study of the temperature and air distribution inside household frost-free refrigerators. It is well known that the correct air and temperature distribution inside the refrigerated chamber is the most important factor that affects refrigerator efficiency. In frost-free refrigerators, the cooled air is supplied directly inside the fresh food and vegetable cabinets. Therefore, several studies were performed to establish the air flow and temperature distributions inside these cabinets, in order to improve temperature homogeneity and to reduce energy consumption.

Unsteady three-dimensional numerical studies were carried out, simulating the cooling process starting from a uniform warm temperature inside the refrigerator. Furthermore, the influence of inlet and outlet ports location was also investigated [23].

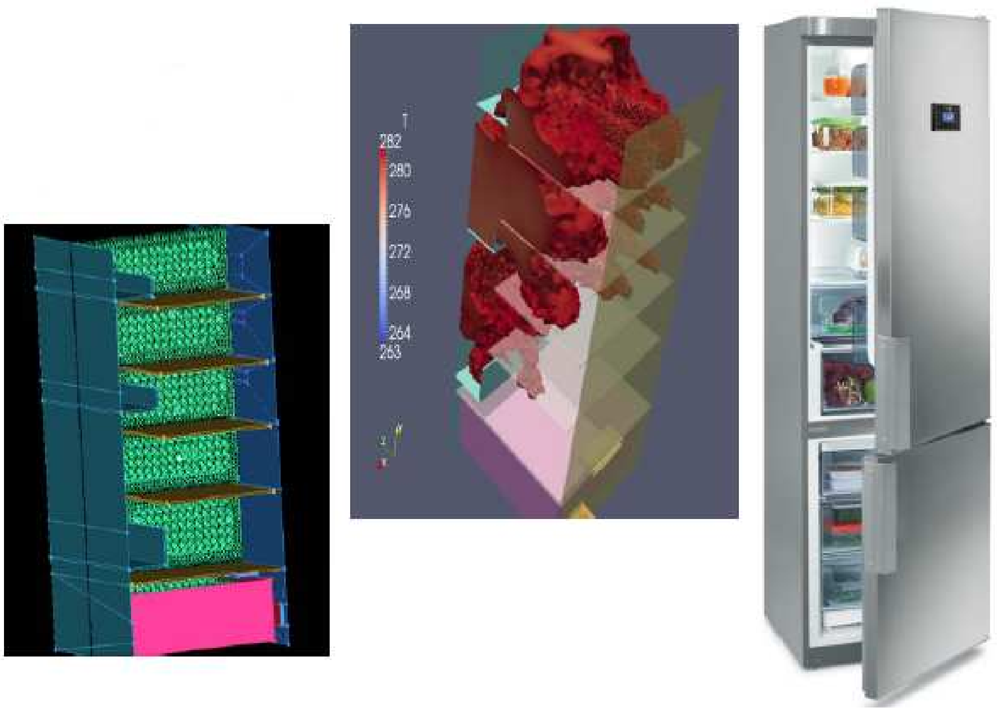


Figure 1.3: Schematics of the household frost-free refrigerator modeled, mesh details and instantaneous snapshot.

1.7 Concluding remarks

Numerical methods and models developed in the framework of the computational fluid dynamics field, are intended to be finally implemented in CFD codes. Proper implementation of these methods and models, to take advantage of the existing supercomputing facilities, is not a minor issue and requires research efforts to progress. Otherwise, we could not benefit from the exponential growth of computing resources to achieve the numerical solution of increasingly complex fluid flows.

In the context of this thesis, there has been a contribution to the development of a general purpose multi-physics parallel CFD software, referred to as TermoFluids, composed of several libraries arranged in a hierarchical scheme. The functionalities and the parallel efficiency of the code are dynamic characteristics, because it is constantly being enhanced by an increasing number of developers in interaction with users. Particularly, in this thesis the work has been focused on the development of two of the most basic libraries that compose TF code, which have been described in this chapter. These are the Basic Object Library, which is an unstructured CFD application programming interface on the top of which are written the rest of TF libraries; and the Linear Solvers Library, which has several general purpose and application specific methods, for the solution of the linear systems that result from discretizations.

These libraries have been designed following the intuitive object oriented paradigm of the C++ language, that allows to expand the code in an orderly and compact form. The classes representing the main concepts treated by the libraries have also been introduced. An implementation for the solution of a Poisson equation on the top of the BOL and the LSL, has also been presented as an example of their user-friendly intuitive design. A proof of it is also the rapid expansion of TF to a multi-physics CFD code, which involves several developers working simultaneously.

Parallelism is another basic design feature of the code. TF is mainly programmed following the distributed memory paradigm. Basic concepts such as the domain decomposition, the definition of halo elements, the local and global identifiers or the communication schemes, have been described in detail. The implications of the parallelism in the design and applicability of a linear solver have also been briefly discussed.

Finally some industrial problems, in which TF code has been an important tool to better understand the physics under study and improve the industrial designs, have been presented.

References

[1] M.T. Dröge. *Cartesian Grid Methods for Turbulent Flow Simulation in Complex Geometries.* PhD thesis, Rijksuniversiteit Groningen, 2007.

[2] S. Hoyas and J. Jiménez . Scaling of the velocity fluctuations in turbulent channels up to Re_τ=2003. *Physics of Fluids*, 18(1):011702, 2006.

[3] H. W. Meuer. Supercomputers – Prestige Objects or Crucial Tools for Science and Industry? In *House of Lords*, April 2012. http://www.top500.org/files/Supercomputers_London_Paper_HWM_HG.pdf.

[4] R. W. C. P. Verstappen and A. E. P. Veldman. Symmetry-Preserving Discretization of Turbulent Flow. *Journal of Computational Physics*, 187:343–368, May 2003.

[5] F.N. Felten and T.S. Lund. Kinetic energy conservation issues associated with the collocated mesh scheme for incompressible flow. *Journal of Computational Physics*, 215(2):465–484, 2006.

[6] O. Lehmkuhl, R. Borrell, C.D. Pérez-Segarra, A. Oliva, and R.W.C.P. Verstappen. LES modeling of the turbulent flow over an Ahmed car. In *DLES8, workshop on Direct and Large-Eddy Simulation*, Eindhoven (The Netherlands), July 2010. Springer.

[7] O. Lehmkuhl, R. Borrell, F. X. Trias, and C.D. Pérez-Segarra. Assessment of large-eddy simulation models in complex flows using unstructured meshes. In *Fifth European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS*, Venice (Italy), July 2008.

[8] O. Lehmkuhl, R. Borrell, I. Rodríguez, C.D. Pérez-Segarra, and A. Oliva. Assessment of the symmetry-preserving regularization model on complex flows using unstructured grids. *Computers and Fluids*, 60:108–116, 2012.

[9] O. Lehmkuhl, F. X. Trias, R. Borrell, and C.D. Pérez-Segarra. Symmetry-preserving Regularization modelling of a turbulent plane impinging jet. In *DLES7, workshop on Direct and Large-Eddy Simulation*, Trieste (Italy), July 2008.

[10] R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva. Parallel direct Poisson solver for discretisations with one Fourier diagonalisable direction. *Journal of Computational Physics*, 230(12):4723–4741, 2011.

[11] G. Colomer, R. Borrell, F.X. Trias, and I. Rodríguez. Parallel algorithms for S_n transport sweeps on unstructured meshes. *Journal of Computational Physics*, (in press).

[12] G. Colomer, R. Borrell, O. Lehmkuhl, and A. Oliva. Parallelization of combined convection-radiation numerical simulations. In 14*th International Heat Transfer Conference*, Washington D.C. (USA), Agust 2010.

[13] J. Ventosa, J. Chiva, O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. Low Mach Navier-Stokes equations on unstructured meshes. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.

[14] Ll. Jofre, N. Balcazar, O. Lehmkuhl, J. Castro, and A. Oliva. Numerical study of the incompressible Richtmyer-Meshkov instability. Interface traking methods on general meshes. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.

[15] N. Balcazar, Ll. Jofre, O. Lehmkuhl, J. Rigola, and J. Castro. Numerical simulation of incompressible two phase flow by conservative level set method. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.

[16] Ll. Jofre, R. Borrell, O. Lehmkuhl, and A. Oliva. Parallelization of the Volume-of-Fluid method for 3D unstructured meshes. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[17] O. Estruch, O. Lehmkuhl, R. Borrell, and C.D. Pérez-Segarra. Large-eddy simulation of turbulent FSI. In *7th Symposium on Turbulence, Heat and Mass Transfer, THMT-12*, Palermo (Italy), September 2012.

[18] O. Estruch, R. Borrell O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. A parallel radial basis function interpolation method for unstructured dynamic meshes. *Computers and Fluids*, (in press).

[19] J. Lopez, O. Lehmkuhl, R. Damle, and J. Rigola. A parallel and object-oriented general purpose code for Simulation of Multiphysics and Multiscale Systems. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[20] I. Rodríguez, R. Borrell, O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. Direct numerical simulation of the flow over a sphere at Re = 3700. *Journal of Fluid Mechanics*, 679:263–287, 2011.

[21] I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Flow dynamics in the turbulent wake of a sphere at sub-critical Reynolds numbers. *Computers and Fluids*, (in press).

[22] G. Colomer, O. Lehmkuhl, R. Borrell, and A. Oliva. CFD simulation of the thermal behavior of a wind mill power generator nacelle. In 6*th International Symposium on Turbulence, Heat and Mass Transfer*, Rome (Italy), September 2009.

[23] J. Jaramillo, J. Rigola, C.D. Pérez-Segarra, and C. Oliet. Numerical Study of Air inside Refrigeration Compartment of Frost-Free Domestic Refrigerators. In

International Refrigeration and Air Conditioning Conference, Purdue (USA), July 2010.

[24] E. Ogasawara and D. de Oliveira and F. Chirigati and C.E. Barbosa and R. Elias and V. Braganholo and A. Coutinho and M. Mattoso. Exploring many task computing in scientific workflows. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. ACM, 2009.

[25] ICEM CFD Hexa. *ANSYS Inc.* Canonsburg, PA, USA, http://www.ansys.com.

[26] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[27] B. Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.

[28] C.L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[29] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2003.

[30] V. Heuveline, M. J. Krause, and J. Latt. Towards a hybrid parallelization of lattice boltzmann methods. *Computers and Mathematics with Applications*, 58(5):1071–1080, 2009.

[31] G. Oyarzun, R. Borrell, O. Lehmkuhl, and A. Oliva. Hybrid MPI-CUDA approximate inverse preconditioner. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[32] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1999.

[33] J. Misra and D. Gries. A constructive proof of vizing's theorem. *Inf. Process. Lett.*, 41(3):131–133, 1992.

[34] A. J. Chorin. Numerical Solution of the Navier-Stokes Equations. *Journal of Computational Physics*, 22:745–762, 1968.

[35] N. N. Yanenko. *The Method of Fractional Steps.* Springer-Verlag, 1971.

[36] Jonathan R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, 1994.

[37] P. Wesseling. *An Introduction to Multigrid Methods.* Wiley, 1992.

[38] O. Lehmkuhl, I. Rodríguez, R. Borrell, C.D. Pérez-Segarra, and A. Oliva. Low frequency variations in the wake of a circular cylinder at $Re = 3900$. In *13th European Turbulence Conference*, Varsaw (Poland), September 2011.

[39] I. Rodríguez, O. Lehmkuhl, A. Baez, R. Borrell, and A. Oliva. Direct numerical simulation of a NACA 0012 in full stall. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.

[40] O. Lehmkuhl, J. Calafell, I. Rodríguez, and A. Oliva. Large-Eddy Simulations of wind turbine dedicated airfoils at high Reynolds numbers. In *Wind Energy and the impact of turbulence on the conversion process. EUROMECH Colloquium 528*, Oldenburg (Germany), February 2012.

Parallel direct Poisson solver for discretizations with one Fourier diagonalizable direction

Main contents of this chapter have been published in:
R. Borrell, O. Lehmkuhl, F. X. Trias and A. Oliva, Parallel direct Poisson solver for discretisations with one Fourier diagonalisable direction, *Journal of Computational Physics*, 230 (12), pp. 4723-4741, 2011.

Abstract. In the context of time-accurate numerical simulation of incompressible flows, a Poisson equation needs to be solved at least once per time-step to project the velocity field onto a divergence-free space. Due to the non-local nature of its solution, this elliptic system is one of the most time consuming and difficult to parallelise parts of the code.

In this chapter, a parallel direct Poisson solver restricted to problems with one uniform periodic direction is presented. It is a combination of a direct Schur-complement based decomposition (DSD) and a Fourier diagonalization. The latter decomposes the original system into a set of mutually independent 2D subsystems, which are solved by means of the DSD algorithm. Since no restrictions are imposed in the non-periodic directions, the overall algorithm is well-suited for solving problems discretized on extruded 2D unstructured meshes. The load balancing between parallel processes and the parallelization strategy are also presented and discussed. The scalability of the solver is successfully tested using up to 8192 CPU cores, for meshes with up to 10^9 grid points. Moreover, the performance of the DSD algorithm as 2D solver, is analyzed by direct comparison with two preconditioned conjugate gradient methods. For this purpose, the turbulent flow around a circular cylinder at Reynolds numbers 3900 and 10000 is used as problem model. Finally some illustrative applications of the solver are outlined.

2.1 Introduction

Direct numerical simulation (DNS) and large-eddy simulation (LES) of incompressible turbulent flows demand huge computing power and require parallel computers to be feasible. The Poisson equation, which arises from the incompressibility constraint and has to be solved at least once per time step, is usually the most time-consuming and difficult to parallelise part of the algorithm. Therefore, in this context the development of efficient and scalable Poisson solvers is of great interest.

Time-accurate DNS/LES simulations generally demand large amount of time-steps (for DNS applications it can reach $\sim 10^6$). If the mesh does not change during the simulation, the Poisson equation is solved repeatedly with different right-hand-side terms, while the system matrix remains constant. In these cases, a solver with large computing pre-processing demands can be accepted. Another usual feature in DNS/LES applications, is to have at least one periodic homogeneous direction in the flow. This property makes the Fourier diagonalization [1, 2] in the periodic direction(s) the best choice. The uniformity of the grid in such directions, imposed by the method, is suitable with the isotropic nature of the flow on them. This work is restricted to problems with only one periodic homogeneous direction, examples of this kind of configuration can be found in [3–6]. No restrictions are imposed for the non-periodic spatial directions, therefore, the method here proposed is suitable for discretizations on extruded 2D unstructured meshes.

Different strategies can be used to solve the set of 2D subsystems resulting from the diagonalization. The most widespread iterative options are the conjugate gradient (CG) [7, 8], and the multigrid [9, 10] methods. The former is easy to implement and shows good scalability. However, its performance is strongly dependent on the spectral condition number of the system and usually requires an application-specific preconditioner [11]. On the other hand, the convergence rate of multigrid methods does not depend on the problem size, and they work very well for certain problems. However, they also require a careful application-specific tuning. Alternatively, direct Schur-complement based methods [12–15], perform irrespectively of the condition number or the specific application and depend only on the sparsity pattern of the system matrix. As a main drawback, they require a computationally demanding pre-processing phase and large memory resources. Nevertheless, as mentioned above, this additional pre-processing cost becomes almost negligible for the time-accurate numerical simulations here considered. And, since the subsystems in which it is applied have a *two-dimensional* sparcity patten, the memory requirements remain still feasible for the range of mesh sizes required for DNS/LES applications.

2.1.1 Motivation and summary of the present work

In this chapter, a direct Schur complement based decomposition (DSD) method is proposed, for the set of mutually independent 2D subsystems resulting from the Fourier diagonalization. For a given a domain decomposition, this method is based in dividing the initial system into a set of inner subsystems, coupled by the interface or Schur-complement subsystem. The unknowns of each subdomain are partitioned into the inner and interface subsets in such a way that two inner unknowns of different subdomains are not coupled by the system matrix. On the solution phase, the inner subsystems are solved independently by each parallel process, while the interface subsystem is solved by all of them with a parallel method. Essentially, Schur complement based algorithms may differ in three aspects: (i) the determination of the interface that separates the inner unknowns of each subdomain, (ii) the solver used for the Schur complement (or interface) subsystem, and (iii) the solver used for the inner subsystems. The prevailing option is to solve the interface subsystem by means of a preconditioned Krylov projection method [8,16,17]. In this case, it is common to use a *double-sided interface*, composed of the nodes of each subdomain which are linearly coupled with nodes of other subdomains. With this strategy, there are at least two interface *mesh-lines* between each pair of inner subdomains. This approach is convenient because the interface can be set locally, and the resulting block structure of the Schur complement matrix is well determined *a priori* [16]. On the other hand, in some situations, usually with relatively low numbers of CPUs, it might be more convenient to use a direct solver for the interface subsystem. To do so, in some works [13,18,19], it is solved by means of a parallel factorization or gathered into a master process to solve it sequentially. Another possible approach is to explicitly compute the inverse of the Schur complement matrix and distribute it among the processing elements [12,20]. In these situations, due to the high memory requirements of direct solvers, a *one-sided* interface strategy (a single *mesh-line* separating the inner subdomains) is more convenient, because the resulting interface size is approximately halved.

In our previous works [12,14], a DSD algorithm in conjunction with a fast Fourier transform (FFT) was successfully used to perform DNS/LES simulations on structured Cartesian grids [3,21]. The interface was naturally built in a balanced manner, a band LU [22] was used as local solver and the inverse of the Schur complement matrix was calculated and stored in parallel. The distribution of data between the parallel processes was carried out in such a way that only a collective communication was needed in the solution phase. Although the latter implied some additional duplication of data, this approach was very suitable for parallel systems with a high latency network. In this chapter, the extension of the DSD algorithm to general unstructured grids and modern supercomputers is presented. To do so, modifications of the three above-mentioned characteristic issues have been introduced. Namely, (i) a new algorithm to balance the *one-sided* interface is described, (ii) the local solver has been replaced by a sparse Cholesky factorization [22], and finally, (iii) the Schur complement matrix is now treated sparsely instead of as a dense matrix. Besides, the distribution of data between the parallel processes has been modified in order to decrease the memory costs. This new strategy requires two additional point-to-point communications the cost of which is almost negligible on modern supercomputers. All these changes have significantly accelerated the set-up and solve phases of the algorithm.

With regard to the overall parallelization strategy, some important improvements are introduced. In our previous works [11, 14], the same partition was used for both physical and spectral spaces. This was a convenient approach for relatively low numbers of CPUs. However, this approach eventually limits the number of parallel processes in the periodic direction. In the present version of the algorithm, in order to overcome this drawback, the physical and spectral partitions can be chosen independently. With this new strategy, the range of efficient scalability in the periodic direction has been significantly increased: maximal tests with 128 parallel processes in the periodic direction show that the scalability is still not exhausted.

All the numerical tests have been carried out on the IBM MareNostrum supercomputer at the Barcelona Supercomputing Center. Computing times and speed-up tests, obtained for meshes with up to 10^9 nodes using 8192 CPUs, illustrate the robustness and scalability of the method. Moreover, a direct comparison of the DSD, as 2D solver, with two preconditioned conjugate gradient (PCG) [8] methods is also presented and discussed.

The rest of the chapter is arranged as follows. In Section 2.2, the numerical methods for the time-accurate solution of Navier-Stokes equations are briefly described. The Poisson solver is presented in Section 2.3. The parallelization strategy is discussed in Section 2.4. In Section 2.5, numerical experiments are carried out to test the performance of the proposed parallel solver on the MareNostrum supercomputer. In Section 2.6, the DSD is successfully compared with the PCG in the context of DNS simulations of the flow around a circular cylinder at $Re = 3900$ and $Re = 10000$. Some illustrative applications of the method are outlined in Section 2.7 and, finally, relevant results are summarized and conclusions are given in Section 2.8.

2.2 Numerical methods for DNS

2.2.1 Geometry discretization

In this work, geometric discretizations obtained by the uniform extrusion of generic 2D meshes are considered. Periodic boundary conditions are imposed in the extrusion direction, thus the linear couplings of the Poisson equation in such direction result into circulant submatrices. Since the proposed algorithm does not impose any restriction for the initial 2D mesh, it is suitable for unstructured meshes. Nevertheless, this lack of structure leads to a more complex data management. An illustrative example of such a geometric discretization is displayed in Figure 2.1.

The following notation is used. The initial 2D mesh and the 1D uniform discretization of the periodic direction are referred as $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively. The total number of nodes is $N := N_{2d}N_{per}$, where N_{2d} and N_{per} are the number of nodes in $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively. For simplicity, N_{per} is taken as an even number. The constant mesh step in $\mathcal{M}_{per}$ is Δ_{per}.

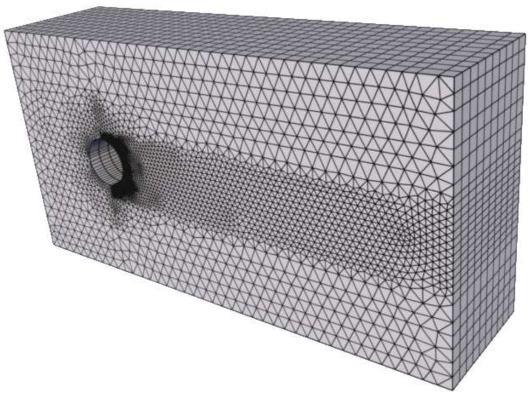


Figure 2.1: 3D mesh around a cylinder generated by the uniform extrusion of a 2D unstructured mesh.

Two indexes define the position of a node on the resultant 3D mesh $\mathcal{M}$, namely the projections in $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$. Hence, two different node orderings are used: the *2D-block-order* and the *1D-block-order*. They are lexicographical orders of the Cartesian products $\mathcal{M}_{per} \times \mathcal{M}_{2d}$ and $\mathcal{M}_{2d} \times \mathcal{M}_{per}$, respectively. Using the *2D-block-order*, a scalar field $v \in \mathbb{R}^N$ reads

$$v \equiv \left[v_0^{2d}, ..., v_{(N_{per}-1)}^{2d} \right], \tag{2.1}$$

whereas using the *1D-block-order* it becomes

$$v \equiv \left[v_0^{per}, ..., v_{(N_{2d}-1)}^{per} \right], \tag{2.2}$$

where $v_k^{2d} \in \mathbb{R}^{N_{2d}}$ and $v_k^{per} \in \mathbb{R}^{N_{per}}$ are the kth *plane of the extrusion* and the kth *span-wise* subvector, respectively.

2.2.2 Governing equations and spatial discretization

The simulation of turbulent incompressible flows of Newtonian fluids is considered. Under these assumptions the velocity field, u, is governed by the Navier-Stokes (NS) and continuity equations

$$\begin{aligned} \partial_t u + u \cdot \nabla u - \frac{1}{Re}\Delta u + \nabla p &= 0, && (2.3)\\ \nabla \cdot u &= 0, && (2.4) \end{aligned}$$

where Re is the dimensionless Reynolds number. In an operator-based formulation, the finite volume spatial discretization of these equations reads

$$\begin{aligned} \Omega \frac{d\boldsymbol{u}_h}{dt} + C\left(\boldsymbol{u}_h\right)\boldsymbol{u}_h + D\boldsymbol{u}_h + \Omega G \boldsymbol{p}_h &= 0_h, && (2.5)\\ M\boldsymbol{u}_h &= 0_h, && (2.6) \end{aligned}$$

where $\boldsymbol{u}_h$ and $\boldsymbol{p}_h$ are the discrete velocity and pressure fields, Ω is a diagonal matrix with the size of the control volumes, $C(\boldsymbol{u}_h)$ and D are the convective and diffusive operators and, finally, M and G are the divergence and gradient operators, respectively. In this chapter, a "symmetry-preserving"/"energy conserving" discretization is adopted: the convective operator is skew symmetric ($C(\boldsymbol{u}_h) + C(\boldsymbol{u}_h)^* = 0$), the diffusive operator is symmetric positive-definite and the integral of the gradient operator is minus the adjoint of the divergence operator ($\Omega G = -M^*$). This last requirement is not exactly satisfied due to the cell-to-face interpolation needed when defining the divergence operator in a collocated arrangement (for further details see [23]). Therefore, in practice, it is $\Omega G \approx -M^*$. Preserving the (skew-)symmetries of the continuous differential operators when discretizing them has been shown to be a very suitable approach for DNS [3, 24, 25].

2.2.3 Time-integration method

For the temporal discretization, a second-order explicit one-leg scheme is used. Thus, assuming $\Omega G = -M^*$, the resulting fully-discretized problem reads

$$\begin{aligned} \Omega \frac{\boldsymbol{u}_h^{n+1} - \boldsymbol{u}_h^{n}}{\delta t} &= R\left(\frac{3}{2}\boldsymbol{u}_h^{n} - \frac{1}{2}\boldsymbol{u}_h^{n-1}\right) + M^*\boldsymbol{p}_h^{n+1}, && (2.7)\\ M\boldsymbol{u}_h^{n+1} &= 0_h, && (2.8) \end{aligned}$$

where $R(\boldsymbol{u}_h) = -C(\boldsymbol{u}_h)\boldsymbol{u}_h - D\boldsymbol{u}_h$. The pressure-velocity coupling is solved by means of a classical fractional step projection method [26, 27]. In short, reordering the equation (2.7), an expression for $\boldsymbol{u}_h^{n+1}$ is obtained,

$$\boldsymbol{u}_h^{n+1} = \boldsymbol{u}_h^{n} + \delta t \Omega^{-1}\left(R\left(\frac{3}{2}\boldsymbol{u}_h^{n} - \frac{1}{2}\boldsymbol{u}_h^{n-1}\right) + M^*\boldsymbol{p}_h^{n+1}\right). \qquad (2.9)$$

Then, substituting this into (2.8), leads to a Poisson equation for $\boldsymbol{p}_h^{n+1}$,

$$-M\Omega^{-1}M^{*}\boldsymbol{p}_h^{n+1} = M\left(\frac{\boldsymbol{u}_h^n}{\delta t} + \Omega^{-1}R\left(\frac{3}{2}\boldsymbol{u}_h^n - \frac{1}{2}\boldsymbol{u}_h^{n-1}\right)\right), \tag{2.10}$$

which must be solved once at each time-step.

2.2.4 Poisson equation

The Laplacian operator of equation (2.10),

$$\mathsf{L} = -M\Omega^{-1}M^{*}, \tag{2.11}$$

is by construction symmetric and negative-definite. Its action on $\boldsymbol{p}_h$ is given by

$$[\mathsf{L}\boldsymbol{p}_h]_k = \sum_{j\in Nb(k)} A_{kj}\frac{\boldsymbol{p}_h(j) - \boldsymbol{p}_h(k)}{\delta n_{kj}}, \tag{2.12}$$

where $Nb(k)$ is the set of neighbors of the kth node. A_{kj} is the area of f_{kj}, the face between the nodes k and j, and $\delta n_{kj} = |n_{kj} \cdot v_{kj}|$, where v_{kj} and n_{kj} are the vector between the nodes and the unitary vector normal to f_{kj}, respectively (see Figure 2.2).

The set $Nb(k)$ can be split into two subsets: $Nb(k) = Nb_{per}(k) \cup Nb_{2d}(k)$, where $Nb_{per}(k)$ and $Nb_{2d}(k)$ refer to the neighbor nodes along the periodic direction and in the same plane of the extrusion, respectively. In this way, the expression (2.12) becomes

$$[\mathsf{L}\boldsymbol{p}_h]_k = \sum_{i\in Nb_{per}(k)} A_{ki}\frac{\boldsymbol{p}_h(i) - \boldsymbol{p}_h(k)}{\Delta_{per}} + \Delta_{per}\sum_{j\in Nb_{2d}(k)} a_{kj}\frac{\boldsymbol{p}_h(j) - \boldsymbol{p}_h(k)}{\delta n_{kj}}, \tag{2.13}$$

where a_{kj} is the length of the edges of f_{kj} contained in $\mathcal{M}_{2d}$ (see Figure 2.2). This can be written in a more compact form by means of the Kronecker product of matrices. Using the *1D-block-order*, the Laplacian operator of the equation (2.13) reads

$$\mathsf{L} = (\Omega_{2d} \otimes \mathsf{L}_{per}) + \Delta_{per}(\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}), \tag{2.14}$$

where $\mathsf{L}_{2d} \in \mathbb{R}^{N_{2d}\times N_{2d}}$ and $\mathsf{L}_{per} \in \mathbb{R}^{N_{per}\times N_{per}}$ are the Laplacian operators discretized on the meshes $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively; $\Omega_{2d} \in \mathbb{R}^{N_{2d}\times N_{2d}}$ is the diagonal matrix representing the areas of the control volumes of $\mathcal{M}_{2d}$, and $\mathsf{I}_{N_{per}}$ is the identity matrix of size N_{per}. With the above-mentioned conditions (uniformly meshed periodic direction), L_{per} results into a symmetric circulant matrix of the form

$$\mathsf{L}_{per} = \frac{1}{\Delta_{per}} circ(-2, 1, 0, \cdots, 0, 1). \tag{2.15}$$

This characteristic of L_{per} allows to use a Fourier diagonalization algorithm in the periodic direction.

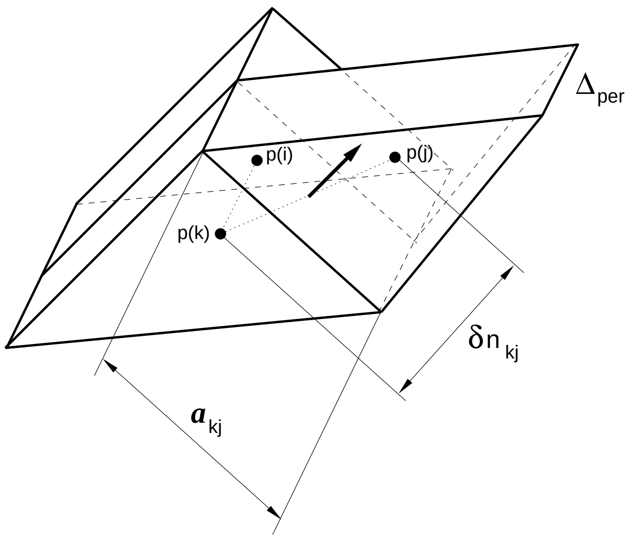


Figure 2.2: Elements of the geometric discretization.

2.3 Direct Poisson solver

In this section, the algorithm to solve the Poisson equation is described. As mentioned above, the problem under consideration reads

$$\mathsf{L}x_i = b_i \qquad i = 1, ..., N_t, \tag{2.16}$$

where the Laplacian operator, L, remains constant during all the simulation and N_t is the total number of time-steps. Under these conditions, the computational cost (per time-step) of any preprocessing phase is reduced by N_t times. Typically, for DNS applications $N_t = 10^5 \sim 10^6$. Therefore, in general, the pre-processing costs become negligible.

As the couplings in the periodic direction are circulant matrices, the initial system (2.16) can be diagonalized by means of a Fourier transform. As a result, it is decomposed into a set of N_{per} mutually independent 2D subsystems, drastically reducing the arithmetical complexity and the RAM memory requirements (see next subsection). Finally, the 2D problems are solved by means of a Direct Schur-complement based Decomposition (DSD) method, described in Subsection 2.3.2.

2.3.1 Fourier diagonalization overview

Any circulant matrix is diagonalizable by means of the discrete Fourier transform (DFT) of the same dimension (see Appendix A). Thus, the circulant matrix, L_{per}, defined in equation (2.15) verifies

$$\mathsf{F}^*_{N_{per}} \mathsf{L}_{per} \mathsf{F}_{N_{per}} = \Lambda, \tag{2.17}$$

where $\mathsf{F}_{N_{per}}$ and $\mathsf{F}^*_{N_{per}}$ are the N_{per}-dimensional Fourier transform and its inverse/adjoint, respectively; and $\Lambda = diag(\lambda_0, \lambda_1, ..., \lambda_{N_{per}-1})$ is the resultant diagonal matrix. A general expression for the eigenvectors can be found in Appendix A. In this particular case

$$\lambda_k = -\frac{2}{\Delta_{per}}\left(1 - \cos\left(\frac{2\pi k}{N_{per}}\right)\right) \qquad k = 0, ..., N_{per} - 1. \tag{2.18}$$

If the unknowns of any field, v, defined in $\mathcal{M}$ are labelled adopting the *1D-block-order*, the operator $(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})$ transforms all the *span-wise* subvectors, v_k^{per}, from the *physical* to the Fourier *spectral* space; $(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})$ carries out the inverse transformation. Applying the same change-of-basis to L, the Laplacian operator in the *spectral* space, $\widehat{\mathsf{L}}$, is obtained: [1]

$$\begin{aligned}
\widehat{\mathsf{L}} &= (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})\mathsf{L}(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}}) = \\
&= (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})((\Omega_{2d} \otimes \mathsf{L}_{per}) + \Delta_{per}(\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}))(\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}}) = \\
&= (\mathsf{I}_{N_{2d}}\Omega_{2d}\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}}\mathsf{L}_{per}\mathsf{F}_{N_{per}}) + \Delta_{per}(\mathsf{I}_{N_{2d}}\mathsf{L}_{2d}\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}}\mathsf{I}_{N_{per}}\mathsf{F}_{N_{per}}) = \\
&= (\Omega_{2d} \otimes \Lambda) + \Delta_{per}(\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}).
\end{aligned} \tag{2.19}$$

Comparing the last expression term-by-term with equation (2.14), it is observed that the change-of-basis only affects the couplings in the periodic direction, whereas the couplings in the non-periodic directions are not modified. This is a consequence of the mesh uniformity in the periodic direction. Switching to the *2D-block-order*, $\widehat{\mathsf{L}}$ reads

$$\widehat{\mathsf{L}} = (\Lambda \otimes \Omega_{2d}) + \Delta_{per}(\mathsf{I}_{N_{per}} \otimes \mathsf{L}_{2d})) = \bigoplus_{k=0}^{N_{per}-1} \widehat{\mathsf{L}}_k, \tag{2.20}$$

where

$$\widehat{\mathsf{L}}_k = \lambda_k \Omega_{2d} + \Delta_{per}\mathsf{L}_{2d} \qquad k = 0, ..., N_{per} - 1. \tag{2.21}$$

Note that the matrices $\widehat{\mathsf{L}}_k$ only differ in the eigenvalue, λ_k, multiplying the diagonal contribution Ω_{2d}.

Therefore, the original system (2.16) is diagonalized into a set of N_{per} mutually independent 2D subsystems

$$\widehat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d} \qquad k = 0, ..., N_{per} - 1, \tag{2.22}$$

where each subsystem, hereafter denoted as *frequency system*, corresponds to a frequency in the Fourier space. In summary, the global algorithm is:

[1]Given matrices A, B, C and D, with appropriate size, $(\mathsf{A} \otimes \mathsf{B}) \cdot (\mathsf{C} \otimes \mathsf{D}) = \mathsf{AC} \otimes \mathsf{BD}$

Algorithm 1:

1. Transform the right-hand-side b, $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})b$
2. Solve the the *frequency systems*, $\widehat{\mathsf{L}}_k \hat{x}^{2d}_k = \hat{b}^{2d}_k$
3. Restore the solution vector: $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})\hat{x}$

At this point, some issues must be addressed:

1. *Fourier decomposition of real-valued, problems.* Since the subvectors b^{per}_k are real-valued, as described in Appendix A, the corresponding discrete Fourier coefficients are paired as follows

$$\left[\hat{b}^{per}_k\right]_i = \left[\hat{b}^{per}_k\right]^*_{N_{per}-i} \qquad i = 1, ..., N_{per} - 1. \tag{2.23}$$

Thus, the 2D subvectors $\hat{b}^{2d}_k$ meet

$$\hat{b}^{2d}_k = (\hat{b}^{2d}_{N_{per}-k})^* \qquad k = 1, ..., N_{per} - 1. \tag{2.24}$$

On the other hand, the eigenvalues of the real-valued sparse matrix L_{per}, defined in equation (2.18), fulfil the following property

$$\begin{aligned} \lambda_0 &= 0, \\ \lambda_k &= \lambda_{N_{per}-k} \qquad k = 1, ..., N_{per} - 1. \end{aligned} \tag{2.25}$$

Hence, plugging the two previous identities into equation (2.21) leads to

$$\begin{aligned} \widehat{\mathsf{L}}_0 &= \Delta_{per}\mathsf{L}_{2d}, \\ \widehat{\mathsf{L}}_k &= \widehat{\mathsf{L}}_{N_{per}-k} \qquad k = 1, ..., N_{per} - 1. \end{aligned} \tag{2.26}$$

Finally, the last equation together with equation (2.24) imply that the solution of the *frequency systems* are paired as follows

$$\hat{x}^{2d}_k = (\hat{x}^{2d}_{N_{per}-k})^* \qquad k = 1, ..., N_{per} - 1. \tag{2.27}$$

Therefore, recalling that N_{per} is an even number, the solution of $N_{per}/2 - 1$ of the *frequency systems* is directly obtained by taking the complex conjugate of its respective pairs.

2. *Complex systems.* The subvectors $\hat{b}_0^{2d}$ and $\hat{b}_{N_{per}/2}^{2d}$ are real-valued whereas the rest of subvectors $\hat{b}_k^{2d}$ have non-null imaginary components [1,2]. This implies that the $N_{per}/2-1$ *paired* systems have a complex-valued solution. Nevertheless, since the coefficients of the matrices $\widehat{\mathsf{L}}_k$ are real, they can be solved as follows:

$$\widehat{\mathsf{L}}_k \left[\ \mathsf{Re}(\hat{x}_k^{2d}) \mid \mathsf{Im}(\hat{x}_k^{2d}) \ \right] = \left[\ \mathsf{Re}(\hat{b}_k^{2d}) \mid \mathsf{Im}(\hat{b}_k^{2d}) \ \right] \quad k = 1, ..., \frac{N_{per}}{2} - 1. \tag{2.28}$$

Therefore, summing up, N_{per} real-valued 2D subsystems need to be solved in total.

3. *Fast Fourier transform.* The inverse and forward Fourier transformations can be carried out by means of a FFT algorithm. This reduces the complexity of steps 1 and 3 of Algorithm 1 from $O((N_{per})^2 N_{2d})$ to $O(N_{per}\log(N_{per})N_{2d})$.

4. *Diagonal dominance of the frequency systems.* Each frequency system is defined as the sum of two components $\Delta_{per}\mathsf{L}_{2d}$ and $\lambda_k \Omega_{2d}$, see equation (2.21). The first component, is a symmetric definite-negative matrix with negative values in its diagonal. The second is a diagonal contribution which varies with the eigenvalue. Since the eigenvalues defined in (2.18) fullfill the next property:

$$0 \leq i < j \leq N_{per}/2 \quad \Rightarrow \quad 0 \geq \lambda_i > \lambda_j, \tag{2.29}$$

the negative contribution $\lambda_k \Omega_{2d}$ decreases when $k \to N_{per}/2$, and the associated system $\widehat{\mathsf{L}}_k$ becomes more diagonal dominant.

2.3.2 Direct Schur-complement based decomposition method (DSD) for the frequency systems

The Schur-complement based algorithms are non-overlapping decomposition methods [8,11,13,16,20], which can be employed for the parallel solution of linear systems. In the present work, this methodology is used to solve the *frequency systems*. However, for the sake of simplicity, a general notation is adopted:

$$Ax = b, \tag{2.30}$$

where $A = (a_{ij})$ is a $n \times n$ sparse, symmetric and positive-definite matrix. The set of unknowns coupled by (2.30) is named $\mathcal{Y}$. In the parallelization this set is partitioned into P_{2d} subsets, $\{\mathcal{Y}_0, ..., \mathcal{Y}_{P_{2d}-1}\}$, where $\mathcal{Y}_k$ are referred to as the *local* unknowns of process k. In order to decouple the system, it is needed an *interface* subset, $\mathcal{S} \subset \mathcal{Y}$, fulfilling the following property:

$$\{i \in \mathcal{Y}_k \cap \mathcal{S}^c, \ j \in \mathcal{Y}_l \cap \mathcal{S}^c \text{ and } k \neq l\} \Longrightarrow \{a_{ij} = a_{ji} = 0\}, \tag{2.31}$$

where $\mathcal{S}^c$ is the complement of $\mathcal{S}$ in $\mathcal{Y}$. That is, two *local* non-*interface* variables of different processes cannot be directly coupled by the system. An example of *interface* is illustrated in Figure 2.3. The subsets $\mathcal{S}_k := \mathcal{Y}_k \cap \mathcal{S}$, and $\mathcal{U}_k := \mathcal{Y}_k \cap \mathcal{S}^c$, are here named the *local interface* and *local inner* unknowns of process k, respectively.

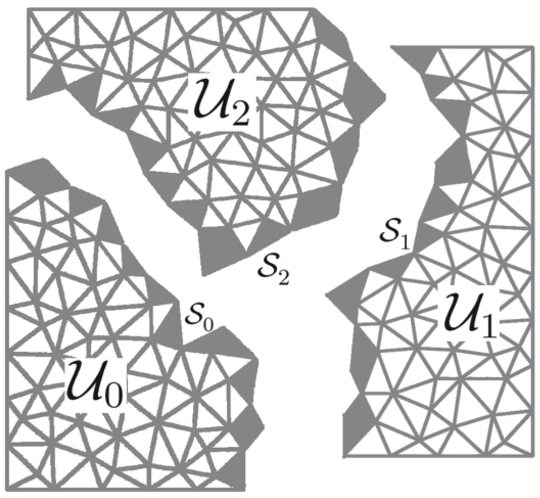


Figure 2.3: Representation of a 'double sided' *interface*. The *interface* nodes are the nodes located in the filled cells $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2$, the remaining nodes form the *local inner* subsets $\mathcal{U}_0$,$\mathcal{U}_1$ and $\mathcal{U}_2$.

Then, labelling the unknowns in the order $\mathcal{U}_0, ...\mathcal{U}_{P_{2d}-1}, \mathcal{S}$, the linear system (2.30) has the following block structure:

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{2.32}$$

where

$$B = \bigoplus_{i=0}^{P_{2d}-1} B_i, \tag{2.33}$$

is a block diagonal matrix, its subblocks $B_i \in \mathbb{R}^{\mathcal{U}_i \times \mathcal{U}_i}$ are the couplings between the ith *local inner* unknowns, E are the couplings between the *inner* and *interface* unknowns, $F = E^T$ are the coupling between the *interface* and the *inner* unknowns and C are the linear couplings between the *interface* variables. Applying a block Gaussian elimination to (2.32), leads to

$$\begin{pmatrix} B & E \\ 0 & \tilde{C} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ \tilde{g} \end{pmatrix}, \tag{2.34}$$

where $\tilde{C} = C - FB^{-1}E$ is the Schur complement matrix, and $\tilde{g} = g - FB^{-1}f$ the new r.h.s term for the *interface* system. Therefore, the whole algorithm is:

Algorithm 2:

1. Evaluate the new r.h.s for the *interface* system $\tilde{g} = g - FB^{-1}f$.
2. Solve the distributed *interface* system $\tilde{C}y = \tilde{g}$.
3. Solve the *local inner* systems, $B_i x_i = f_i - E_i y$, where E_i and f_i are submatrices of E and f, formed by the rows corresponding to nodes in $\mathcal{U}_i$.

Note that, although the *inner* systems with matrices B_i are solved twice (steps 1 and 3), they are mutually independent and can be solved simultaneously. This is the main concept of the Schur complement techniques: to separate, by means of a common distributed *interface*, a subset of the *local* unknowns of each process and solve them independently.

This constitutes a general framework for these type of algorithms. At this point, three critical issues need to be addressed: (i) the determination of the *interface* subset, (ii) the *local* solver for the *inner* systems and (iii) the solver for the *interface* system.

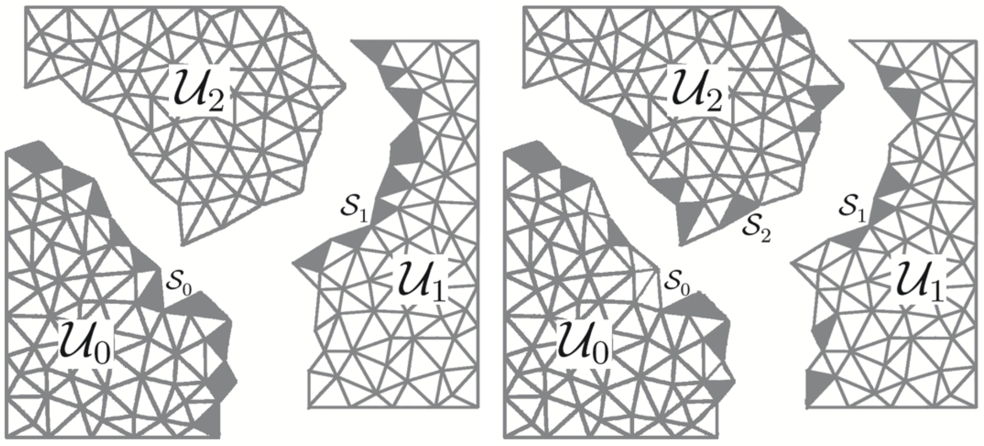


Figure 2.4: Representation of the '*one-sided interface*'. Left: *One-sided interface* before balancing. Right: *One-sided interface* after balancing.

Interface subset

There are different options to determine the *interface* subset. One option used in [16, 17] , herewith denoted *double sided interface*, is to define the *local interface* of each parallel process, $\mathcal{S}_k$, as the *local* unknowns coupled with unknowns of other processes, *i.e.*

$$\mathcal{S}_k = \{i \in \mathcal{Y}_k : St(i) \cap \mathcal{Y}_k{}^c \neq \emptyset\}, \tag{2.35}$$

where $St(i) := \{j \in \mathcal{Y} : a_{ij} \neq 0\}$ is the *stencil* of i. This alternative may be convenient because $\mathcal{S}_k$ can be fixed independently by each process (see Figure 2.3) and, therefore, the structure of the Schur complement matrix is known *a priori* [16]. However, the size of the resulting *interface* $\mathcal{S}$ is not minimal.

In this chapter, in order to reduce the size of the *interface* system, a *one-sided interface* strategy is adopted (see Figure 2.4). To carry this out, firstly the *local interface* subsets $\mathcal{S}'_k$ are defined as

$$\mathcal{S}'_k = \{i \in \mathcal{Y}_k : \exists l > k \text{ with } St(i) \cap \mathcal{Y}_l \neq \emptyset\}. \tag{2.36}$$

That is, given two coupled unknowns of two different processes, only the one which belongs to the process with lower rank is included in the *interface* (see Figure 2.4, left). In this way, the *interface* size is approximately halved. However, it generally results on an unbalanced *interface*. To circumvent this problem, a balancing algorithm is proposed in the next paragraphs.

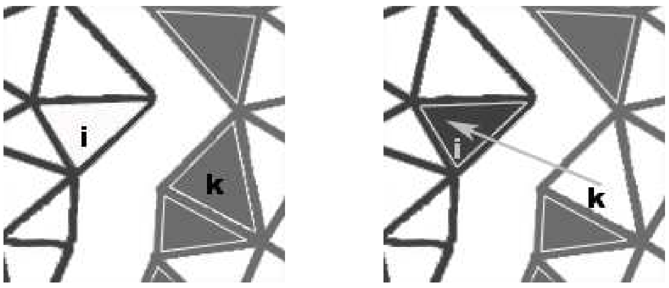


Figure 2.5: Illustration of the balancing process. Node k is *moved* from the *interface* to the *inner* set and, subsequently, its neighbor node i, which belongs to another processor, is *moved* to the interface.

Lets consider two non-intersecting subsets $\mathcal{J}_1$, $\mathcal{J}_2 \subset \mathcal{Y}$, the *halo* of $\mathcal{J}_1$ in $\mathcal{J}_2$ is defined as

$$\mathcal{H}(\mathcal{J}_1, \mathcal{J}_2) := \{k \in \mathcal{J}_2 : \exists s \in \mathcal{J}_1 \text{ with } k \in St(s)\}. \tag{2.37}$$

That is, $\mathcal{H}(\mathcal{J}_1, \mathcal{J}_2)$ is the subset of elements of $\mathcal{J}_2$ which are coupled with elements of $\mathcal{J}_1$. The interface imbalance, $Imb(\mathcal{S})$, is defined as:

$$Imb(\mathcal{S}) = \{max(|\mathcal{S}_i| - |\mathcal{S}_j|),\ \forall\, i, j \in [0, ..P-1]\}, \tag{2.38}$$

where $|\mathcal{S}_k|$ is the number of elements of the set $\mathcal{S}_k$. With this nomenclature, the proposed balancing algorithm reads:

Algorithm 3:

```
1.  while( $Imb(\mathcal{S}) > \mathrm{tol} \cdot |\mathcal{S}|$ )
2.      for all( $i, j \in [0, ..P-1]$ )
3.          if( $|\mathcal{S}_i| > |\mathcal{S}_j|$ )
4.              cont := min( $(|\mathcal{S}_i| - |\mathcal{S}_j|)/2$ , $|\mathcal{H}(\mathcal{Y}_j, \mathcal{S}_i)|$ )
5.              $\mathcal{J} := \emptyset$
6.              while( $\mathrm{cont} > 0$ )
7.                  take $s$ from $\mathcal{H}(\mathcal{Y}_j, \mathcal{S}_i)$
8.                  $\mathcal{S}_i = \mathcal{S}_i - \{s\}$
9.                  $\mathcal{J} = \mathcal{J} \cup \{s\}$
10.                 $\mathrm{cont} = \mathrm{cont} - 1$
11.             endwhile
12.             for all($l \neq i$)
13.                 $\mathcal{S}_l = \mathcal{S}_l \cup \mathcal{H}(\mathcal{J}, \mathcal{Y}_l)$
14.             endfor
15.         endif
16.     endfor
17. endwhile
```

The basic step of the balance algorithm consists on moving a subset of unknowns $\mathcal{J}$ from $\mathcal{S}_i$ to $\mathcal{U}_i$ and, subsequently, all the *inner* unknowns of neighbor subdomains coupled with elements of $\mathcal{J}$ are moved to their *local* interfaces. This process is illustrated in Figure 2.5 and the resulting interface in Figure 2.4 (right). This algorithm is an optimization of the previous version presented in [15], where the balancing process was carried out element by element (*i.e.* considering subsets $\mathcal{J}$ of only one element). The previous approach needed much more steps and communications to converge to a balanced interface. Figure 2.6 shows the imbalance reduction achieved with the new version of the algorithm for different 2D meshes partitioned into 20 subdomains. In these tests, the algorithm has been runned until the imbalance could no longer be reduced. For all the meshes the final imbalance is lower than 0.1% except for the mesh m250m which finishes with a 0.7% of imbalance.

Solution of inner and interface systems

The set of *inner* systems are solved by means of a sparse Cholesky factorization [22]. The *interface* system, which couples unknowns of different processes, requires a parallel algorithm. To solve it, an explicit evaluation of $\tilde{C}^{-1}$, where the kth process evaluates $\tilde{C}_k^{-1}$ (the rows in $\mathcal{S}_k$), is performed. Each process calculates the sparse

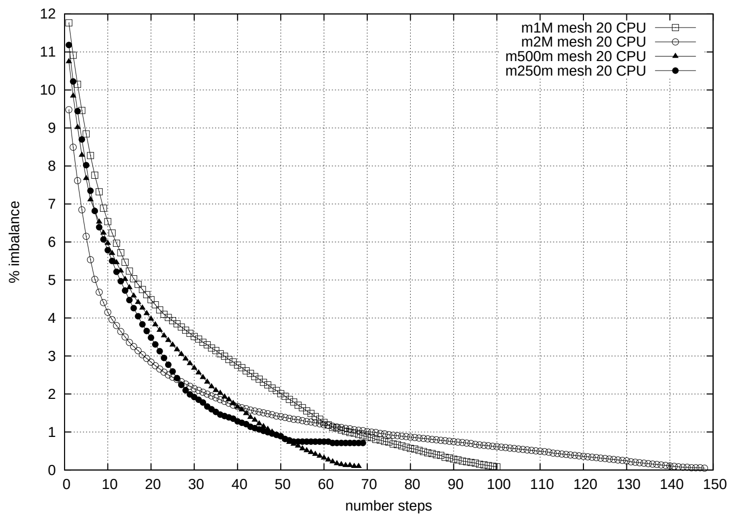


Figure 2.6: Imbalance reduction on each iteration of the main loop of the Algorithm 3, for the different unstructured 2D meshes listed in Table 1 partitioned into 20 subdomains.

Cholesky factorization of $\tilde{C}$ and evaluates the rows of $\tilde{C}^{-1}$ corresponding to its local interface. The reasons to use $\tilde{C}^{-1}$ are twofold: (i) a direct solution of the *interface* system is obtained, and (ii) since the solution phase is a matrix-vector product its parallelization is straightforward. However, this methodology is only feasible when (like in 2D problems) the size of the *interface* is much smaller than the total size of the system. Note that since both methods are direct solvers, the global algorithm is a direct solver as well. The preprocessing phase of the chosen algorithms can have a significant computational cost. However, since the applications here considered demand a huge number of time-steps, and the system matrix is constant during all the simulation, this additional cost becomes, in general, negligible.

Therefore, the detailed algorithm of the DSD solution phase is:

```
Algorithm 4: .
  1. Evaluate the local r.h.s for the interface system $\tilde{g}_i$:
      (a) Solve $B_i t_i = f_i$.
      (b) Get necessary components of $t$ before product by $F_i$ (point-to-point comm.).
      (c) $\tilde{g}_i = g_i - F_i t$.
  2. Solve local interface unknowns $y_i$.
      (a) Obtain all components of $\tilde{g}$ (collective comm.).
      (b) $y_i = \tilde{C}_i^{-1} \tilde{g}$.
  3. Evaluate r.h.s for the local inner systems.
      (a) Get necessary components of $y$ before product by $E_i$ (point-to-point comm.).
      (b) $h_i = f_i - E_i y$.
  4. Solve the inner systems $B_i x_i = h_i$.
```

Note that the vectors t and h are temporary storage data. F_i, C_i, y_i and $\tilde{g}_i$ are the submatrices of F, C, y and $\tilde{g}$ corresponding to the nodes in $\mathcal{S}_i$. Three communication episodes, performed by means of the MPI standard, are needed: two point-to-point communications before sparse matrix-vector products in steps 1.b and 3.a, and a collective communication, performed by means of the MPI_Allgather function, before the product by the dense matrix $\tilde{C}_i^{-1}$ in step 2.a. These communications, together with the solution of the *interface* system (step 2.b), are the parts of the algorithm whose cost increases with the number of CPUs. Therefore, these steps, further referred to as *interface computations*, eventually degrade the speed-up of the global algorithm. The rest of the steps, referred to as *inner computations*, tend to accelerate with the number of CPUs.

2.4 Distribution of parallel processes

The parallelization of the solver is based on a geometric domain decomposition into P subdomains, one for each parallel process. The partition of $\mathcal{M}$ is carried out by dividing $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$ into P_{2d} and P_{per} parts respectively, being $P = P_{2d}P_{per}$. This is referred as a $P_{2d} \times P_{per}$-partition. To exemplify it, a 2×2- and a 4×1-partitions of a mesh are displayed in Figure 2.7.

The parallelization of Algorithm 1 can be divided into two parts: (i) the parallelization of steps 1 and 3, which are the change-of-basis from the *physical* to the *spectral* space and vice versa; and (ii) the parallelization of step 2, which is the solution of the *frequency systems*.

Looking at step 1, the r.h.s of the *frequency systems*, $\hat{b}$, is given by

$$\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})b. \tag{2.39}$$

Actually, these are N_{2d} mutually independent Fourier transformations, one for each *span-wise* subvector of b. Since a distributed memory parallelization for the FFT is out of consideration, the *span-wise* component of the mesh is not partitioned. Thus, $\mathcal{M}_{2d}$ is divided into P subdomains and a $P \times 1$-partition of $\mathcal{M}$ follows. In this way a sequential FFT algorithm can be directly applied to each *span-wise* subvector, b_k^{per}, respectively. An identical reasoning is applied to the change-of-basis from the *spectral* to the *physical* space, and the same $P \times 1$-partition is chosen.

On the other hand, in step 2 of the Algorithm 1, the solution of the *frequency systems* (2.28) is obtained as follows

$$\hat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d} \qquad k = 0, ..., \frac{N_{per}}{2}, \tag{2.40}$$

where the DSD algorithm is adopted to solve each system. In this case, for large values of P, the $P \times 1$-partition chosen for the steps 1 and 3 can be sub-optimal as the strong speed-up of the DSD algorithm is limited (see Section 2.5.1). Thus, partitions with $P_{per} > 1$ may be necessary to keep P_{2d} in the region of linear scalability of the DSD algorithm. In this case, the N_{per} frequencies to be solved are divided into P_{per} subsets, and groups of P/P_{per} processes are used to solve the frequencies of each subset. When partitioning $\mathcal{M}_{per}$, it should be taken into account that the *frequency systems* are paired (see Section 2.3.1), keeping the *paired* subsystems in the same subdomain.

Therefore, in order to achieve the maximum parallel performance for each part of the algorithm, two different partitions are used: one for the change-of-basis (steps 1 and 3) and another for the solution of the *frequency systems* (step 2). With this new strategy, two additional redistributions of data between those partitions are needed. The following algorithm replaces Algorithm 1:

Algorithm 5:

1. Evaluate $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}})b$ on the $P \times 1$-partition.
2. Redistribute $\hat{b}$ from the $P \times 1$-to the $P_{2d} \times P_{per}$-partition.
3. Solve the the *frequency systems*, $\hat{\mathsf{L}}_k \hat{x}^{2d}_k = \hat{b}^{2d}_k$, on the $P_{2d} \times P_{per}$-partition.
4. Redistribute $\hat{x}$ from the $P_{2d} \times P_{per}$-to the $P \times 1$-partition.
5. Evaluate $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}})\hat{x}$ on the $P \times 1$-partition.

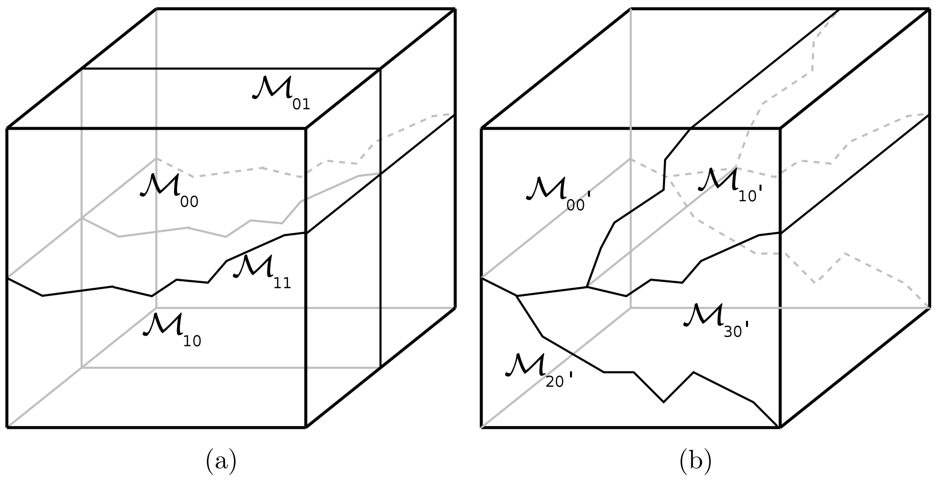


Figure 2.7: Illustration of a 2×2 (right) and a 4×1 (left) partitions of a mesh.

In order to simplify the redistributions of data (steps 2 and 4), a multilevel partition strategy is used. The P-partition of $\mathcal{M}_{2d}$, used in steps 1 and 5, is obtained from the P_{2d}-partition used in the step 3 by dividing each of its subdomains in P_{per} parts. An example is shown in Figure 2.7: the 2D subdomains in the right part, are directly obtained by splitting the 2D subdomains in the left. As a result, in this example, when redistributing the data between these two partitions, two independent transmissions are done involving the subdomains $\mathcal{M}_{00}, \mathcal{M}_{01}$ and $\mathcal{M}_{00'}, \mathcal{M}_{10'}$ on the one hand, and the subdomains $\mathcal{M}_{10}, \mathcal{M}_{11}$ and $\mathcal{M}_{20'}, \mathcal{M}_{30'}$ on the other. In the general case, on each redistribution of data, each parallel process is involved in a collective communication with P_{per} processors. These collective communications are performed by means of the MPI_Alltoall routine.

In Algorithm 5, increasing P_{per} has two counteracting effects: it tends to increase the computational cost of steps 2 and 4, as each MPI_Alltoall communication would involve more parallel processes, whereas it benefits the speed-up of the step 3. Therefore, for each problem and computational architectute an optimal P_{per} needs to be found.

2.5 Numerical experiments

All the numerical tests presented in this chapter have been carried out on the MareNostrum supercomputer of the Barcelona supercomputing center (BSC). This is an IBM BladeCenter JS21 Cluster with 10240 PowerPC 970MP processors at 2.3 GHz. Quad-core nodes with 8 Gb are coupled by mean of a high-performance Myrinet network.

The code has been compiled in a Linux SuSe distribution using the IBM XL C/C++ enterprise edition compiler version 10.1. Four external libraries are linked: (i) the automatically tuned linear algebra software (ATLAS) [28] version 3.5.1, to perform the product by the inverse of the Schur complement matrix; (ii) the FFTW [29] version 3.1.1, to perform the Fourier-based change of basis between the physical and spectral spaces; (iii) the METIS [30] software to compute fill-reducing orderings for the Sparse Cholesky factorizations; and (iv) the MPICH implementation of MPI, the standard message-passing interface library, version mx.

Results have been obtained after averaging over several time-steps, to avoid dispersion. To test the performance of the solver, four different unstructured 2D meshes have been generated with the ANSYS ICEM CFD package [31] (see Table 2.1 for details). They cover a square domain with side length 1 using triangular elements. Their partition is carried out by means of the graph partitioning tool METIS [30].

In a multi-core architecture, the performance of the CPU cores can be affected by the shared memory bandwidth resulting in lower performance of each core when all CPU cores are engaged [32]. To circumvent this variable, all the tests have been performed by using all the cores within each quad-core processor. Consequently, the number of MPI processes is restricted to multiples of 4.

Name	size
m250m	255,468
m500m	500,590
m1M	1,028,350
m2M	2,094,974

Table 2.1: Name and size of the 2D unstructured meshes used on the numerical experiments. The size refers to the number of nodes of the mesh.

2.5.1 Strong speed-up of DSD algorithm

The *strong* speed-up measures the acceleration of the algorithm with the number of CPUs. As the DSD is a direct solver, and the *frequency systems* are defined by symmetric negative definite and diagonal dominant matrices, its performance only depends on their sparsity pattern. Thus, recalling that the family of *frequency systems* to be solved (2.22) only differ on the diagonal elements, the analysis can be restricted to the first *frequency system* $\widehat{\mathsf{L}_0} = \mathsf{L}_{2d}$.

Speed-up results starting from 4 and up to 100 CPUs are displayed in Figure 2.8. All the meshes show the same qualitative behavior. Nevertheless, as expected, the speed-up improves with the size of the problem. Such effect is due to the growth of the percentage of time spent in the *inner computations* (see Section 2.3.2) with the problem size (see Figure 2.8, bottom).

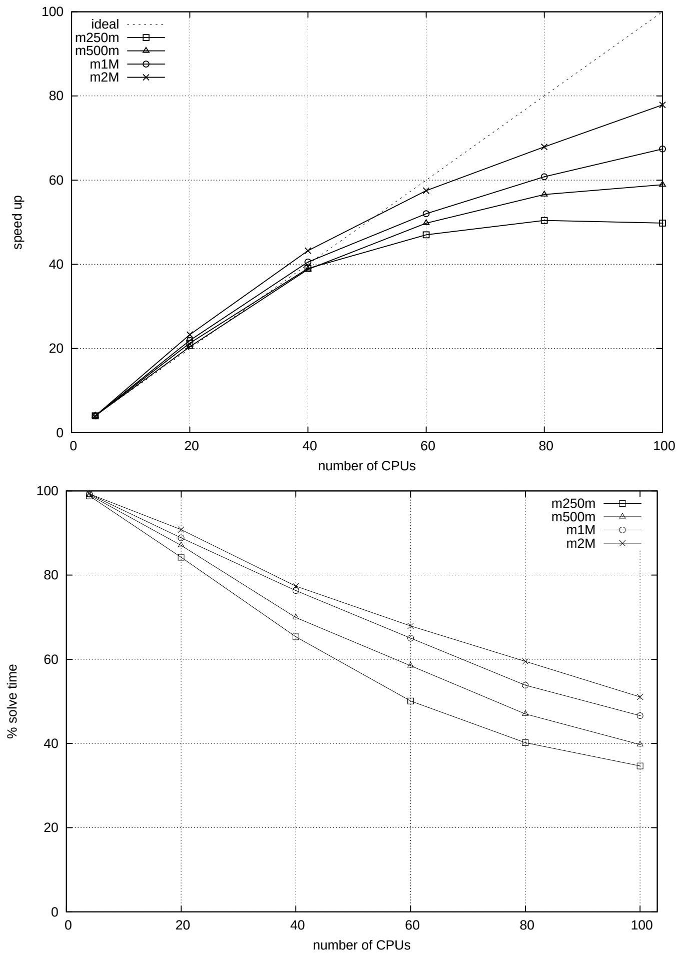


Figure 2.8: Top: *Strong* speed-up of DSD solution phase measured in meshes of different sizes. Bottom: Percentage of time spent in *inner computations* for the different meshes and numbers of CPU.

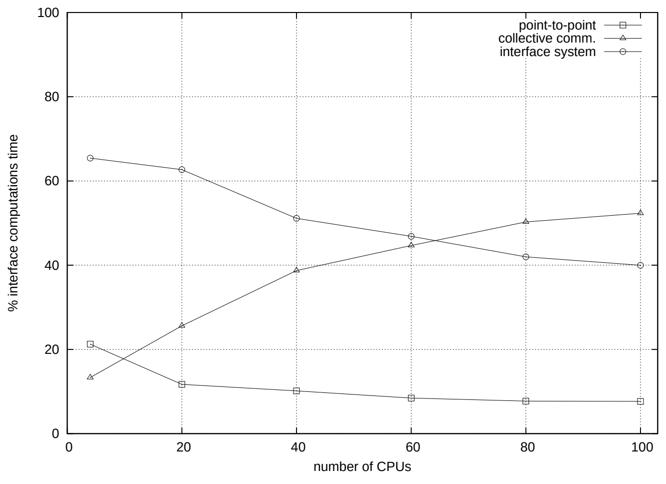
For each problem, the scalability of the DSD algorithm is eventually limited by the decrease of the percentage of time spent in *inner computations* and, consequently, the growth of the *interface computations*. Recalling Algorithm 4, the *interface computations* are: (i) the point-to-point communications in the steps 1.b and 3.a, (ii) the collective communication in step 2.a, performed by means of the MPI_Allgather function, and (iii) the product by the inverse of the Schur complement matrix in step 2.b. The relative contribution of each of these parts within the total *interface computations* is shown in Figure 2.9. For the sake of simplicity, only the results corresponding to the two limiting meshes, *i.e.* m250m and m2M, are analyzed. The product by the inverse of the Schur complement system and the collective communication are by far the most time consuming parts, however, their tendencies are opposite. On the one hand, the size of the *interface* is $O(\sqrt{P_{2d}N_{2d}})$. Thus the operation count of the dense product is $O(P_{2d}N_{2d})$ and the operations per CPU are $O(N_{2d})$. Therefore, the cost per CPU remains constant when increasing the number of CPUs. In contrast, under certain approximations, the collective communication cost per CPU is $O(log_2(P)\sqrt{P_{2d}N_{2d}})$ (for details see [11]). This explains the opposite tendencies of these two parts of the algorithm when increasing the number of CPU, and why the percentage of time spent in the dense product grows with the size of the mesh. Finally, the percentage of the point-to-point communications remains almost negligible in both cases.

2.5.2 Weak speed-up test in the periodic direction

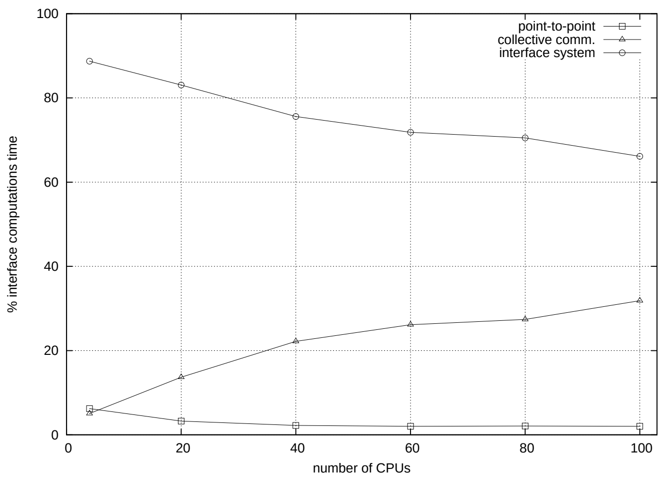
The *weak* speed-up shows the scalability of the algorithm when the ratio between the mesh size and the number of CPUs remains constant. Ideally the solution time should also remain constant, however, the local cost of the communications and the algorithm tend to grow with the number of processors resulting in a slow down.

In Figure 2.10, the weak scalability when the mesh grows in the periodic direction is displayed. The discretizations are generated by extrusion of the 2D meshes m250m, m500m, m1M and m2M, respectively. The initial tests have been performed using 40 CPUs and setting $N_{per} = 8$, for the meshes m250m, m500m and m1M; and $N_{per} = 4$ for the mesh m2M. Thus, the loads per CPU are: 50000, 100000, 200000 and 200000 nodes, respectively. P_{2d} is kept constant equal to 40, which is inside the DSD linear scalability region for all cases (see Figure 2.8). Therefore, the parallel efficiency of the DSD component will always be close to one. Then, P_{per} is successively increased from 1 ($P = 40$) to 24 ($P = 960$) keeping the ratio N_{per}/P_{per} constant.

The results show that, despite the size of the mesh and the number of CPUs increase 24 times, the solution time grows a factor between 1.41 and 1.53, depending on the 2D mesh size. In all cases, when $P_{per} = 1$, approximately 80% of the time is spent in solving the *frequency systems*, while the rest of time is spent in the changes of basis from the physical to the spectral space and *vice versa*. The cost of the first part remains constant, while keeping P_{2d} and the ratio N_{per}/P_{per} constant. However, when increasing P_{per} and N_{per}, the cost due to the MPI_Alltoall communications



(a)

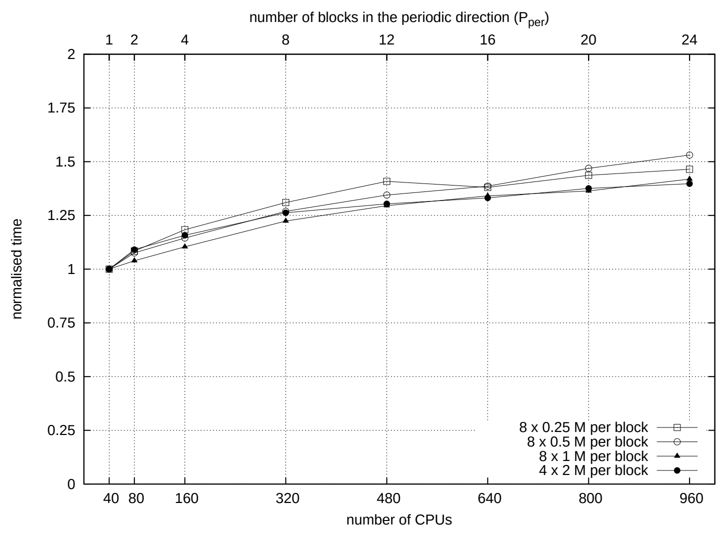


(b)

Figure 2.9: Percentage of time spent in the different parts of the *interface computations* for the meshes m250m (left) and m2M (right).

(steps 2 and 4 of Algorithm 5) and the Fourier transforms (steps 1 and 5) also increases. As a consequence, for $P_{per} = 24$, the maximal value tested, the cost of performing the two changes-of-basis already represents approximately 45% of the total solution time. In Table 2.2, the wall clock times for the solution and the setup phases, together with the size of the problems, are shown. It can be seen that the setup time is almost constant.

Finally, a large-scale study with up to 8192 CPU is shown in Figure 2.11. The meshes for this test have been generated using m2M as 2D component. For the biggest one, the total number of grid points is 1024 million. P_{2d} is fixed equal to



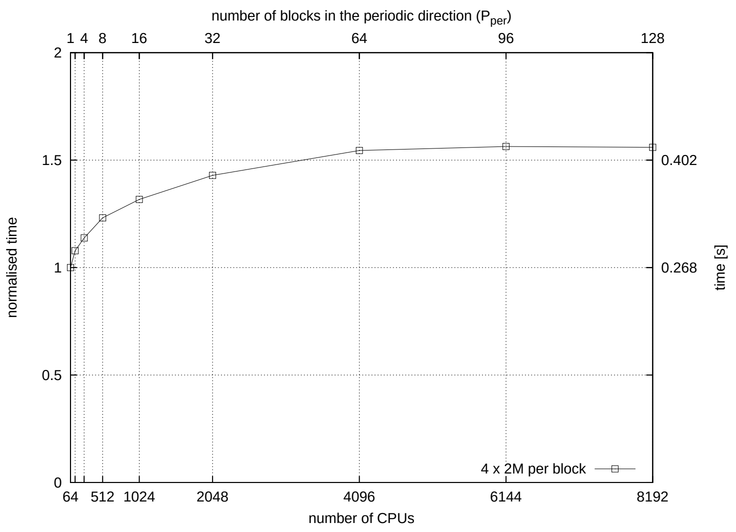
(a)

Figure 2.10: *Weak* speed-up in the periodic direction for different extruded 2D unstructured meshes. The loads per CPU are: 50000, 100000, 200000 and 200000 nodes for each of the tests, respectively. P_{2d} is kept constant equal to 40 and P_{per} is successively increased from 1 ($P = 40$) to 24 ($P = 960$).

CPUs		40	80	160	320	480	640	800	960
P_z		1	2	4	8	12	16	20	24
m250m	solve	0.071	0.077	0.084	0.093	0.100	0.098	0.102	0.104
	setup	89	94	87	87	87	90	91	91
	size	2	4	8	16	24	32	40	48
m500m	solve	0.145	0.156	0.166	0.184	0.195	0.201	0.213	0.222
	setup	194	193	193	188	188	199	191	203
	size	4	8	16	32	48	64	80	96
m1M	solve	0.308	0.320	0.340	0.377	0.399	0.413	0.420	0.437
	setup	558	559	557	552	561	546	553	574
	size	8	16	32	64	96	128	160	192
m2M	solve	0.363	0.395	0.419	0.457	0.472	0.482	0.498	0.506
	setup	1094	1073	1063	1069	1062	1065	1078	1089
	size	8	16	32	64	96	128	160	192

Table 2.2: Solution and setup times, in seconds, and size of the mesh, in million of nodes, for the points in Figure 2.10.

64, which is in the limit of the linear speedup region of the DSD algorithm for this mesh (see Figure 2.8). Therefore, the parallel efficiency of the DSD component will be close to one. Initially N_{per} and P_{per} are 8 and 1, respectively. Thus the load per CPU is approximately 125000 nodes. As shown in the figure, P_{per} and N_{per} are increased 128 times, while the solution time only grows 1.5 times. The size of the problem varies from 8 to 1024 million nodes, and the wall clock time spent in the solution from 0.27 to 0.42 s. The setup time is less than 30 min in all cases. In practice, for time-accurate simulations on such meshes, this time is almost negligible compared with the expected accumulated solution time.



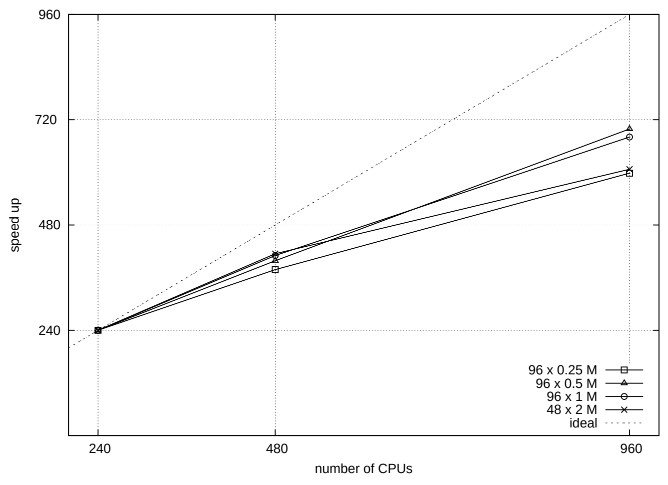
(a)

Figure 2.11: *Weak* speed-up in the periodic direction for meshes generated by the extrusion of the mesh m2M, the load per CPU is kept around 125000 nodes. The size of the problem (and the wall-clock time) varies from 8 million (0.27s) to 1024 million (0.42s), respectively.

2.5.3 Strong speed-up tests for the overall algorithm: a demonstrative example

In the previous sub-sections, the two components of the solver have been studied separately, and for the DSD, on an isolated *frequency system*. The acceleration produced by the combination of both components within the overall algorithm is now considered. The acceleration of the overall algorithm is produced by the acceleration of the DSD and by the reduction of *frequency systems* to be solved per parallel process (N_{per}/P_{per}), when P_{2d} and P_{per} are increased, respectively. Figure 2.12 shows these two acceleration factors. The discretization meshes are generated from the 2D meshes of Table 3.1, and setting N_{per} equal to 128 in all cases, except for m2M for which N_{per} is set equal to 64 due to the memory constraints. Initially, P_{2d} and P_{per} are 20 and 12, respectively. In the first step, P_{2d} is doubled ($P_{2d} = 40$, $P_{per} = 12$)

and, in accordance with subsection 2.5.1, the speed up is better when larger the 2D component of the mesh. The parallel efficiency of this step ranges between 0.79 and 0.86 for the different cases. In the second step P_{per} is doubled ($P_{2d} = 40$, $P_{per} = 24$), and the acceleration no longer depends on the 2D component size. For this step the parallel efficiency ranges between 0.73 and 0.88. The overall parallel efficiency is the product of the two previous ones.



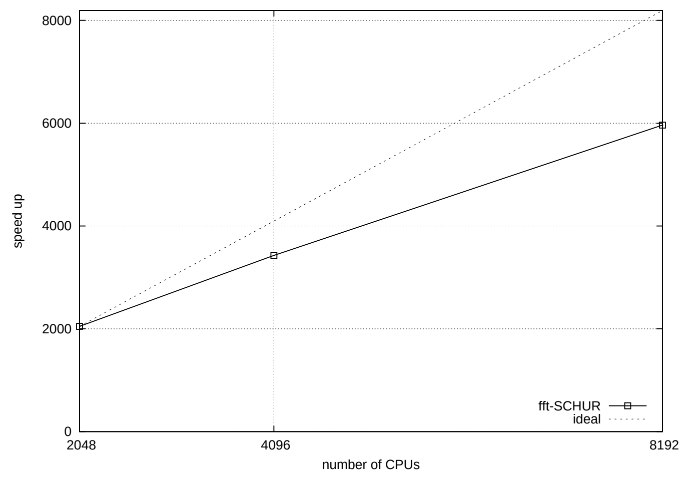
(a)

Figure 2.12: *Strong* speedup of the overall algorithm for different meshes. On the first step P_{2d} is doubled from 20 to 40 and on the second P_{per} is doubled from 12 to 24.

Finally, the same study was carried out for a mesh of 512 million nodes generated from the extrusion of m2M ($N_{per} = 256$), see Figure 2.13. Initially $P = 2048$ ($P_{2d} = 32$, $P_{per} = 64$), when doubling P_{2d} and P_{per} (up to 8192 CPUs) the parallel efficiencies obtained are 0.84 and 0.87, respectively. The walk clock time spent in the solution varies from 0.69 to 0.24 s.

2.6 Challenging DSD. Flow around a circular cylinder

In this section, the performance of the DSD as frequency solver is analyzed by direct comparison with the standard preconditioned conjugate gradient (PCG) method [7, 8] with two different preconditioners. To carry out this analysis, direct numerical simulations of the flow around a circular cylinder at Reynolds numbers 3900 and 10000 (based on the cylinder diameter, D, and the free-stream velocity), are used as problem models.



(a)

Figure 2.13: *Strong* speedup of the overall algorithm for a 512 million nodes mesh generated from the extrusion of m2M. On the first step P_{2d} is doubled from 32 to 64 and on the second P_{per} is doubled from 64 to 128.

2.6.1 Flow around a circular cylinder

The dimensions of the computational domain are $[-4D, 20D]$ (*stream-wise*), $[-8D, 8D]$ (*cross-stream*) and $[0, \pi D]$ (*span-wise*), respectively. The axis of the cylinder is located at $x = y = 0$. Periodic boundary conditions are assumed in the *span-wise* direction; symmetry boundary conditions in the *cross-stream*; at the inflow a constant velocity profile, $u = 1$, $v = w = 0$, is prescribed; and pressure-based boundary conditions are used at the outflow. Finally, non-slip boundary conditions are imposed at the surface of the cylinder. Moreover, a buffer zone with a higher artificial viscosity is prescribed at the exit in order to prevent instabilities.

To cover each *stream-wise/cross-stream* plane an unstructured grid composed of triangular elements is used. The number of control volumes of the 2D meshes are $43,445$ ($Re = 3900$) and $154,070$ ($Re = 10000$), respectively. Both 2D meshes are solved with two different values of N_{per}, 64 and 128, in order to study the influence of the *span-wise* direction. The results of the simulations have shown good agreement with previous studies available in the literature [33–35]. An illustrative snapshot showing the near wake vortex structures at $Re = 3900$ is displayed in Figure 2.14.

2.6.2 DSD vs (block) Jacobi PCG

Comparison of solvers is always a difficult task. Since multiple factors may influence their performance, they must be tested in the context of one specific application. This point becomes even more important when comparing direct and iterative solvers. In general, the latter class is strongly dependant on the condition

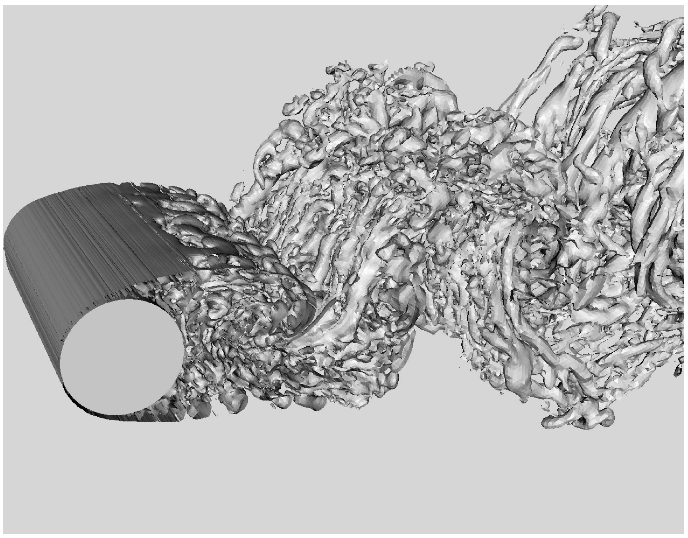


Figure 2.14: Near wake vortex structures of the flow around a circular cylinder at $Re = 3900$.

number of the system, the initial guess and the residual criterion, whereas the direct methods performance is irrespective of these factors.

Here, the standard iterative solvers chosen to compare with are the Jacobi and a block-Jacobi Preconditioned CG (JPCG and bJPCG) [8]. Although Jacobi-type are not the best preconditioning methods available, they have been chosen because their simplicity makes the performance of the whole algorithm dependant only on basic optimized sub-routines, such as matrix-vector products and global norm evaluations, which are the same as those used for the DSD algorithm. To this end, for the bJPCG the size of the blocks correspond to the 2D subdomains and the same sparse Cholesky factorization is used. The comparison with other iterative solvers could be performed using the standard JPCG as a reference.

The comparison of the Poisson solvers in the different above-mentioned situations was carried out on a statistically stationary regime of the flow, and the measurements were averaged over 1000 time-steps. The residual criterion for the iterative solvers was fixed to 10^{-5} and the solution of the previous time-step was used as initial guess. As the time-integration scheme is fully explicit (see Section 2.2.3) a CFL criterion was used to determine the time-step, Δt, keeping the simulation stable.

The number of iterations necessary to reach the prescribed level of accuracy by the JPCG solver as a function of the relative number of frequency, defined as

$$\xi(i, N_{per}) = \frac{2i}{N_{per}} \qquad i = 0, ..., \frac{N_{per}}{2}, \tag{2.41}$$

is displayed in Figure 2.15. It must be recalled that the number of iterations plotted are the average between the iterations necessary to solve the real and complex parts

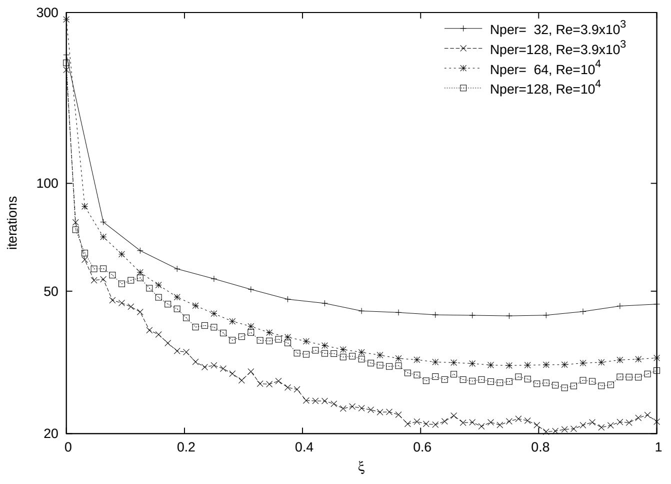
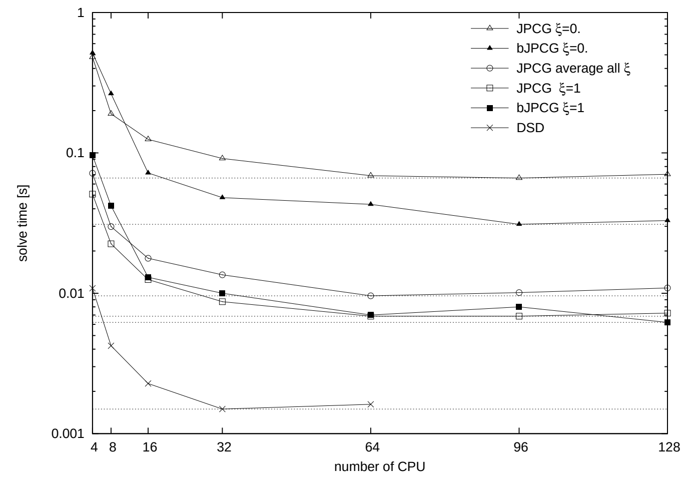


Figure 2.15: Number of iterations needed by JPCG depending on the relative number of frequency for the different situations under study.

of each *frequency system* (2.28). As expected, all the cases show the same qualitative behavior: since the conditioning of the systems improves with ξ_i (see Section 2.3.1), the number of iterations does the same.

The computing times as a function of the number of CPUs, are depicted in Figure 2.16. For the iterative solvers, the two limiting situations, *i.e.*, $\xi = 0$ (maximum number of iterations) and $\xi = 1$ (minimum) are plotted. Additionally, for the JPCG the average time of all the $N_{per}/2 + 1$ *frequency systems* is also shown. Notice that the performance of the DSD is the same for all the frequencies and therefore, only one line is depicted. For the sake of simplicity, this study is carried out only for the most favorable situation for the iterative methods, *i.e.*, the discretizations with $N_{per} = 128$ (see Figure 2.15). At first sight, it can be seen that despite the acceleration of the JPCG and the bJPCG extends to higher numbers of CPUs, they do not reach, by far, the computing times obtained with the DSD solver. Actually, even for the highest frequency system ($\xi = 1$), the DSD algorithm clearly outperforms the rest. The minimum time obtained with each method, together with the number of CPUs and the number of iterations, are shown in Table 2.3. For the lowest frequency ($\xi = 0$), increasing the complexity of the preconditioner (*i.e.*, solving local blocks instead of just the diagonal) the solution time reduces by 53% and 37%, for Reynolds numbers 3900 and 10000, respectively. On the other hand, the highest frequency ($\xi = 1$) has a much more diagonal dominant system, what benefits the convergence of the JPCG. In particular, in our test cases, with $N_{per} = 128$, the value of the diagonal is on average 6 times greater than the sum of the remaining entries in each row for the $Re = 3900$, and 4 times greater for the $Re = 10000$. In this context, the JPCG becomes a much more competitive option and therefore,



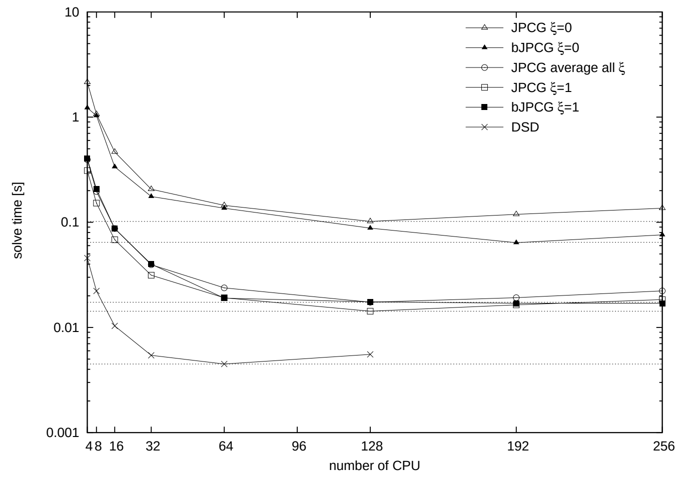


Figure 2.16: Comparison of the DSD, the JPCG and the bJPCG as frequency solvers. Solution times for different *frequency systems* as a function of the number of CPUs used. Top: *Re*= 3900, $\mathcal{M} = 128 \times 43445$ nodes. Bottom: *Re*= 10000, $\mathcal{M} = 128 \times 154070$ nodes.

the margin for improvement reduces. Notice that, although the bJPCG reduces the number of iterations, it does not outperform the JPCG.

On the other hand, recalling that the number of iterations required by the JPCG method does not change with the number of CPUs, it can be stated that for the JPCG the maximum iteration speed is obtainend with 96 and 128 CPUs, for the *Re* 3900 and 10000, respectively. Using this minimal iteration time as a reference, the DSD solve time is equivalent to 4.3 and 9.5 JPCG iterations, for the 2D meshes used on the *Re* 3900 and 10000, respectively. Note that, the DSD cost (in fastest JPCG iterations) increases with the size of the mesh, but also the complexity of the system to be solved iteratively. Normally reducing the number of CG iterations by

	$Re = 3900$			$Re = 10000$		
	time	CPUs	iters	time	CPUs	iters
JPCG $\xi = 0$	6.6×10^{-2}	96	207	1.0×10^{-1}	128	217
JPCG $\xi = 1$	6.8×10^{-3}	96	21	1.4×10^{-2}	128	30
bJPCG $\xi = 0$	3.1×10^{-2}	96	74	6.4×10^{-2}	192	82
bJPCG $\xi = 1$	6.2×10^{-3}	128	15	1.7×10^{-2}	192	15
DSD	1.4×10^{-3}	32	1	4.4×10^{-3}	64	1

Table 2.3: Minimal solution time, together with the number of CPUs and iterations required, for the different methods under study, at the $Re = 3900$ and $Re = 10000$.

means of a more complex preconditioner makes the cost per iteration higher. Thus, to outperform the DSD, these limits of 4.5 and 9.5 iterations should be reduced, but keeping the cost per iteration low enough so that the total solve time is also reduced .

2.7 Illustrative applications

The parallel algorithm developed has been successfully used for solving different turbulent flows which involve one periodic homogeneous direction. Hereafter, some of these cases will be briefly commented. Note that this is not an exhaustive compendium of all the cases solved since now with this methodology, but a selection of cases which have in common that all of them are direct numerical simulations of flows around bluff bodies. Flows of this kind are of great interest for a large number of engineering applications such as vehicle aerodynamics, wings at high angle of attack, interaction of the wind with buildings or cooling devices using forced convection. Prediction of flows which exhibit massive separation, such as those mentioned before, remains nowadays one of the principal challenges to the computational fluids dynamics.

The cases here summarized are: i) the flow past a sphere at sub-critical Reynolds numbers, ii) the flow past a circular cylinder at Reynolds number 3900 and iii) the flow past a NACA 0012 at Reynolds number 50000 with different angles of attack (including the full stall state).

2.7.1 Flow past a sphere at subcritical Reynolds numbers

The unsteady flow around a sphere at sub-critical Reynolds numbers ($Re < 3 \times 10^5$) has a complex nature characterized by the transition from laminar to turbulent flow in the detached shear layer, the existence of a turbulent wake behind the sphere and the unsteady shedding of vortices in the wake. In this case, the direct numerical

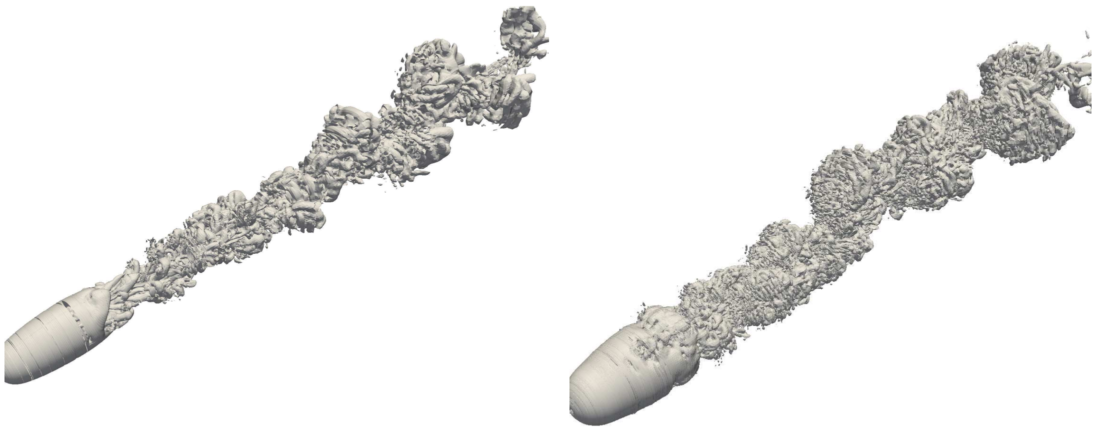


Figure 2.17: Visualization of instantaneous vortical structures in the wake of the sphere by means of Q-iso-surfaces. a) Re=3700. b) Re=10000

simulations (DNS) of the flow at $Re = 3700$ and $Re = 10000$ was carried out. It was the aim of this work to investigate the characteristics of the wake of the sphere at these two Reynolds numbers, in order to provide an insight of the instantaneous and time-average near-wake flow, as well as an spectral analsys of it. The computational domain simulated was a cylindrical domain and the meshes used were generated by the constant-step revolution, in the azimuthal direction, of two-dimensional (2D) unstructured grids.Meshes computed were of 9.48 MCV (million control volumes) and 18.2 MCV, for $Re = 3700$ and $Re = 10000$, respectively. For solving both cases, the JFF cluster and the MareNostrum Supercomputer were used, requiring up to 240 CPUs. In figure 2.17, instantaneous plots of the turbulent wake at both Reynolds numbers are depicted. More details can be found in [36, 37].

2.7.2 Flow past a circular cylinder at Reynolds number of 3900

Similar to the flow past a sphere, the flow around a circular cylinder exhibits different behaviors depending on the Reynolds number. At $Re = 3900$, the flow is subcritical, i.e. it separates laminarly from the cylinder surface and turbulence transition occurs in the separated shear layers. Although there have been numerous investigations about this flow, there are many unanswered questions about the unsteady behavior of the vortex formation region and how this affects the average turbulent statistics in the near wake. Indeed, there is a large scattering in the mean flow solutions in the near wake reported in the literature, which seems to converge into the same solution as the flow moves downstream. The main focus of this work is on the vortex formation region modulation, to examine its possible influence on the wake configuration, as well as, to derive more time-accurate flow parameters and first and second-order statistics. Similar to the mesh generation process for the sphere domain. the meshes computed were obtained by the constant step extrusion in the homogeneous direction of a 2D unstructured grid. Grids with up to 20 MCV were solved using up to 128 CPUs. In figure 2.18, some illustrative streamlines and spanwise vorticity contours are represented. More details about this case can be found in [38–40].

2.7.3 Flow past a NACA 0012 at Reynolds number of 50000 and different angles of attack

In the case of flow around airfoils, transition to turbulence undergoes in the initially laminar shear layer. After separation, at low angles-of-attack (AoA), the flow reattaches to the airfoil surface forming a bubble (known as laminar separation bubble, LSB), which directly affects the airfoil efficiency. At a certain AoA, the flow fails to reattach yielding to a complete separation (stall condition). Thus, the study of the separation mechanism and the correct prediction of boundary layer transition are both key aspects for improving engineering designs. In the case of NACA 0012 airfoil, it exhibits combined leading-edge/trailing-edge stall at moderate Reynolds numbers showing the presence of a turbulent boundary layer separation moving forward from the trailing-edge and a small laminar bubble in the leading-edge region failing to reattach which complete the flow breakdown. In order to gain insight in the stall mechanism, DNS of a NACA 0012 profile at Reynolds number of $Re = 5 \times 10^4$ and at AoA=5, 8, 9.25, 12 were performed. It should be noted that the last two angles corresponded with full stall situations. This study aimed at: i) advancing in the understanding of the physics of turbulent flows and in particular to gain insight in the mechanism of the shear-layer transition, the dynamics of the laminar separation bubble and in the behavior of airfoils in full-stall conditions and ii) developing a detailed database with a complete set of turbulence statistics to be used for the study of flows with massive separation and transitional flows. Meshes required for solving these complex phenomena ranged between 26 and 50 MCV and up to 256 CPUs were used. For solving these cases MareNostrum Supercomputer at Barcelona Supercomputing Center, Magerit CesVima at Universidad Politécnica de Madrid and JFF cluster at CTTC were used (see Appendix B). Instantaneous illustrative results of the flow in the separated region for the AoAs under study are depicted in figure 2.19. For more details, see [41, 42].

2.8 Concluding remarks

A parallel direct algorithm for the solution of the Poisson equation arising in incompressible flows with one periodic direction has been presented. It is a combination of a direct Schur-complement based decomposition (DSD) and a Fourier diagonalization. The latter decomposes the original system into a set of mutually

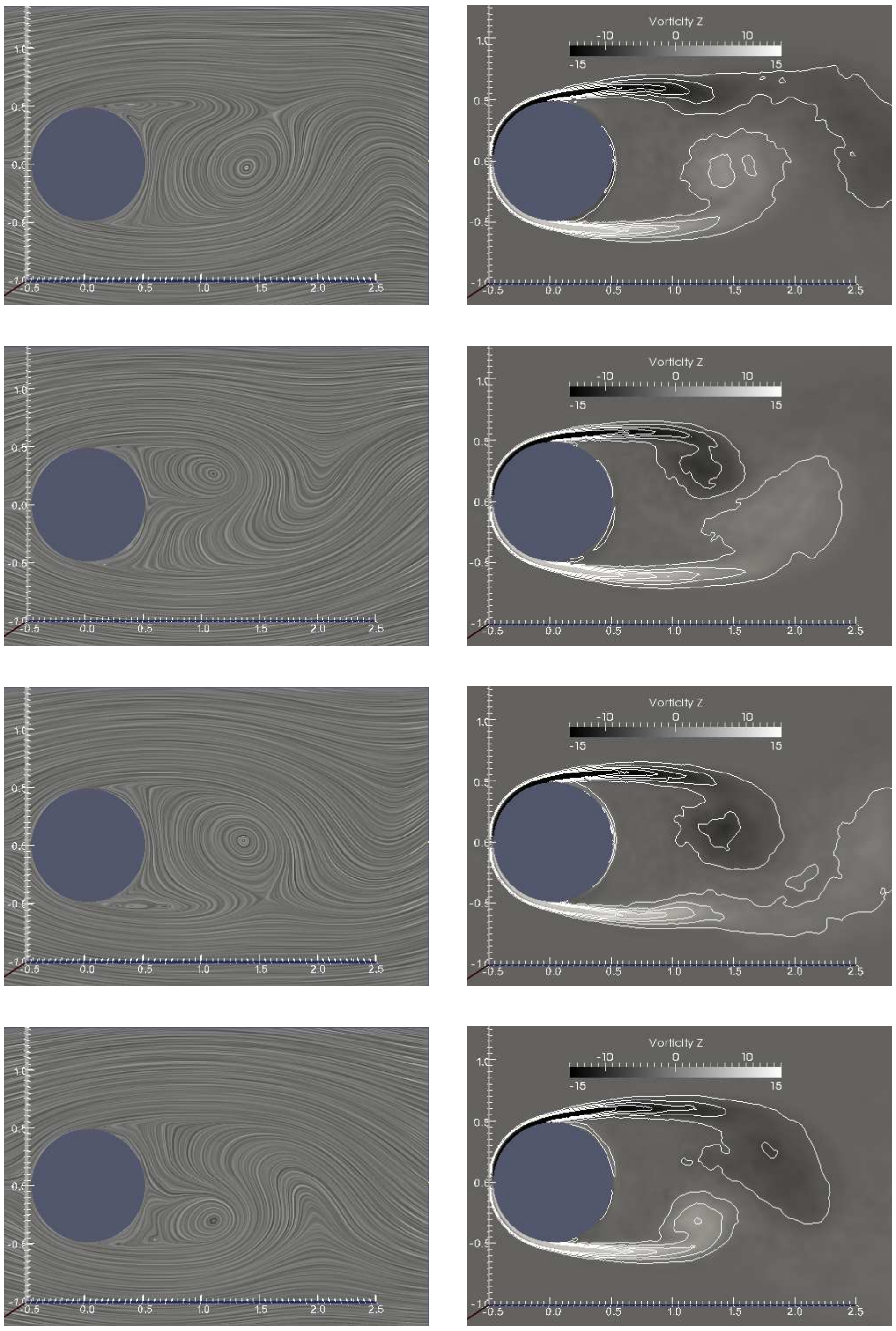


Figure 2.18: Flow around circular cylinder, streamlines (left) and Span-wise (right) vorticity contours.

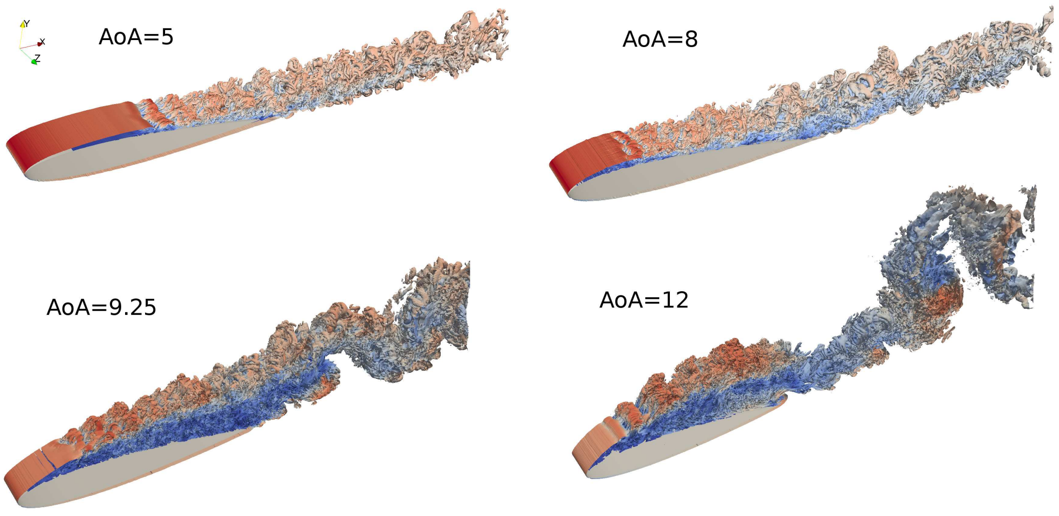


Figure 2.19: Instantaneous flow at different AoA

independent 2D subsystems which are solved by means of the DSD algorithm. Since no restrictions are imposed in the non-periodic directions, the overall algorithm is well suited for solving problems discretized on extruded 2D unstructured meshes.

The parallelization is based on a geometric domain decomposition. Different partitions are employed for the FFT-based change-of-basis (from *physical* to *spectral* space and vice versa) and for the solution of the *frequency systems*. The former operation must be performed without partitioning the mesh in the periodic direction whereas, for the latter, the number of processes to solve each 2D subsystem must be kept in the range of linear scalability of the DSD algorithm. Despite the additional transmissions of data between these two partitions, this strategy benefits the scalability of the overall algorithm.

The scalability and efficiency of the proposed method have been shown by performing several numerical experiments on the MareNostrum Supercomputer. Scalability tests using up to 8192 parallel processes with up to 10^9 million nodes meshes have demonstrated the algorithm capability on solving large-scale problems with a very short time. Finally, to benchmark it, direct numerical simulation of a turbulent flow around a circular cylinder, at Reynolds numbers 3900 and 10000, have been used as problem models. Measured computing times of the DSD solver have been compared with those obtained by the standard Conjugate Gradient method with Jacobi and block Jacobi diagonal scaling as preconditioners. It has been shown that despite the fact that the iterative methods have better speed-up, for the range of problems under consideration, they are clearly outperformed by the proposed DSD algorithm. Moreover, since the JPCG is rather simple and has no parameters, these results can be used as a reference to compare with other iterative methods.

References

[1] R. M. Gray. Toeplitz and Circulant Matrices: A review. *Foundations and Trends in Communications and Information Theory*, 2:155–239, 2006.

[2] P. J. Davis. *Circulant Matrices*. Wiley-Interscience, New York, 1979.

[3] F. X. Trias, M. Soria, A. Oliva, and C. D. Pérez-Segarra. Direct numerical simulations of two- and three-dimensional turbulent natural convection flows in a differentially heated cavity of aspect ratio 4. *Journal of Fluid Mechanics*, 586:259–293, 2007.

[4] Hongyi Xu. Direct numerical simulation of turbulence in a square annular duct. *Journal of Fluid Mechanics*, 621:23–57, 2009.

[5] H. Hattori and Y. Nagano. Direct numerical simulation of turbulent heat transfer in plane impinging jet. *International Journal of Heat and Fluid Flow*, 25:749–758, 2004.

[6] Seongwon Kang, Gianluca Iaccarino, and Frank Ham. DNS of buoyancy-dominated turbulent flows on a bluff body using the immersed boundary method. *Journal of Computational Physics*, 228(9):3189 – 3208, 2009.

[7] Jonathan R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, 1994.

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[9] William L. Briggs. *A Multigrid Tutorial*. SIAM, 1987.

[10] P. Wesseling. *An Introduction to Multigrid Methods*. Wiley, 1992.

[11] A. Gorobets, F. X. Trias, M. Soria, and A. Oliva. A scalable parallel Poisson solver for three-dimensional problems with one periodic direction. *Computers & Fluids*, 39:525–538, 2010.

[12] M. Soria, C. D. Pérez-Segarra, and A. Oliva. A Direct Parallel Algorithm for the Efficient Solution of the Pressure-Correction Equation of Incompressible Flow Problems Using Loosely Coupled Computers. *Numerical Heat Transfer, Part B*, 41:117–138, 2002.

[13] S. Kocak and H. U. Akay. Parallel Schur complement method for large-scale systems on distributed memory computers. *Applied Mathematical Modelling*, 25:873–886, 2001.

[14] F. X. Trias, M. Soria, C. D. Pérez-Segarra, and A. Oliva. A Direct Schur-Fourier Decomposition for the Efficient Solution of High-Order Poisson Equations on Loosely Coupled Parallel Computers. *Numerical Linear Algebra with Applications*, 13:303–326, 2006.

[15] R. Borrell, O. Lehmkuhl, M. Soria, and A. Oliva. Schur Complement Methods for the solution of Poisson equation with unstructured meshes. In *Parallel Computational Fluid Dynamics*, Antayla (Turkey), May 2007. Elsevier.

[16] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:1337–1356, 1999.

[17] Chi Shen and Jun Zhang. Parallel two level block ILU Preconditioning techniques for solving large sparse linear systems. *Parallel Computing*, 28:1451–1475, October 2002.

[18] Georg Pingen, Anton Evgrafov, and Kurt Maute. A parallel Schur complement solver for the solution of the adjoint steady-state lattice Boltzmann equations: application to design optimisation. *Int. J. Comput. Fluid Dyn.*, 22:457–464, 2008.

[19] C.A. Dorao and H.A. Jakobsen. A parallel time-space least-squares spectral element solver for incompressible flow problems. *Applied Mathematics and Computation*, 185(1):45 – 58, 2007.

[20] Natalja Rakowsky. The Schur Complement Method as a Fast Parallel Solver for Elliptic Partial Differential Equations in Oceanography. *Numerical Linear Algebra with Applications*, 6:497–510, 1999.

[21] F.X. Trias, R.W.C.P. Verstappen, A. Gorobets, M. Soria, and A. Oliva. Parameter-free symmetry-preserving regularization modeling of a turbulent differentially heated cavity. *Computers & Fluids*, 39:1815–1831, 2010.

[22] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1989.

[23] F.N. Felten and T.S. Lund. Kinetic energy conservation issues associated with the collocated mesh scheme for incompressible flow. *Journal of Computational Physics*, 215(2):465–484, 2006.

[24] R. W. C. P. Verstappen and A. E. P. Veldman. Symmetry-Preserving Discretization of Turbulent Flow. *Journal of Computational Physics*, 187:343–368, May 2003.

[25] Y. Morinishi, T.S. Lund, O.V. Vasilyev, and Moin. Fully Conservative Higher Order Finite Difference Schemes for Incompressible Flow. *Journal of Computational Physics*, 143:90–124, 1998.

[26] A. J. Chorin. Numerical Solution of the Navier-Stokes Equations. *Journal of Computational Physics*, 22:745–762, 1968.

[27] N. N. Yanenko. *The Method of Fractional Steps*. Springer-Verlag, 1971.

[28] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[29] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[30] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1999.

[31] ICEM CFD Hexa. *ANSYS Inc.* Canonsburg, PA, USA, http://www.ansys.com.

[32] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[33] X. Ma, G. S. Karamanos, and G. E. Karniadakis. Dynamics and low-dimensionality of a turbulent near wake. *Journal of Fluid Mechanics*, 410:29–65, 2000.

[34] P. Parnaudeau, J. Carlier, D. Heitz, and E. Lamballais. Experimental and numerical studies of the flow over a circular cylinder at Reynolds number 3900. *Physics of Fluids*, 20:85–101, 2008.

[35] S. Dong and G. E. Karniadakis. DNS of flow past a stationary and oscillating cylinder at Re= 10000. *Journal of Fluids and Structures*, 20:519–531, 2005.

[36] I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Flow dynamics in the turbulent wake of a sphere at sub-critical Reynolds numbers. *Computers and Fluids*, (in press).

[37] I. Rodríguez, R. Borrell, O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. Direct numerical simulation of the flow over a sphere at Re = 3700. *Journal of Fluid Mechanics*, 679:263–287, 2011.

[38] I. Rodríguez, R. Borrell, O. Lehmkuhl, A. Oliva, and C.D. Pérez-Segarra. CFD simulation of the thermal behavior of a wind mill power generator nacelle. In 6*th International Symposium on Turbulence Heat and Mass Transfer*, Rome (Italy), September 2009.

[39] O. Lehmkuhl, I. Rodríguez, R. Borrell, C.D. Pérez-Segarra, and A. Oliva. Low frequency variations in the wake of a circular cylinder at $Re = 3900$. In *13th European Turbulence Conference*, Varsaw (Poland), September 2011.

[40] I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Low-frequency unsteadiness in the vortex formation region of a circular cylinder. In *7th Symposium on Turbulence, Heat and Mass Transfer, THMT-12*, Palermo (Italy), September 2012.

[41] O. Lehmkuhl, I. Rodríguez, R. Borrell, and A. Oliva. High-Performance computing of flows with massive separation: flow past a NACA 0012. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[42] I. Rodríguez, O. Lehmkuhl, A. Baez, R. Borrell, and A. Oliva. Direct numerical simulation of a NACA 0012 in full stall. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.

3

On parallel Sn transport sweep algorithms

Main contents of this chapter have been published in:
G. Colomer, R. Borrell, F.X. Trias, and I. Rodríguez. Parallel algorithms for S_n transport sweeps on unstructured meshes. *Journal of Computational Physics*, *(in press, doi:10.1016/j.jcp.2012.07.009).*

Abstract. The Boltzmann Transport Equation is solved on unstructured meshes using the Discrete Ordinates Method. The flux for each ordinate is swept across the discretization mesh, within a source iteration loop that accounts for the coupling between the different ordinates. In this chapter, a spatial domain decomposition strategy is used to divide the work among the available CPUs. The sequential nature of the sweeping process makes the parallelization of the overall algorithm the most challenging aspect. Several parallel sweep algorithms, which represent different options of interleaving communications and calculations in the solution process, are analyzed. The option of grouping messages by means of buffering is also considered. One of the heuristics proposed consistently stands out as the best option in all the situations analyzed, which include different geometries and different sizes of the ordinate set. With this algorithm, good scalability results have been achieved regarding both in weak and strong speedup tests with up to 2560 CPUs.

3.1 Introduction

The availability of computational resources has been growing in the last years, mainly in the form of multi-core supercomputers. To use them, distributed memory (mostly based on MPI) and shared memory (mostly based on OpenMP) parallelization models are combined. Recently, with the appearance of general purpose GPUs, an additional shared memory parallelization model is available: one single CPU can have access to one or several GPUs, where it is possible to launch a large number of threads performing similar operations (vectorization). Existing algorithms need to evolve to use this increasing availability of computational power and take advantage of it. However, its efficient usage is not trivial and, therefore, part of the research on the numerical simulation field has been focused on the efficient parallelization of solution methods.

In this work we consider the numerical solution of the Boltzmann Transport Equation (BTE). Its solution has a high cost in terms of memory requirements and computational time. Therefore, its parallelization results in two benefits: problems are solved faster and their size can be increased.

The parallel solution of the BTE has been studied from several perspectives: MPI based [1, 2], OpenMP based [3, 4], and GPU based [5, 6]. Furthermore, the two possible ways to decompose the problem, in angular and spatial subdomains, have been thoroughly compared (see [7, 8], for instance). Spatial domain decomposition is the method that best suits our needs.

3.1.1 Solution methods for the BTE

Among the many numerical methods used to solve the BTE, the results of this work are applicable to those that employ a first-order finite volume discretization for the spatial domain, and any quadrature method to perform the angular integration. The Discrete Ordinates Method (DOM) [9] and the Finite Volume Method (FVM) [10] are two of the most popular angular quadratures. Thus, the BTE is discretized in a number of spatial nodes and angular ordinates, where, in general, the different ordinates are mutually coupled. In this work, we use the *source iteration* method to take into account such couplings: each ordinate is solved independently, while the couplings between different ordinates are deferred to the source term.

Different methods can be applied to solve each ordinate subsystem. Some authors use general iterative solvers based on Krylov or multigrid kernels. For instance, to name a few examples, Murthy and Mathur [11] applied an additive correction multigrid, Liu et al. [12, 13] used a conjugate gradient square (CGS) with the Dupond-Kendall-Rachford preconditioner and An et al. [14] applied a CGS and a BiCGSTAB. Nevertheless, for this specific problem, a *sweeping based* method, which consists on sweeping the flux across the grid from upstream to downstream nodes, is a more suitable choice because it mimics the way in which the information propagates physically. Its application is possible because, using an upwind like interpolation scheme, the discretization matrices become lower triangular.

The inherent sequential nature of the sweep process makes the parallelization of the overall algorithm the most challenging aspect for the sweeping based methods. When the spatial domain decomposition strategy (SDD) is adopted, for each ordinate the subdomains that are located in the downstream regions of the mesh need to wait until the upstream nodes have been swept. This phase, in which a processor cannot complete its tasks because it does not have all the information it requires to proceed with the calculations, is referred to as *idle time*.

To cope with this limitation imposed by the SDD, some authors use a block Jacobi approximation [3, 4, 15], in which the unknown values at the upstream boundaries of the spatial subdomains are obtained from old iterations. This method can be tuned with different prioritization strategies but, in general, suffers from degradation in the convergence rate when the number of parallel processes is increased. This degradation is inversely proportional to the optical thickness of the media [7]. Note that this approach requires an iterative procedure even in the absence of ordinate coupling.

On the other hand, the idle time that comes with the SDD can also be diminished by overlapping the waiting for the required upstream nodes at some ordinates, with the evaluation of the flux for the others. This is possible at some level because each ordinate has a different sweeping order. Maximizing this overlap is a critical point to achieve a good parallel performance. Note that in this case, contrary to the block Jacobi strategy, full sweeps across the entire grid are performed instead of asynchronous local sweeps. Thus, a direct solution is obtained for each ordinate.

Two relevant works, developing the strategy of full sweeps across the grid for unstructured meshes are the papers by Pautz [2] and Plimpton et al [1]. Both works start from a similar basic algorithm, and try to improve its performance with specific sweeping orderings that reduce the idle time originated in the parallel executions. Pautz poses this question in the framework of scheduling theory. However, general scheduling problems like this one have shown to be NP-complete [16]. Therefore, the optimal solution can only be approached by means of intuitive heuristics. Pautz proposes several low-complexity list ordering heuristics to determine the sweep order on any partitioned mesh. For some problems he obtains nearly linear strong speedup with up to 126 CPUs. On the other hand, Plimpton et al. [1] base their prioritization heuristics in simple but effective geometric criteria, obtaining good strong speedup results on up to 2048 CPUs. They also present a grid partitioning algorithm with the aim to generalize the effective columnar KBA style [17, 18] to unstructured grids.

In addition, theoretical studies of the parallel performance of MPI-based parallel sweeping based methods, have been carried out by different authors on structured [19, 20] and unstructured [21] meshes. With them it is possible to estimate the parallel performance on large numbers of CPUs.

3.1.2 Summary of the present work

In this chapter we are concerned with the parallel solution of the BTE on unstructured grids with an MPI-based spatial domain decomposition approach. It is worth noting that the distributed memory parallelization can be complemented with other parallelization strategies. However, it is an indispensable element to run the code on most of the present day supercomputers. We prefer a sweeping based strategy and we focus on the sweep operations, which are the heart of the algorithm.

Inspired by the good scalability results presented by Plimpton et al. [1], we developed several variants of the parallel sweep algorithm, using the works by Plimpton et al. and by Pautz [2] as starting point. We have studied different possibilities of alternating the communication and calculation stages and the effect of delaying the communications by means of buffering.

In this chapter only general partitioning approaches are considered. Since our goal is to couple the solution of the BTE with other equations in multi-physics problems, using a specific partition strategy (e.g. those found in [1, 19, 22]) might not be suitable in the solution of other physical phenomena.

All the numerical tests have been carried out on the IBM MareNostrum supercomputer at the Barcelona Supercomputing Center. Computing times and speedup tests obtained using up to 2560 CPUs illustrate the efficiency and scalability of the method. Additional tests on different geometries also show its robustness.

The rest of this chapter is organized as follows: in the next section, the mathematical formulation of the Boltzmann transport equation and some guidelines for its discretization are presented. The procedure to solve the derived linear systems is explained in Sections 3 (sequential version) and 4 (parallel version). Some heuristic enhancements of the basic algorithm are detailed in Section 5. Relevant results are presented in Section 6, where the performance of the different algorithms is analyzed, and final conclusions are drawn in Section 7.

3.2 The Boltzmann Transport Equation

3.2.1 Mathematical formulation

The time independent Boltzmann Transport Equation (BTE) is a conservation equation that accounts for the number ϕ of straight-propagating particles crossing a unit area normal to a given direction per time unit, and their interaction with the medium where they are propagating in. The flux ϕ will depend on the location $\mathbf{x} \in \mathbb{R}^3$ and the angular direction $s \in S^2$. In a general form, and omitting the spatial dependence in all terms for clarity, the BTE can be written as

$$\frac{d\phi^s}{d\ell(s)} + \beta_1 \phi^s = \beta_0 + \beta_2 \int_{S^2} \phi^{s'} \psi^{s's} \, d\Omega'. \tag{3.1}$$

This equation describes the variation of the flux of particles in a straight path in the angular direction s, through a unit area normal to s, where the variable $\ell(s)$ is the length of such path. This variation is divided into three contributions, with different functional dependence on ϕ, to better capture the behavior of different physical phenomena. The meaning of the various coefficients $\{\beta_i\}$ will depend on the phenomena being modeled by Equation 3.1. For example, in the context of radiative heat transfer, the coefficients $\beta_{0,1,2}$ stand for the emission, extinction and scattering processes, respectively. Note that in the integral term the unknown $\phi^{s'}$ is multiplied by the *phase function* $\psi^{s's}$. This function represents the probability of a particle changing its propagation direction from s' to s. Although we are considering the time independent form of the BTE, the parallel algorithms explained below are not affected by the addition of a transient term.

The boundary conditions for the BTE are summarized in the following equation:

$$\phi^s_{\text{bound}} = \phi^s_0 - \beta_3 \int_{\cos\theta<0} \phi^{s'} \cos\theta \, d\Omega', \tag{3.2}$$

where ϕ^s_0 is a given value at the boundary, and the integral term accounts for the reflection of particles at the boundary, being θ the angle between the inward normal to the surface and the propagating direction s'.

The BTE is used mainly in neutron transport models [23] and radiative heat transfer [24]. In the former case, the unknown ϕ represents the neutron flux, and Equation 3.1 is slightly modified by adding a transient term which accounts for the fact that not all neutrons propagate at the same velocity. In the latter case, the radiative energy is assumed to be carried by photons, and the unknown ϕ represents the number of photons per unit area and time propagating in a particular direction.

3.2.2 Discretization

The discretization of the BTE is discussed in some detail for unstructured meshes, using the Discrete Ordinates Method (DOM) to account for the angular dependence of ϕ. However, the parallel algorithms presented in Sections 3.4 and 3.5 are directly applicable for any mesh (Cartesian, body-fitted, unstructured) and for any angular discretization (DOM, FVM).

In the DOM, the angular integrals are approximated by means of an m-point Gaussian quadrature, where the integrand is evaluated at some prescribed directions $\hat{\mathbf{s}}^i$, weighted accordingly:

$$\int_{S^2} f(s) \, d\Omega \simeq \sum_{i=1}^{m} \omega^i f^i, \tag{3.3}$$

where f^i and ω^i are the value of the function and the weighting coefficient for ordinate i, respectively. Götz [25] gives an overview of different quadrature sets.

Using the DOM, the BTE becomes a system of m differential equations coupled together, one equation for each quadrature point:

$$\frac{d\phi^i}{d\ell(s^i)} = \hat{\mathbf{s}}^i \cdot \boldsymbol{\nabla}\phi^i = -\mathcal{B}^i\phi^i + \mathcal{S}^i, \tag{3.4}$$

where the superindex i stands for the ordinate at which the term is evaluated, and

$$\mathcal{B}^i = \beta_1 - \beta_2\omega^i\psi^{ii} \quad \text{and} \quad \mathcal{S}^i = \beta_0 + \beta_2\sum_{j\neq i}\omega^j\psi^{ji}\phi^j. \tag{3.5}$$

The corresponding boundary conditions are discretized as:

$$\phi^i_{\text{bound}} = \phi^i_0 - \beta_3 \sum_{\cos\theta^j<0} \omega^j\phi^j\cos\theta^j. \tag{3.6}$$

The spatial discretization on unstructured meshes is carried out by means of a finite volume approach. Integrating the left-hand side (derivative term) of Equation 3.4 over a control volume, like the one shown in Figure 3.1, we get:

$$\begin{aligned} \int_V \hat{\mathbf{s}}^i \cdot \boldsymbol{\nabla}\phi^i\, dV &= \int_V \boldsymbol{\nabla}\cdot(\phi^i\hat{\mathbf{s}}^i)\, dV \\ &= \int_{\partial V} (\phi^i\hat{\mathbf{s}}^i)\cdot\hat{\mathbf{n}}\, d\mathbf{S} \\ &\simeq \sum_f \phi^i_f A_f(\hat{\mathbf{s}}^i\cdot\hat{\mathbf{n}}_f). \end{aligned} \tag{3.7}$$

In the first step, the fact that $\hat{\mathbf{s}}^i$ is constant is taken into account. The divergence theorem is applied in the second step and, finally, in the third, ϕ^i is assumed to have a constant value ϕ^i_f on each face f, being A_f the area of such a face.

Applying the divergence theorem to a constant vector field, it is easy to see that $\sum_f A_f\hat{\mathbf{n}}_f = 0$. This relation allows us to introduce the nodal value, ϕ^i_P, in Equation 3.7:

$$\int_V \hat{\mathbf{s}}^i \cdot \boldsymbol{\nabla}\phi^i\, dV \simeq \sum_f (\phi^i_f - \phi^i_P)\, A_f(\hat{\mathbf{s}}^i\cdot\hat{\mathbf{n}}_f). \tag{3.8}$$

There exist several methods to derive ϕ^i_f from the values of ϕ^i at the neighbor nodes [26]. In this work, we use a first order upwind-like interpolation scheme (*i.e.* the value ϕ^i_f on any face is made equal to ϕ^i_P in its upstream neighbor node). Therefore, in the last equation, the contribution from the downstream faces becomes zero:

$$\int_V \hat{\mathbf{s}}^i \cdot \boldsymbol{\nabla}\phi^i\, dV \simeq \sum_f a^i_f(\phi^i_P - \phi^i_{f,\text{up}}), \tag{3.9}$$

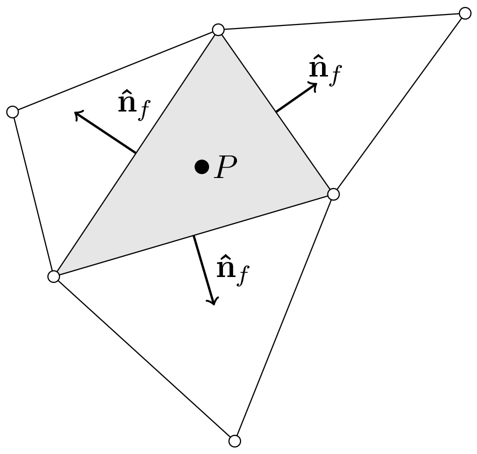


Figure 3.1: Typical control volume (shaded) in an unstructured grid. The values of ϕ^i are calculated at point P and are assumed constant within the volume.

where $a^i_f = -A_f \min(0, \hat{\mathbf{s}}^i \cdot \hat{\mathbf{n}}_f)$, and $\phi^i_{f,\text{up}}$ is the value of ϕ^i on the upstream node of the face f. The sign change in the term $(\phi^i_P - \phi^i_{f,\text{up}})$ with respect to Equation 3.8 makes the coefficients a^i_f positive.

The spatial discretization of the right-hand side term of Equation 3.4 is trivial. Thus, the fully discretized form for the BTE is:

$$\sum_f a^i_f(\phi^i_P - \phi^i_{f,\text{up}}) = -\mathcal{B}^i\phi^i_P V_P + \mathcal{S}^i V_P, \tag{3.10}$$

where V_P is the volume of the cell at node P. Rearranging the terms to clarify the structure of the system, we get:

$$\left(\sum_f a^i_f + \mathcal{B}^i V_P\right)\phi^i_P - \sum_f (a^i_f)\phi^i_{f,\text{up}} = \mathcal{S}^i V_P. \tag{3.11}$$

Using a matrix based formulation, this system can be read as $A\boldsymbol{\phi} = \mathbf{b}$, where $A \in \mathbb{R}^{mN\times mN}$, being m and N the number of quadrature points and nodes, respectively. Notice that the dependences between different ordinates are deferred to the term $\mathcal{S}^i$ (*i.e.* to the right-hand side vector $\mathbf{b}$), leading to a block diagonal form for the matrix A, one block for each ordinate. Moreover, using an upwind-like scheme, each block $A^i \in \mathbb{R}^{N\times N}$ has a lower triangular sparsity pattern, thus it can be solved by means of a forward substitution. In the next section we explain an efficient strategy to solve all ordinate blocks simultaneously.

The introduction of the nodal value in Equation 3.8 makes the system matrix diagonal dominant, even if $\mathcal{B}^i$ is zero. Care must be taken if $\mathcal{B}^i < 0$, as the diagonal coefficient could become zero, thus invalidating the forward substitution as a solution method. For physically relevant equations, as in the radiative and neutron transport equations, $\mathcal{B}^i$ is indeed positive or zero.

3.3 Solution of the discrete equation

To solve the BTE we use the source iteration (SI) procedure outlined in Algorithm 1. In short, the dependences of ϕ^i on any ordinate $j \neq i$ are treated as constants, and deferred to the source term, taking the values from the previous iteration. Then, once all the ordinates have been solved, the source term is updated with the recently obtained values. This procedure is repeated until convergence is achieved. Although we perform a Jacobi like update of the source term, a Gauss-Seidel like update is also feasible. In any case, in this chapter we are concerned about the sweeping procedure rather than the coupling between the angular ordinates. The pseudo-code for the SI algorithm is:

Algorithm 1: source iteration algorithm

```
1. difference = $\varepsilon + 1$
2. while (difference $> \varepsilon$):
3.     for each ordinate $i$:
4.         calculate the source term $\mathcal{S}^i$
5.     $\boxed{\text{parallel sweep}}$
6.     difference = 0
7.     for each ordinate $i$:
8.         difference = $\max$(difference, $\|\phi^i_{\text{actual}} - \phi^i_{\text{previous}}\|$)
```

Prior to discuss the parallel sweep algorithm, we describe the sequential procedure of sweeping ϕ along a particular direction $\hat{\mathbf{s}}^i$. As mentioned above, if the BTE is discretized using an upwind-like interpolation scheme, the resulting discretization matrix for each ordinate is lower triangular. Thus, these subsystems can be solved by sweeping the nodes in a particular order, performing what is known as a forward substitution in the linear algebra context. These ideas are developed in the next paragraphs.

Consider the node B shown in Figure 3.2. Following the discretization method detailed in Section 3.2.2, the value of ϕ at B for the ordinate i, ϕ^i_B, depends on ϕ^i_A, if and only if the projection of $\hat{\mathbf{s}}^i$ on the vector normal to the common face (and directed to B) is positive. Therefore, given an ordinate i, for each pair of adjacent nodes only one of them can contribute to the value of ϕ^i in the other. These dependences can be described by means of a directed graph, where the vertices represent the mesh nodes, and the edges (or arrows) represent the faces between two adjacent

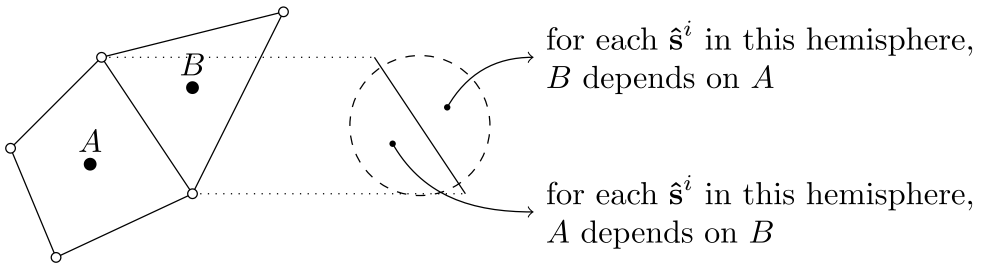
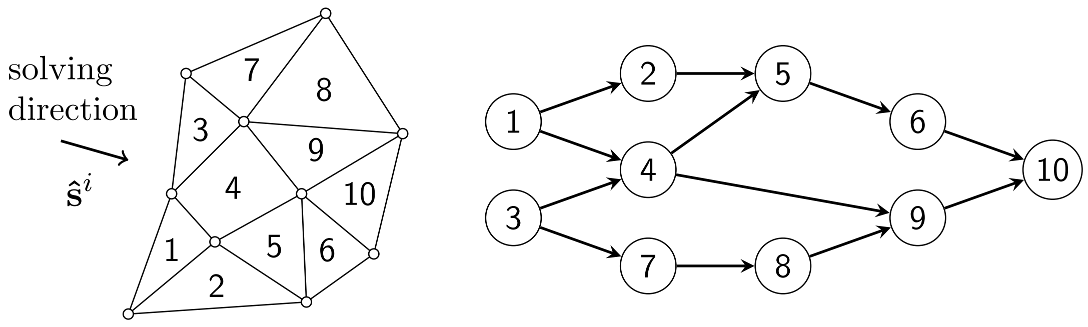

B
A
for each $\hat{\mathbf{s}}^i$ in this hemisphere,
B depends on A
for each $\hat{\mathbf{s}}^i$ in this hemisphere,
A depends on B

Figure 3.2: Two generic control volume nodes are shown. Their coupling depends on the direction, $\hat{\mathbf{s}}^i$, being solved.

nodes, and point to the *downstream* one. This is illustrated in Figure 3.3 for a two-dimensional unstructured mesh.

For each pair node/ordinate, the upstream and downstream neighbors of the node respect to the ordinate, are referred as the *incoming* and *outgoing* sets, respectively. For example, in Figure 3.3, the incoming and outgoing sets of node 4 are {1,3} and {5,9}, respectively.



$$\begin{tikzcd}
 & 2 \arrow[r] & 5 \arrow[r] & 6 \arrow[r] & 10 \\
1 \arrow[ur] \arrow[r] & 4 \arrow[ur] \arrow[rr] & & 9 \arrow[ur] & \\
3 \arrow[ur] \arrow[r] & 7 \arrow[r] & 8 \arrow[ur] & &
\end{tikzcd}$$

Figure 3.3: Dependence relations, for a particular angular direction $\hat{\mathbf{s}}^i$ on a two dimensional unstructured mesh, represented by means of a directed graph.

If the dependency graph does not contain cycles (see an example of cycle in Figure 3.4), then it is a directed acyclic graph (DAG). In this case, it can be organized in a hierarchy of levels of mutually uncoupled nodes. Level 0 contains the *upstream boundary nodes,* which are the nodes with empty incoming set. Level 1 contains the nodes that depend only on the nodes of level 0 and, as a general rule, nodes on level $k+1$ are the outgoing nodes of nodes from level k that are not in the outgoing set of nodes from levels higher than k. For example, the levels of the graph depicted in Figure 3.3 are {1,3}, {2,4,7}, {5,8}, {6,9} and {10}.

In general, given a domain, it is possible to discretize it with a mesh formed by convex polyhedra that has an associated DAG. However, some loops can occur in complex meshes. In order to overcome this problem, Plimpton et al. [1] developed a parallel algorithm for detecting and eliminating loops in the mesh. This algorithm may be used on a pre-processing stage to ensure the acyclicity of the associated graphs.

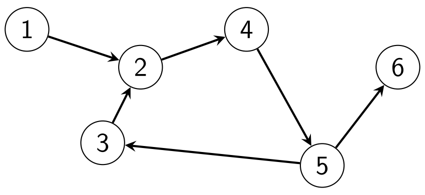


Figure 3.4: Directed graph with a cycle.

From Equation 3.11, it is clear that any node for which the elements of its incoming set are already solved, can be evaluated explicitly as

$$\phi_P^i = \frac{\mathcal{S}^i V_P + \sum_f a_f^i \phi_{f,up}^i}{\sum_f a_f^i + \mathcal{B}^i V_P}, \tag{3.12}$$

where $\sum_f a_f^i \phi_{f,up}^i$ is the contribution of the already evaluated incoming nodes, and $\mathcal{S}^i$ is evaluated taking the values from the previous source iteration. Therefore, all the nodes can be evaluated explicitly if they are swept in an order that respects the levels defined above or, more generally, an order in which each node comes before the nodes in its outgoing set. Such an order is referred as a topological sort of the associated DAG [27] and there are algorithms with linear cost to do it.

For each ordinate i, its associated DAG represents also the sparsity pattern of the submatrix A^i: the coefficient $\{A^i\}_{jk}$ is not null if and only if there is an arrow from vertex k to vertex j in the graph. Actually, a topological sort is an ordering of the unknowns such that A^i becomes lower triangular.

Therefore, an option to solve all the ordinate directions is to find a topological sort for each ordinate and then solve one system after the other by means of forward substitutions. However, in order to find a topological sort it is necessary to have information from the whole mesh, and this is not convenient with the spatial domain decomposition strategy adopted in this work (see Section 3.4). We therefore use a more indirect algorithm without such a limitation:

Algorithm 2: sweep algorithm (sequential version)

```
1. for each node $k$ and ordinate $i$:
2.     evaluate $count_{k,i}$ = # of incoming nodes for $k$ at ordinate $i$
3.     if ($count_{k,i} == 0$): insert the pair $(k,i)$ into *solvable* list $l$
4. while ($l \neq \{\emptyset\}$):
5.     remove pair $(k,i)$ from $l$ and solve it (using equation 3.12)
6.     for each outgoing node $k'$ of $k$ at ordinate $i$:
7.         decrement $count_{k',i}$
8.         if ($count_{k',i} == 0$): add pair $(k',i)$ to $l$
```

The above algorithm is based on managing a list, l, of solvable pairs node/ordinate. These are the pairs for which all the incoming neighbors have already been solved. Initially, l will contain the nodes of level 0 of each ordinate. For the rest of pairs there is a counter of the number of their unevaluated incoming nodes. As the pairs of the *solvable* list are evaluated, the counter of their outgoing elements is decremented and some of them become solvable too. This process is continued until all pairs node/ordinate are solved. A similar version of this algorithm, referred as AHOT-C-UG, can be found in [28].

3.4 Strategies for parallelization

When addressing the problem of solving the BTE in parallel, two kinds of domain decomposition are possible: either the spatial domain is divided into several parts (SDD), each processor holding data for all ordinates; or, on the other hand, the angular domain is divided into several parts (ADD), assigning a subset of ordinates to each processor. A combination of both partitioning types is also possible, although it is not considered in this chapter.

We prefer the SDD mainly to facilitate the coupling of the BTE solver with other solvers in multi-physics problems. Moreover, memory requirements are lower in the SDD since the mesh is divided among several processors. This is a major advantage when unstructured meshes are used. Besides, when angular integrals are computed, an expensive all-to-all communication is required for the ADD as the ordinates are spread over all processors. In contrast, in the SDD, this expensive communication is not needed because each processor holds the data for all ordinates. Finally, in the ADD, the number of processors is limited by the number of ordinates and, to keep the loads balanced, it is required that divides the number of ordinates. In this regard the SDD is much more flexible and the number of processors is not limited for all practical purposes.

Fischer and Azmy [8] compared the two strategies under a large range of mesh sizes and number of ordinates on a Beowulf cluster. They tested different communication models, including a new one proposed by them for the ADD. They concluded that the ADD is only best suited for cases with a large number of ordinates and few spatial nodes, when using a small number of processors.

Using the SDD, the parallel sweep algorithm is similar to the sequential one, but some point-to-point communications between processors become necessary. The parallel counterpart of Algorithm 2, based on the work by Plimpton et al. [1], is detailed in Algorithm 3:

Algorithm 3: BASIC parallel sweep algorithm for processor p

```
1.  for each owned node $k$ and ordinate $i$:
2.      evaluate $count_{k,i}$ = # of incoming nodes for $k$ at ordinate $i$
3.      if ($count_{k,i} == 0$): insert pair $(k, i)$ into the solvable list $l_p$
4.  evaluate $work_p = m \times \#$ of owned nodes in processor $p$
5.  while ($work_p > 0$):
6.      while ($l_p \neq \{\emptyset\}$):
7.          remove pair $(k, i)$ from $l_p$ and solve it (using equation 3.12)
8.          decrement $work_p$
9.          for each outgoing node $k'$ of $k$ at ordinate $i$:
10.             if $k'$ is an owned node:
11.                 decrement $count_{k',i}$
12.                 if ($count_{k',i} == 0$): add to $l_p$
13.             else:
14.                 SEND info. to the owner processor
15.     while (messages to be read):
16.         READ next pair $(k, i)$ from message
17.         for each owned outgoing node $k'$ of $k$ at ordinate $i$:
18.             decrement $count_{k',i}$
19.             if ($count_{k',i} == 0$): add to $l_p$
```

In the above algorithm, the main loop (lines 5–19) is repeated until all owned nodes at each ordinate have been solved. Hereinafter we refer to an iteration of the main loop as a *stage* of the sweep algorithm. In a partitioned mesh, for each ordinate, each subdomain will have some nodes with incoming or outgoing neighbors belonging to other subdomains. Therefore, communications between processors are necessary to complete the sweeps. When a node with an outgoing neighbor belonging to another subdomain is solved, its information is immediately sent to the adjacent subdomain (line 14). Finally, when one processor has no more tasks in its *solvable* list, messages sent from other processors are read with the aim to enable new nodes to be solved (lines 15–19).

At this point, it is worth noting the importance of solving all ordinates simultaneously. The solution time for a sweep along a single direction can hardly be reduced by the SDD parallelization, because the subdomains need to be sweeped from the upstream to the downstream zones, and the overlapping of computations becomes difficult. On the other hand, when solving all ordinates at once there are more chances of more processors working at the same time because, in general, the sweeping order is different for each ordinate. The higher the number of ordinates (*i.e.* non-equivalent DAGs), the higher is the chance of more processors working simultaneously. For example, Figure 3.5 shows a mesh decomposed in two parts and a direction $\hat{\mathbf{s}}^i$ to be solved. If only the direction $\hat{\mathbf{s}}^i$ is solved, initially only the processor which owns the left part of the mesh (white cells) works, while the other processor (which owns the gray cells) must wait to receive the needed incoming elements. However, if directions $\hat{\mathbf{s}}^i$ and $-\hat{\mathbf{s}}^i$ are solved at the same time, the two processors work simultaneously most of time.

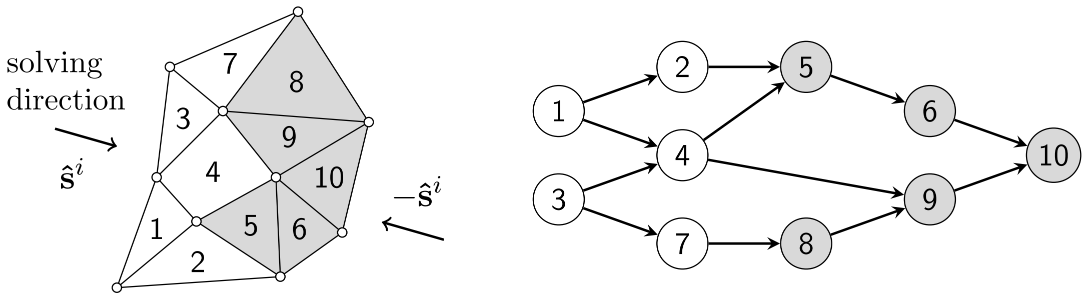


Figure 3.5: Left: Partitioned two-dimensional mesh. Right: Directed graph representing the dependencies for direction $\hat{\mathbf{s}}^i$. The graph for $-\hat{\mathbf{s}}^i$ is the same, but with all arrows reversed.

The message passing of the BASIC algorithm (Algorithm 3), is asynchronous in nature. When a pair node/ordinate is evaluated, if one of its outgoing nodes belongs to another processor, a small message is sent to it, whether the other processor is waiting for it or not. To do so, we use the standard MPI functions MPI_BSend, MPI_Iprobe and MPI_Recv. Note that to read all the arrived messages (lines 15–19 of Algorithm 3), it is necessary to call the functions MPI_Iprobe and MPI_Recv repeatedly, until the function MPI_Iprobe indicates that there are no more messages to be read. Only the neighbor subdomains need to be tested with MPI_Iprobe.

Apart from the communication costs, the main aspect that degrades the performance of the algorithm is the idle time that occurs when a processor has no tasks in the *solvable* list, l_p, but has still work to be done. In this situation, the processor is simply waiting to receive the information required to proceed.

The BASIC algorithm (Algorithm 3) is similar to the basic algorithm described by Plimpton et al. [1], and to the one used by Pautz [2]. An enhancement that is studied in these and other works, *e.g.* Kumar et al. [22], is the reordering/prioritizing of tasks within the *solvable* list, l_p, in order to minimize the idle time. This could be accomplished by using a priority queue instead of the list l_p. In this work we group the tasks of the *solvable* list by directions, instead of using such prioritizing techniques. We also focus our attention on the management of message passing between processors, studying aspects such as how to alternate the solution of the solvable tasks and the communication episodes, and the effect of reducing the number of communications by means of buffering.

3.5 Enhancements

3.5.1 Reordering tasks by directions

The strategy used by Plimpton et al. [1] is based on the prioritization of pairs node/ordinate that impose the most dependencies on other nodes. They sustain this approach by stating that solving these nodes early can potentially reduce the idle time. They use a geometric heuristic to estimate the level of dependencies and combine this criterion with trying to group the tasks by ordinates.

On the other hand, Pautz [2] uses scheduling criteria based on the properties of the associated DAGs. He uses static schedules, fixed a priori as a preprocessing stage, while the algorithms of Plimpton et al. prioritize the tasks on run-time as they appear in the *solvable* list. As a consequence, the option used by Plimpton et al. is more flexible to possible delays of incoming information due to interprocessor communications, while the reorderings archived by Pautz are more accurate.

For each of the parallel sweep (PS) algorithms that are used in this chapter, we set as a sole criterion that the tasks are grouped by directions. The objective is to advance as quickly as possible within each ordinate (avoiding jumps from one ordinate to another), and send the outgoing data to the processors that are waiting for it. For this purpose, a list of tasks for each direction is used, and in each list the tasks are solved in the same order as they are stored (first-in first-out order). The pseudo-code of the algorithm is:

Algorithm 4: parallel sweep algorithm by directions (PSD) for processor p

```
1.  for each node $k$ and ordinate $i$:
2.      evaluate $count_{k,i}$ = # of incoming nodes for $k$ at ordinate $i$
3.      if ($count_{k,i} == 0$): insert node $k$ into solvable list $l_p^i$
4.  evaluate $work_p = m \times \#$ of owned nodes in processor $p$
5.  while ($work_p > 0$):
6.      for each ordinate $i$:
7.          while ($l_p^i \neq \{\emptyset\}$):
8.              remove node $k$ from $l_p^i$ and solve it (using equation 3.12)
9.              decrement $work_p$
10.             for each outgoing node $k'$ of $k$:
11.                 if ($k'$ is an owned node):
12.                     decrement $count_{k',i}$
13.                     if ($count_{k',i} == 0$): add to $l_p^i$
14.                 else
15.                     SEND info to the owner processor
16.     while (messages to be read):
17.         READ next pair $(k, i)$ from message
18.         for each owned outgoing node $k'$ of $k$ at ordinate $i$:
19.             decrement $count_{k',i}$
20.             if ($count_{k',i} == 0$): add to $l_p^i$
```

In the algorithm described above, the *solvable* list is split by ordinates and the new lists l_p^i are solved one after the other (lines 6–15). While the tasks of l_p^i are being solved, if new nodes become solvable for ordinate i, they are appended to l_p^i (line 13). A different option could be leaving this new incoming tasks for the next stage of the main loop. In this case, the ordinate being solved would alternate more often and more reading episodes would be interspersed with calculations. The pseudo-code of this new option is:

Algorithm 5: parallel sweep algorithm alternating directions (PSAD) for processor p

```
(1–12). lines 1–12 of Algorithm 4.
⋮
13.                 if ($count_{k',i}$ == 0): add to $\ell_p^i$
14.             else
15.                 SEND info to the owner processor
16.     $l_p^i = \ell_p^i$
⋮
(17–21). lines 16–20 of Algorithm 4
```

In Algorithm 5, an auxiliary list, ℓ_p^i, is used to store the new solvable nodes at ordinate i (line 13). Once all the tasks of the list l_p^i are solved, the elements of ℓ_p^i are moved to l_p^i in order to be solved in the next stage of the main loop (line 16).

Finally, the last option here considered consists of solving only one ordinate in each stage of the main loop, choosing the one that accumulates more tasks in its list. The algorithm for this option reads:

Algorithm 6: parallel sweep algorithm by single direction (PSSD) for processor p

```
(1–5). lines 1–5 of Algorithm 4.
⋮
6.      for the ordinate $i$ with the longest $l_p^i$:
⋮
(7–20). lines 7–20 of Algorithm 4.
```

Note that the only change respect to Algorithm 4 is that in line 6, rather than iterate through all ordinates, the one that has more tasks on its list is chosen.

3.5.2 Message buffering

We also consider the option of grouping messages by means of buffering. This option reduces the number of communications between processors and, therefore, the costs associated with the latency of the parallel machine being used. However, as the processors will hold information necessary to other processors to complete their work, it may result in more idle time. In any case, we have considered two options: the first is not to use buffers, as in algorithms 4, 5, and 6, and the second consists in buffering all the outgoing messages to send them once the solution of tasks in the *solvable list* is finished, just before the reading process. In summary, we consider six variants of the BASIC algorithm: the three algorithms defined above (PSD, PSAD, PSSD), and the three algorithms derived from them by buffering the messages that were immediately sent in non-buffered versions. These new versions of the algorithms are referred to as PSD-b, PSAD-b and PSSD-b. For example, the pseudo-code of the algorithm PSD-b is:

Algorithm 7: parallel sweep algorithm by directions and buffering (PSD-b) for processor p

```
1.  for each node $k$ and ordinate $i$:
2.      evaluate $count_{k,i}$ = # of incoming nodes for $k$ at ordinate $i$
3.      if ($count_{k,i} == 0$): insert node $k$ into solvable list $l_p^i$
4.  evaluate $work_p = m \times \#$ of owned nodes in processor $p$
5.  while ($work_p > 0$):
6.      for each processor $q$:
7.          BUFFER$_q = \{\emptyset\}$
8.      for each ordinate $i$:
9.          while ($l_p^i \neq \{\emptyset\}$):
10.             remove node $k$ from $l_p^i$ and solve it (using equation 3.12)
11.             decrement $work_p$
12.             for each outgoing node $k'$ of $k$:
13.                 if ($k'$ is an owned node):
14.                     decrement $count_{k',i}$
15.                     if ($count_{k',i} == 0$): add to $l_p^i$
16.                 else
17.                     $q$ = owner of node $k'$
18.                     add info for node $k'$ in BUFFER$_q$
19.     for each processor $q$:
20.         SEND BUFFER$_q$
21.     while (elements to be read):
22.         READ next pair $(k, i)$ from message
23.         for each owned outgoing node $k'$ of $k$ at ordinate $i$:
24.             decrement $count_{k',i}$
25.             if ($count_{k',i} == 0$): add to $l_p^i$
```

Note that the size of the buffers depends on the algorithm chosen. In the PSD-b algorithm, the buffer accumulates information from all ordinates. This is also the case for the PSAD-b algorithm, but with smaller buffer sizes (because the new incoming tasks are deferred to the next stage of the main loop). Finally, in the

PSSD-b algorithm the buffers only accumulate information from the chosen ordinate. In all three cases, the buffer sizes are variable and it is necessary to call the function MPI_Get_count before reading them. The function MPI_Buffer_attach is used to ensure that enough memory is available for the sending buffers.

In the buffered versions, the prioritizing techniques mentioned earlier, *e.g.* those used in [1,2,22], have no effect on the parallel performance because the communications are delayed. On the other hand, if GPUs were used to vectorize the solution of the solvable nodes, buffering would reduce the number of communications between the GPUs and the CPU.

The variants of the BASIC algorithm introduced in this work represent different forms of combining communications and calculations in a parallel sweep algorithm. The numerical experiments carried out in the next section show that these variants have a significant impact on the parallel performance. According to the results of these experiments, the PSD-b stands out as the best algorithm in terms of solution time and parallel performance.

3.6 Numerical experiments

Our main interest is to acquire a deep understanding of the results obtained with the different PS algorithms. For this purpose, the different (but interrelated) parts that compose the algorithms have been analyzed separately to better understand the importance of each one in the global parallel performance. Transparent medium ($\beta_1 = \beta_2 = 0$) and Dirichlet boundary conditions ($\beta_3 = 0$) are considered. In this particular situation, the source term is constant, so the use of the source iteration algorithm (SI) is not required. Note that almost all of the computational cost of an iteration of the SI loop corresponds to the PS algorithm. Therefore, the scalability properties of the complete SI loop, will be similar to those obtained for the PS algorithm.

The geometries used in the tests have been discretized on the unstructured meshes summarized in Table 3.1. These meshes do not contain cycles, which is an indispensable requirement for the PS algorithms described before. Although possible, it is uncommon that cycles appear in geometric discretizations. For these cases, Plimpton et al. [1] developed a parallel algorithm for detecting and eliminating cycles that could be used as a pre-processing stage in the general case.

We used relatively small meshes because at most 2560 CPUs were available for the tests. With bigger meshes, scalability would be extended to higher number of CPUs, and the speedup degradation would start out of the range of processors available. Unless otherwise stated, the angular domain is divided into 80 ordinates arranged according to the S_8 quadrature (satisfying the odd moment condition [29]).

All the numerical tests have been carried out on the MareNostrum supercomputer at the Barcelona Supercomputing Center (BSC). At the moment when these

Name	Size	C.V.	Domain description
mI-S	18 757	tetrahedron	sphere
mI-L	151 265	tetrahedron	sphere
mII	18 878	tetrahedron	prism with aspect ratio 3
mIII	18 691	tetrahedron	cube with an empty sphere in the center
mIV	18 762	triangles	square (2D geometry)

Table 3.1: Meshes used on the numerical experiments. The size of a mesh refers to its number of cells, and C.V. states the type of control volume.

tests have been performed, this was an IBM BladeCenter JS21 Cluster with 10 240 PowerPC 970MP processors at 2.3 GHz, with 1 MB cache per processor. Quad-core nodes with 8 GB RAM, were coupled by means of a high-performance Myrinet network. The code has been compiled in a Linux SuSe Distribution using the IBM XL C/C++ enterprise edition compiler, version 10.1. Moreover, the version mx of MPICH was used for the message-passing. Results have been obtained by averaging over 300 runs of the same problem to reduce dispersion.

In a multi-core architecture, the performance of the CPU cores can be affected by the shared memory bandwidth, resulting in lower performance of each core when all cores are engaged [30]. To circumvent this variable, all the tests have been performed by using all the cores within each quad-core processor. Consequently, the number of MPI processes is restricted to multiples of 4.

3.6.1 Analysis of the algorithms

The execution of any of the PS algorithms can be divided into three parts:

i) *Communications call*: this includes the filling/emptying of the buffers and the calls to MPI routines on the send and receive phases. As the MPI modes used are asynchronous and non-blocking, the costs related to the bandwidth are not considered in this part.

ii) *Idle time*: inactivity phase that occurs when a CPU requires information from other CPUs to be able to proceed. This waiting time can be due to the network bandwidth limitation (information has already been sent but has still not arrived), or to the *algorithm blocking* (other processors have yet to solve the required upstream nodes, thus the information has still not been sent).

iii) *Solve*: solve the tasks of the *solvable* list and add the new solvable tasks into it.

Figure 3.6 shows the time spent on each part for the different PS algorithms, when solving the mesh mI-L using 32 (top) and 1024 (bottom) CPUs, respectively.

Note that, in the top plot, the scale of the time axis is about 40 times larger. The values for these measurements are explicitly shown in Table 3.2, together with the number of stages of the main loop required to complete the work. The stages carried out during the idle time are not counted. We use the function gettimeofday to measure the wall clock time spent in each part of the algorithm. The increase in the execution time caused by this monitoring is less than 1% in all cases. The function MPI_Barrier is called before starting and after finishing the sweeping process, thus the time obtained is the same for all CPUs involved in the execution. However, the time spent in each part of the algorithm differs from one CPU to the other. The following analysis, and the data reported in Figure 3.6 and Table 3.2, are based on the averaged values over all the CPUs.

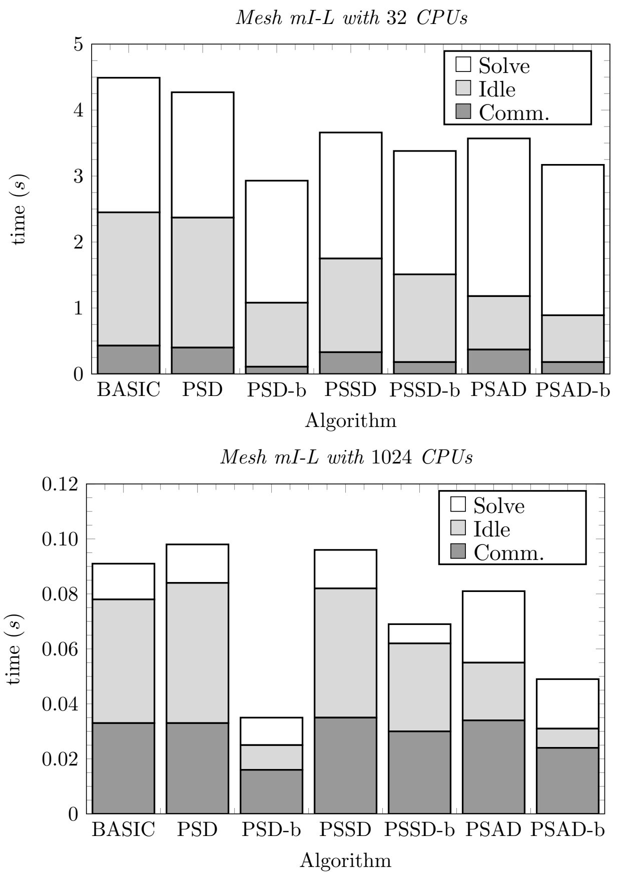


Figure 3.6: Wall clock time spend in each part for the different PS algorithms. Tests are carried on the mesh mI-L using 32 (top) and 1024 (bottom) CPUs, respectively.

In the buffered versions of the PS algorithms, the messages are not sent immediately but held back until the solution of the tasks of the solvable lists is finished.

	BASIC	PSD	PSD-b	PSSD	PSSD-b	PSAD	PSAD-b
				32 CPU			
total	4.49	4.27	2.93	3.66	3.38	3.58	3.17
comm.	0.43	0.40	0.11	0.33	0.18	0.37	0.18
idle	2.02	1.97	0.97	1.42	1.33	0.81	0.71
solve	2.04	1.90	1.85	1.91	1.87	2.39	2.28
stages	5927	5953	963	6631	2894	8711	3204
				1024 CPU			
total	0.091	0.098	0.035	0.096	0.069	0.081	0.049
comm.	0.033	0.033	0.016	0.035	0.030	0.034	0.024
idle	0.045	0.051	0.009	0.047	0.032	0.021	0.007
solve	0.013	0.014	0.010	0.014	0.007	0.026	0.018
stages	1040	981	515	1451	1156	1218	764

Table 3.2: Wall clock time (in seconds) spent in the different parts and number of stages of the main loop, for each of the PS algorithms. Tests are carried out on the mesh mI-L, using 32 and 1024 CPUs respectively.

Although it seems that a priori this strategy should increase the *idle time*, results in Table 3.2 show that it actually decreases. By using buffering, the number of communications is reduced and, therefore, also the costs of the *communications call* part. In addition, the solution of tasks from the list is carried on more continuously, with less interruptions due to the communications. This produces a drop in the *solve* part costs, especially using 1024 CPUs when most of the CPU data fits in the cache. In conclusion, by using buffering the sweep operations become faster and, as a consequence, the *algorithm blocking* gets reduced.

Looking at each part separately, we see that using 32 CPUs, the percentage of time spent on the *communications call* is rather small (between 4% and 10%), while when using 1024 CPUs this percentage grows significantly (up to between 34% and 49%). The reason for this behavior is that, although the size of the subdomains is reduced 32 times, the time spent in the *communications call* part is only reduced by a factor between 7 and 13, depending on the algorithm. As a consequence, this part ends up slowing down the global speedup. The PSD-b algorithm is the one with less costs associated to the *communications call* part.

The *solve* part is the one with the highest speedup. When the number of CPUs is increased, the amount of tasks to be done per CPU decreases in the same proportion. The number of stages required also decreases, making the algorithm more efficient. Both factors, together with cache effects, produce a superlinear speedup that, for all algorithms, averages around 150 times for an increase of only 32 times in the number of CPUs. Recalling that the number of tasks to be done is the same for all the algorithms, differences on the performance for this part lie on the different ways of alternating between the solve, send and receive stages, and the effect that this produces on the efficiency of the computations. For the best algorithm, the PSD-b, this part has a speedup of 181 times: using 32 CPUs it represents a 63% of the total time, while using 1024 CPUs it represents only a 28%.

Finally, the influence of the *idle time* on the global speedup depends on the PS algorithm. In particular, the reduction of the *algorithm blocking* is strongly linked to the acceleration of the *solve* part: if the sweep operations become faster, the neighbor upstream subdomains require less time to complete their tasks. Except for the PSD-b and the PSAD-b, the acceleration of this part is slightly lower than the acceleration of the overall algorithm (on average 37 vs. 45 times). However, a much higher acceleration has been observed for the PSD-b and PSAD-b algorithms: the partial speedups are of 108 and 101 times (resulting in global speedups of 84 and 65 times), respectively.

Summarizing, three interrelated factors, listed below from the least to the most beneficial, determine the speedup of the PS algorithms:

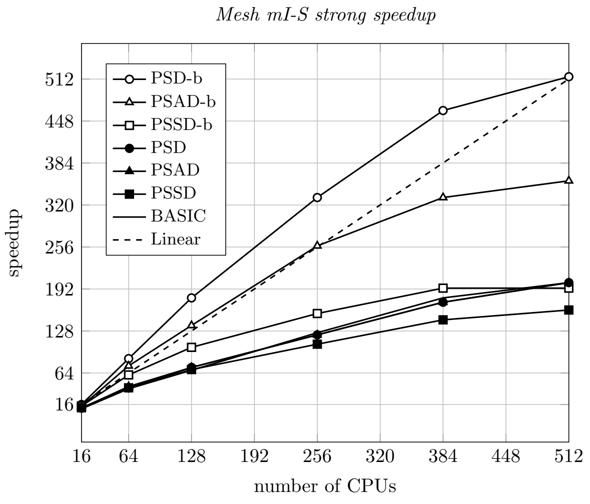
1. The *communications call* is the part of the PS algorithms with the most limited speedup. The ratio of time spent in it increases with the number of CPUs and ends up slowing down the global acceleration.

2. The *idle time* is a degradation factor for the PS algorithms. Nevertheless, in contrast to the *communications call*, it contributes to the speedup as it has a superlinear reduction (except for the PSSD). The acceleration of the *algorithm blocking* component in the idle time is directly linked with the acceleration of the *solve* part.

3. The *solve* part constitutes the most significant contribution to the speedup of the PS algorithms. Increasing the number of CPUs, the number of unknowns to be solved by each one decreases proportionally and the operations are performed more efficiently, resulting in a superlinear speedup for this part.

3.6.2 Strong speedup

The *strong* speedup measures the acceleration of an algorithm with the number of CPUs. Strong speedup tests for the mesh mI-S are depicted in Figure 3.7 (top). Considering that the angular domain is discretized into 80 ordinates, the total amount of unknowns is of about $1.5M$. At first sight, we observe that the PSD-b clearly outperforms the rest of the PS algorithms. The speedup gets exhausted at around 512 CPUs, with a parallel performance of 100%, remaining only about 37 nodes (2960 unknowns) per CPU, approximately. The walk clock time spent by this algorithm to perform the parallel sweep with 512 CPUs is 0.009 s.



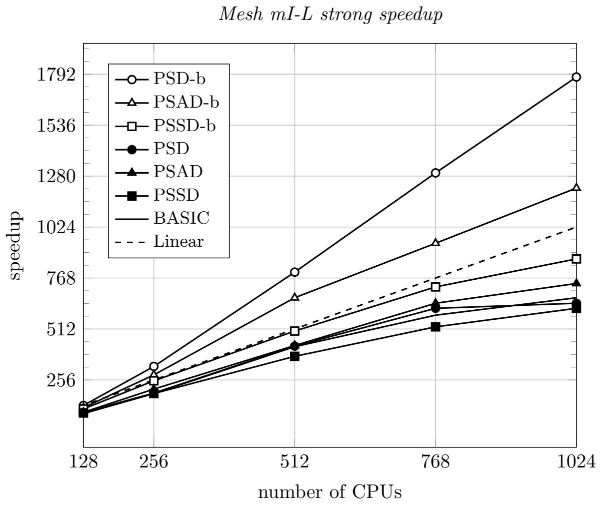


Figure 3.7: Strong speedup tests for the different PS algorithms. The speedup is calculated based on the time of the fastest algorithm, namely the PSD-b. Top: mesh mI-S. Bottom: mesh mI-L.

Tests on the larger mesh, mI-L, from 128 to 1024 CPUs, are shown in Figure 3.7 (bottom). In this case, the total number of unknowns is around $12.1M$, and the PSD-b clearly outperforms the rest of algorithms too. Figure 3.8 shows the evolution of the speedup of the PSD-b algorithm for an increased range up to 2560 CPUs (the maximal number available for the tests). With this number of CPUs, the algorithm has a parallel performance of 126%, remaining only about 60 nodes (4800 unknowns) per CPU approximately. The minimal walk clock time obtained is 0.019 seconds.

In Table 3.3, the distribution of the time and the number of stages, for the executions on the meshes mI-S and mI-L, are compared. The loads per CPU are the same, and equal to the minimal load achieved with the largest test on the mesh

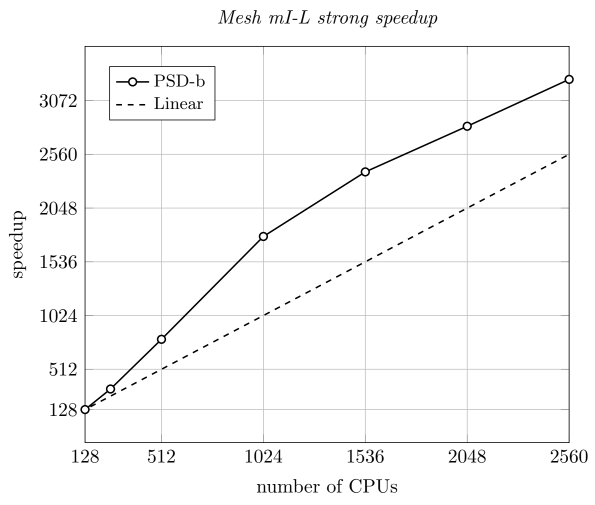


Figure 3.8: Strong speedup for the PSD-b algorithm on the mesh mI-L.

mI-L. We see that increasing the mesh size leads to a larger number of stages of the main loop. This mainly results in larger *communication call* costs, because there are more communication episodes, and longer *idle time*. The cost of the *solve* part stays the same.

Mesh	CPUs	load/CPU	total time	solve	comm.	idle	stages
mI-S	320	59	0.011	0.003	0.006	0.002	174
mI-L	2560	59	0.019	0.003	0.011	0.005	572

Table 3.3: Wall clock time spent in the different parts of the algorithm, and number of stages of the main loop, for the PSD-b algorithm. Tests are carried out on the meshes mI-S and mI-L using 320 and 2560 CPUs, respectively.

3.6.3 Superlinearity

In general, in other more common parallel linear solvers, such as Krylov or multigrid methods, the execution process is divided into only two parts: the *inner calculations* and the *communications*. One compensates the other in the speedup. For the PS algorithms, there is an extra contribution, given by the *algorithm blocking*, that favors the superlinearity of the overall algorithm. As mentioned earlier, the algorithm blocking is one of the two components of the *idle time*; the other one is caused by the network bandwidth limitations.

In order to see the effect of the algorithm blocking on the speedup, it would be necessary to remove its contribution from the global speedup, and compare the results. However, we couldn't devise a way to measure the two different components of the idle time separately. Therefore, we can only measure the effect of the algorithm blocking by observing the speedup when we omit the *idle time* as a whole.

In Figure 3.9, the speedup with the idle time removed is shown, along with the speedup of the complete algorithm for reference. It can be seen that the superlinearity gets reduced, although it is still present. However, as we are omitting all the *idle time*, we are not considering the time spent by the messages in the network. Thus, we are only considering the latency of the communications but not the network costs, that depend on its bandwidth. If we could discard only the algorithm blocking the result would be closer to the linear behavior.

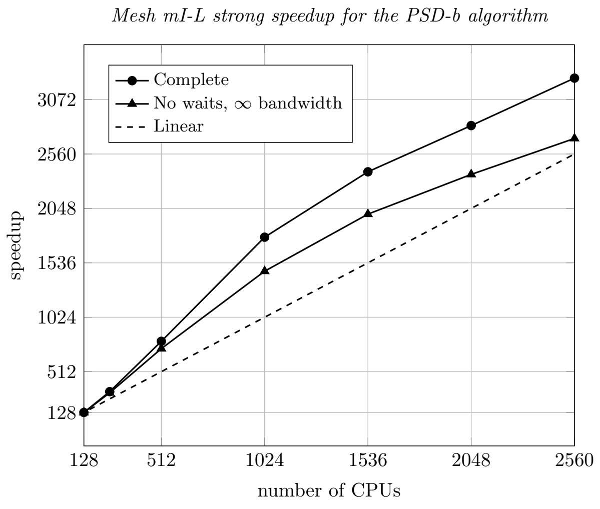


Figure 3.9: Strong speedup of the PSD algorithm for the mesh mI-L both considering and disregarding the idle time.

3.6.4 Weak speedup

The weak speedup shows the scalability of the algorithm when the ratio between the number of unknowns and CPUs remains constant. Ideally, the solution time remains also constant. However, both the cost of the *communications call* and the *idle time* tend to grow with the number of processors, eventually limiting the scalability.

A weak scalability test for the PSD-b algorithm in a spherical domain is depicted in Figure 3.10. Several meshes have been tested, keeping the ratio nodes/CPU equal to that obtained for the mI-L mesh using 128 CPUs (1180 mesh nodes, about 94 400 unknowns, per CPU). Although when using unstructured meshes it is difficult to obtain a specific number of nodes, the maximal deviation obtained is around 1%. Despite the fact that the size of the mesh and the number of CPUs are increased 160 times, the solution time grows only by a factor of 2.31. Moreover, the degradation from 128 to 2560 CPUs is only about 38% while the size of the problem increases 20

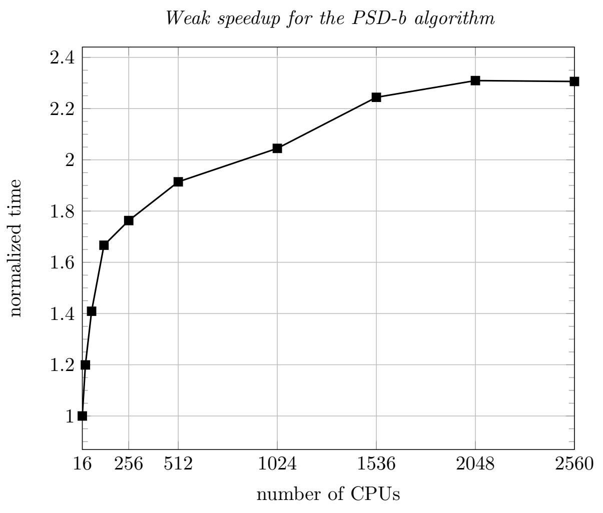


Figure 3.10: Weak speedup for the PSD-b algorithm in a spherical domain. The number of nodes per CPU is kept constant at around 1180 elements (about 94 400 unknowns per CPU).

times. The largest mesh solved has $3M$ nodes ($240M$ of unknowns) and the solution time on 2560 CPUs is 0.67 s.

3.6.5 Influence of angular discretization

Up to now, all tests have been performed using the S_8 angular discretization, with a size of 80 ordinates. In this section, additional ordinate sets are tested: the S_4 (24 ordinates) and S_6 (48 ordinates) quadratures, as defined in [29] (satisfying the odd moment condition), and the S_{10} (120 ordinates) and S_{12} (168 ordinates) quadratures, as defined in [31]. The increase of the solution time with the number of ordinates for the mesh mI-L, using 128 CPUs, is shown in Figure 3.11. We see that, as expected, the solution time and the number of ordinates are proportional.

Additional strong and weak speedup tests have been performed for the S_4 and S_{12} quadratures, using the same spatial meshes as in the previous tests. New results, together with those obtained with the S_8 quadrature, are depicted in Figure 3.12. Increasing the size of the ordinate set improves both the strong and weak speedups. There are two reasons for this: i) As, in general, each ordinate has a different associated DAG, by increasing the number of ordinates there are more chances that more processors work simultaneously (this reduces the algorithm blocking effect). ii) When increasing the number of ordinates, the percentage of time spent in the *communications call* also decreases because, proportionally, fewer messages (buffering information of more ordinates) are sent. As a result, the PSD-b algorithm spends a higher percentage of time in the solve part, which is the part of the algorithm that scales the best. This leads to an improvement of the overall speedup.

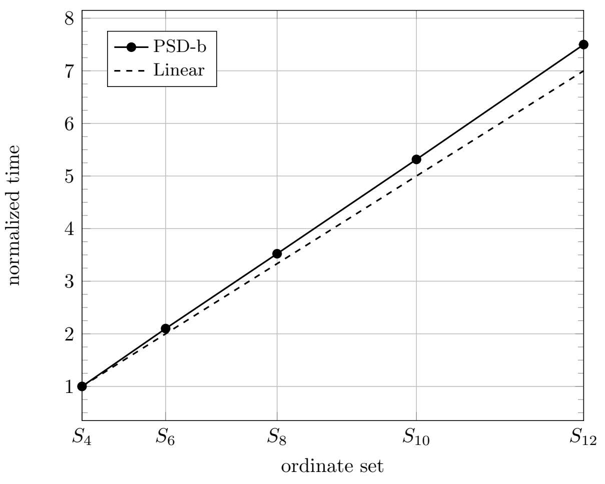


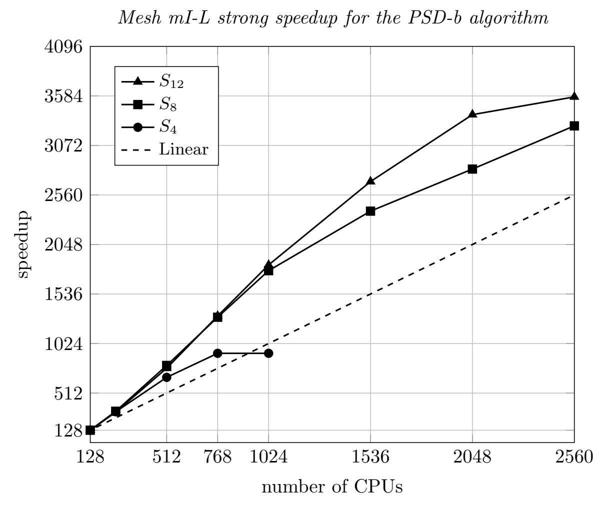
Figure 3.11: Normalized completion time (time(S_n)/time(S_4)) as a function of the ordinate set for the mesh mI-L.

Results of the strong speedup test are shown in Figure 12 (top). With 768 CPUs (the last point in which the speedup for the three quadratures still increases), the percentage of time spent in the *solve* part is 22%, 32% and 42% for the angular discretizations S_4, S_8 and S_{12}, respectively. Furthermore, on the weak speedup test (Figure 3.12 bottom), using 2560 CPUs the *solve* part represents 11%, 24% and 31% of the total time for each of the quadratures, respectively. Note that in Figure 3.12 (top and bottom) the gap between S_4 and S_8 is bigger than the gap between S_8 and S_{12}. This is because the number of ordinates is increased by a factor of 3.3 in the first case, and by a factor of 2.1 in the last one.

3.6.6 Tests in different geometries

So far, all the calculations have been carried out in a canonical spherical domain. With the aim of testing the effect of the geometry on the performance of the PS algorithms, we have done additional tests on the different geometries shown in Figure 3.13 and described in Table 3.1. These geometries are a prism with aspect ratio 1:1:3 and a cube with a spherical hole in the center, both discretized with tetrahedral meshes, and a 2D square domain discretized by means of triangles. All these new discretizations have approximately the same size as the mesh mI-S. Note that if the number and type of the control volumes of two discretizations are equal, the variations on the geometry of the domain only affect the structure of the DAG associated to each ordinate.

The solution times for the different geometries and PS algorithms, using 16 and 384 CPUs, are reported in Table 3.4. It is important to remark that the PSD-b algorithm performs the best in all geometries tested. Moreover, for the PSD-b the solution time is similar in all the 3D geometries tested. This indicates that, in these



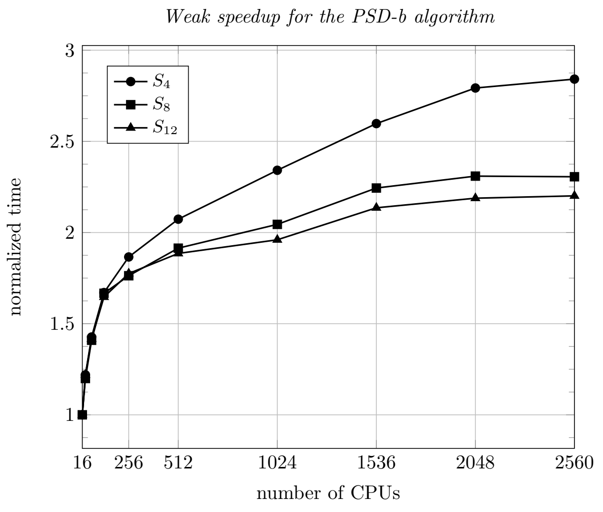


Figure 3.12: Strong (top) and weak (bottom) speedups for several angular discretizations: S_4, S_8, S_{12}.

situations, the dependence of the solution time in the DAGs structure is not very strong. More significant changes occur if 2D and 3D geometries are compared. In this case, the stencils obtained with triangular and tetrahedral elements are not the same. Therefore, besides the differences in the DAGs, the sparsity of the matrices also changes.

3.7 Concluding remarks

We have presented different variants of a sweeping based algorithm for the numerical solution of the Boltzmann Transport Equation on unstructured meshes. The Discrete Ordinates Method and a first-order upwind like discretization have been

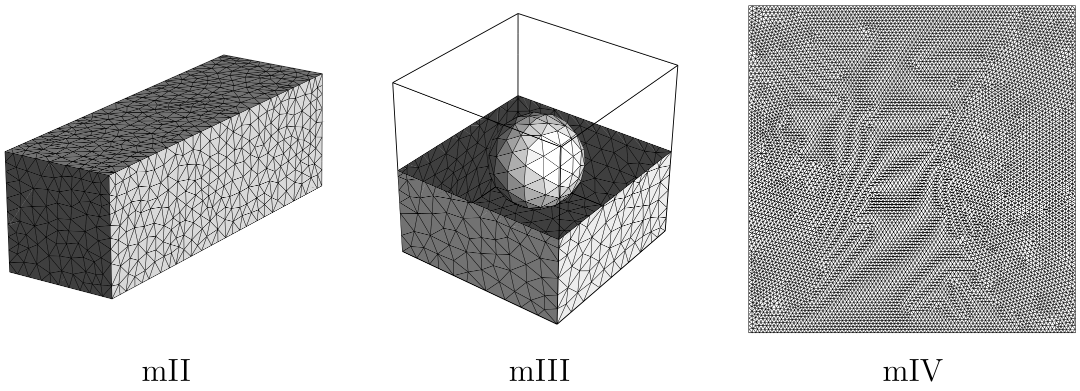


Figure 3.13: Meshes of the new geometries. All of them have around 18.8k nodes. Mesh mII is a prism with aspect ratio 1:1:3, mIII is a cube with a spherical hole in the center, and mIV is a two dimensional square domain.

	Mesh used			
	mI-S	mII	mIII	mIV (2D)
	(16 - 384 CPU)	(16 - 384 CPU)	(16 - 384 CPU)	(16 - 384 CPU)
BASIC	0.50 - 0.026	0.45 - 0.024	0.41 - 0.024	0.25 - 0.018
PSD	0.45 - 0.027	0.38 - 0.025	0.36 - 0.023	0.22 - 0.015
PSD-b	*0.29 - 0.010*	*0.27 - 0.012*	*0.26 - 0.010*	*0.18 - 0.010*
PSSD	0.42 - 0.032	0.37 - 0.032	0.37 - 0.027	0.23 - 0.021
PSSD-b	0.34 - 0.024	0.31 - 0.025	0.31 - 0.021	0.20 - 0.016
PSAD	0.45 - 0.027	0.42 - 0.026	0.41 - 0.024	0.25 - 0.022
PSAD-b	0.35 - 0.014	0.36 - 0.017	0.34 - 0.014	0.24 - 0.015

Table 3.4: Total time for several geometries and algorithms. The two numbers on each entry are the total time for 16 and 384 CPUs respectively.

used for the angular and spatial subdomains, respectively. The parallelization is carried out by means of a spatial domain decomposition strategy. Its main drawback is the idle time produced when a CPU requires information from upstream nodes belonging to other CPUs to be able to proceed with the sweeping process at any ordinate. This handicap is overcome by overlapping the calculations of the flux at some ordinates with the waiting for upstream values at the others.

The PSD-b algorithm (Algorithm 7 described in Section 3.5.2), which consists of completing all the solvable tasks for all ordinates prior to performing any communication episode, stands out to be the best option in all the situations considered. This is the algorithm that requires less communication episodes and, consequently, uses bigger buffers to accumulate information. This means that with the PSD-b algorithm there is a longer withholding of data in each CPU before any communication occurs. Far from increasing the idle time, the results show that, with this strategy, the solvable tasks are completed faster and, as a consequence, the *algorithm blocking* (and thus the idle time) gets reduced.

The PSD-b algorithm differs from the algorithms described in [1, 2] in that it is not based on reordering/prioritizing the tasks on the *solvable* list. At each stage of the PSD-b algorithm the information to be sent is held back in order to be accumulated in only one message. Thus, the order in which the elements are solved and stored in the buffer is irrelevant, because they all arrive at the same time to the receiver CPU. Therefore, in the PSD-b algorithm, the sweeping order does not affect the idle time. Furthermore, in this case, the best option is to use an order which contributes to the efficient solution of the tasks. In this regard, our strategy is to group the tasks by ordinates, avoiding jumps form one ordinate to the other that produce memory overhead.

The numerical experiments carried out on the MareNostrum supercomputer highlight a notable parallel performance for the PSD-b algorithm. We have obtained superlinear strong speedup using up to 2560 CPUs for a discretization of $151k$ nodes and 80 ordinates in the spatial and angular subdomains, respectively. This corresponds to a load of about 4800 unknowns/CPU for 2560 CPUs. The weak speedup tests show a degradation of the solution time of only 2.31 times when the size of the problem and the number of CPUs are increased 160 times (up to 2560 CPUs), keeping the ratio unknowns/CPU at around $94k$ elements. We have asserted that these results improve when the number of ordinates grows. The S_4, S_8 and S_{12} ordinate sets have been used for this purpose. Finally, we have compared the performance of the algorithms in different geometries, asserting that the PSD-b performs the best in all the situations tested.

A direct comparison between the results obtained by different authors is not easy because there are some points of uncertainty, as the differences in the hardware or in the implementation. The works by Plimpton et al. [1] and Pautz [2], which propose reordering/prioritizing strategies in order to improve the parallel performance of a basic sweep algorithm, have been used as a reference in this chapter. Our starting point is a similar basic algorithm. The improvement achieved by the PSD-b represents between 50 and 100 percentage points in the parallel performance, depending on the number of CPUs and the size of the problem, while the heuristics presented by Pautz and Plimpton et al. represent improvements that do not exceed 30 percentage points. With this scenario in mind, the PSD-b algorithm is a feasible option to be considered when solving the BTE.

References

[1] Steven J. Plimpton, Bruce Hendrickson, Shawn P. Burns, William McLendon III, and Lawrence Rauchwerger. Parallel S_n Sweeps on Unstructured Grids: Algorithms for Priorization, Grid Partitioning and Cycle Detection. *Nuclear*

Science and Engineering, 150:267–283, 2005.

[2] Shawn D. Pautz. An Algorithm for Parallel S_n Sweeps on Unstructured Meshes. *Nuclear Science and Engineering*, 140:111–136, 2002.

[3] Eric E. Aubanel and Faysal El Khettabi. Parallelization of Radiation Transport on Unstructured Triangular Grids with Spatial Decomposition and OpenMP. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS (2002).

[4] Eric E. Aubanel and Faysal El Khettabi. Parallelization of the Three-Dimensional Transport Equation for Boron Neutron Capture Therapy. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS (2003).

[5] Chunye Gong, Jie Liu, Lihua Chi, Haowei Huang, Jingyue Fang, and Zhenghu Gong. GPU accelerated simulations of 3D deterministic particle transport using discrete ordinates method. *J. Comput. Phys.*, 230(15):6010–6022, July 2011.

[6] William F. Godoy and Xu Liu. Introduction of parallel GPGPU acceleration algorithms for the solution of radiative transfer. *Numerical Heat Transfer, Part B*, 59(1):1–25, January 2011.

[7] P. J. Coelho and J. Gonçalves. Parallelization of the finite volume method for radiation heat transfer. *Int. J. Num. Meth. Heat Fluid Flow*, 9(4):388–404, 1999.

[8] James W. Fischer and Y.Y. Azmy. Comparison via parallel performance models of angular and spatial domain decompositions for solving neutral particle transport problems. *Progress Nucl. Ener.*, 49(1):37–60, January 2007.

[9] W. A. Fiveland. Discrete-ordinates solutions of the radiative transport equation for rectangular enclosures. *Journal of Heat Transfer*, 106:699–706, November 1984.

[10] G. D. Raithby and E. H. Chui. A finite volume method for predicting radiant heat transfer in enclosures with participating media. *Journal of Heat Transfer*, 112:415–423, 1990.

[11] J. Y. Murthy and S. R. Mathur. Finite Volume Method for Radiative Heat Transfer using Unstructured Meshes. *Journal of Thermophysics and Heat Transfer*, 12(3):313–321, 1998.

[12] J. Liu, H. M. Shang, Y. S. Chen, and T. S. Wang. Prediction of radiative transfer in general body-fitted coordinates. *Numerical Heat Transfer, Part B*, 31(4):423–439, 1997.

[13] J. Liu, H. M. Shang, and Y. S. Chen. Development of an unstructured radiation model applicable for two-dimensional planar, axisymmetric, and three-dimensional geometries. *J. Quant. Spectrosc. Radiat. Transfer*, 66(1):17–33, July 2000.

[14] W. An, L. M. Ruan, H. Qi, and L. H. Liu. Finite element method for radiative heat transfer in absorbing and anisotropic scattering media. *J. Quant. Spectrosc. Radiat. Transfer*, 96(3–4):409–422, December 2005.

[15] Shawn P. Burns and Mark A. Christon. Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications. *Numerical Heat Transfer, Part B*, 31(4):401–421, 1997.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[17] R. S. Baker and R. E. Alcouffe. Parallel 3-D SN performance for DANTSYS/MPI on the CRAY T3D. In *In Proc. of the Joint International Conference on Mathematical Methods and Supercomputing for Nuclear Applications*, volume 1, page 337, 1997.

[18] R.S. Baker and K.R. Koch. An s_n algorithm for the massively parallel cm-200 computer. *Nuclear Science and Engineering*, 128(3):312–320, 1998.

[19] Teresa S. Bailey and Robert D. Falgout. Analysis of massively parallel discrete-ordinates transport sweep algorithms with collisions. In *Int. conf. on mathematics, computational methods & reactor physics*, Saratoga Springs, NY, May 2009.

[20] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *Intl. J. High Performance Computing Applications*, 14(4):330–346, 2000.

[21] Mark M. Mathis and Darren J. Kerbyson. A general performance model of structured and unstructured mesh particle transport computations. *J. of Supercomputing*, 34(2):181–199, November 2005.

[22] V.S. Anil Kumar, Madhav V. Marathe, Srinivasan Parthasarathy, Aravind Srinivasan, and Sibylle Zust. Provable algorithms for parallel generalized sweep scheduling. *J. Parallel Distrib. Comput.*, 66:807–821, 2006.

[23] K.M. Case and P.F. Zweifel. Existence and Uniqueness Theorems for the Neutron Transport Equation. *Journal of Mathematical Physics*, 4(11):1376–1385, November 1963.

[24] Michael F. Modest. *Radiative Heat Transfer*. McGraw Hill, 1993.

[25] Götz, T. Coupling heat conduction and radiative transfer. *J. Quant. Spectrosc. Radiat. Transfer*, 72:57–73, 2002.

[26] William F. Godoy and Paul E. DesJardin. On the use of flux limiters in the discrete ordinates method for 3D radiation calculations in absorbing and scattering media. *J. Comput. Phys.*, 229(9):3189–3213, May 2010.

[27] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[28] Y. Y. Azmy and D. A. Barnett. Arbitrarily High Order Transport Method of the Characteristic Type for Tetrahedral Grids. In *ANS International Meeting on Mathematical Methods for Nuclear Applications*, 2001.

[29] K. D. Lathrop and Bengt G. Carlson. Discrete ordinates angular quadrature of the neutron transport equation. Technical Report LA-3186, Los Alamos Scientific Laboratory, February 1965.

[30] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[31] Guillem Colomer. *Numerical methods for radiative heat transfer*. PhD thesis, Universitat Politècnica de Catalunya, 2006.

4

Conclusions and ongoing research

4.1 Concluding remarks

In the abstract of this thesis, high performance computing (HPC) is referred as an essential tool to extend the capabilities of Computational Fluid Dynamics (CFD) on the resolution of complex flows for “real live” applications. Regarding to such issue some contributions have been presented.

Design and implementation of a CFD code

One of the main goals of this thesis, has been to contribute at the development of a general purpose multi-physics parallel CFD code, referred to as TermoFluids (TF). Particularly, the work has been focused on the development of two of the most basic libraries that compose TF. The first one is the Basics Object Library (BOL), which is an unstructured CFD application programming interface, that supports the basic finite-volume geometrical and algebraic operations in parallel. On the top of the BOL the rest of libraries that compose TF code have been developed. Afterwards, the Linear Solvers Library (LSL) was created for the solution of the linear systems of equations arising from the discretizations. The solution of these systems is one of the most time-consuming parts of CFD simulations, therefore, it has an important influence on the overall performance. Two application-specific solvers included in this library are described in the following chapters of this thesis.

The BOL and LSL libraries have been designed following the intuitive object oriented paradigm of the C++ language, that allows to expand the code in an orderly and compact form. The classes representing the main concepts treated by the libraries have also been introduced. An implementation for the solution of a Poisson equation on the top of the BOL and the LSL, has also been presented as an example of their user-friendly intuitive design. A proof of it is also the rapid expansion of TF to a multi-physics CFD code, which involves several developers working simultaneously.

Parallelism is another basic design feature of the code. TF is mainly programmed following the distributed memory paradigm. Basic concepts such as the domain decomposition, the definition of halo elements, the local and global identifiers or the communication schemes, have been described in detail. The implications of the parallelism in the design and applicability of linear solvers have also been briefly discussed.

Finally, some industrial problems, in which TF code has been an important tool to better understand the physics under study and improve the industrial designs, have been presented.

FFT-based Poisson solver

A parallel direct algorithm for the solution of the Poisson equation arising in incompressible flows with one periodic direction has been presented in the second chapter of this thesis. It is a combination of a direct Schur-complement based decomposition (DSD) and a Fourier diagonalization. The latter decomposes the original system into a set of mutually independent 2D subsystems which are solved by means of the DSD algorithm. Since no restrictions are imposed in the non-periodic directions, the overall algorithm is well suited for solving problems discretized on extruded 2D unstructured meshes.

The parallelization is based on a geometric domain decomposition. Different partitions are employed for the FFT-based change-of-basis (from *physical* to *spectral* space and vice versa) and for the solution of the 2D subsystems. The former operation must be performed without partitioning the mesh in the periodic direction whereas, for the latter, the number of processes to solve each 2D subsystem must be kept in the range of linear scalability of the DSD algorithm. Despite the additional transmissions of data between these two partitions, this strategy benefits the scalability of the overall algorithm.

The scalability and efficiency of the proposed method has been shown by performing several numerical experiments on the MareNostrum Supercomputer. Scalability tests using up to 8192 parallel processes with up to 10^9 million nodes meshes have demonstrated the algorithm capability on solving large-scale problems with a very short time. Moreover the performance DSD algorithm as 2D solver has been successfully compared with the preconditioned conjugate gradient method. Finally, some illustrative applications of the solver for the direct numerical simulation of flows around bluff bodies have been outlined.

Sn transport sweep algorithms

On the third chapter, a solver for the Boltzmann Transport Equation (BTE) has been presented. It can be used to solve radiation phenomena interacting with flows. The Discrete Ordinates Method and a first-order upwind like scheme are used for the discretization of the angular and spatial subdomains, respectively. Under these conditions, a *sweeping based* direct method, which consists on sweeping the flux across the grid from upstream to downstream nodes, is a suitable choice to solve the spatial couplings for each ordinate, while the coupling between diferent ordinates are accounted by means of a source iteration algorithm. The parallelization is carried out by means of a spatial domain decomposition strategy. Its main drawback is the idle time produced when a CPU requires information from upstream nodes belonging to other CPUs to be able to proceed. This handicap is overcome by overlapping the calculations of the flux at some ordinates with the waiting for upstream values at the others.

Different variants of sweep based algorithms have been presented, among them the PSD-b algorithm, which consists of completing all the solvable tasks for all ordinates prior to performing any communication episode, stands out to be the best option in all the situations considered. The numerical experiments carried out in the MareNostrum supercomputer highlight a notable parallel performance for the PSD-b: i) a superlinear strong speedup using up to 2560 CPUs for a rather small discretization of $151k$ nodes and 80 ordinates, and ii) a slowdown of only 2.31 times when the size of the problem and the number of CPUs are increased 160 times (up to 2560 CPUs), keeping the ratio unknowns/CPU at around $94k$ elements. These results improve when the number of ordinates of the angular discretization grows. Finally, the performance of the algorithms is compared in different geometries, asserting that the PSD-b also performs the best in all the situations tested.

A direct comparison between the results obtained by different authors is not easy because there are some points of uncertainty, as the differences in the hardware or in the implementation. The works by Plimpton et al. (Nuclear Science and Engineering 150 (2005) 267) and Pautz (Nuclear Science and Engineering 140 (2002) 111), which propose reordering/prioritizing strategies in order to improve the parallel performance of a basic sweep algorithm, have been used as a reference in this chapter. Our starting point is a similar basic algorithm. The improvement achieved by the PSD-b represents between 50 and 100 percentage points in the parallel performance, depending on the number of CPUs and the size of the problem, while the heuristics presented by Pautz and Plimpton et al. represent improvements that do not exceed 30 percentage points. With this scenario in mind, the PSD-b algorithm is a feasible option to be considered when solving the BTE.

4.2 Ongoing research

Some other research topics in which I am involved along with my colleagues in the CTTC are:

4.2.1 Sweep based preconditioners for radiation transport

We are concerned with the parallel solution of the Radiation Transport Equation on unstructured grids with an MPI-based spatial domain decomposition approach. For the angular coupling, an alternative strategy to the source iteration (SI) method is being considered. Instead of decoupling the different angular ordinates, the RTE is discretized into one single matrix where are represented both the spatial and angular couplings. This system is solved by means of a preconditioned Krylov subspace method, using the sweep based algorithm described in the third chapter of the present thesis as a block diagonal preconditioner. This methodology has been tested on several configurations showing its benefits respect to the classical SI strategy, specially in cases with high scattering and reflectivity coefficients. More details of this work can be found in [1].

4.2.2 Hybrid MPI-CUDA approximate inverse preconditioners

The recent incorporation of GPU as coprocessors in supercomputer nodes has provoked the study and the implementation of Krylov methods into the GPU paradigm, however, due the recent character of this technology there are fields, such as GPU-oriented Krylov preconditioning, that are not deeply studied yet. Actually, we are exploring the aspects of the Approximate Inverse Preconditioner (AIP) on GPUs. Specifically the development of an Hybrid MPI-CUDA krylov solver preconditioned with an AIP. For the matrix vector products, we use the routines of the CUSPARSE library, and to compute other vector operations, such as dot products, vector additions or scalar multiplications, the CUBLAS library is used. These libraries, provided by NVIDIA, have been created to take advantage of the CUDA parallel paradigm on a single GPU, however our main objective is to be able to use multiple GPUs connecting them throw an MPI interface.

Both CG and AIP present the sparse matrix vector multiplication as the most time consuming part of the algorithm. The main bottle neck of the parallelization with the hybrid MPI-CUDA approach are the MPI communications that imply three stages: 1) download data from the GPU to the CPU, ii) communicate data between the different CPUs by means of MPI routines, iii) upload the data back from the CPU to the GPU.

Some strategies to minimize the degradation produced by this communication episodes have been studied. Numerical experiments carried out in the TGCC Curie supercomputer (see Appendix B), showed that the most successful strategy, of which were tested, consists on splitting the system matrix in two parts in order to overlap operations on the GPU with transmission of data between different devices. Further details can be found in [2].

4.2.3 Parallelization of the Volume-of-Fluid method for 3D unstructured meshes

The Volume-of-Fluid (VOF) is one of the most widely used methods for interface tracking on fixed grids. The interface between different fluids is generated from the volume fraction scalar fields, which account for the ratio of volume of each fluid in each control volume. In the cells where two fluids coexist, an interface geometry is reconstructed to divide them according to the values of the volume fraction fields. Then, the volume fraction advection equation is solved to obtain the new distributions of the fluids after momentum is applied. Since this is a time-consuming process, parallelization techniques play an essential role.

All the VOF methods have in common that most of computational cost of the algorithm is concentrated in operations with the cells that form the interface. For this reason, if the interface is not homogeneously distributed throughout the domain, the standard domain decomposition strategy results in an unbalanced method.

A possible strategy to overcome this problem consists in adapting the domain partition to the interface distribution. This strategy already proposed by different authors such as Walshaw et al. (Applied Mathematical Modelling 25 (2000) 123), has a number of drawbacks: i) the general distribution of the interface may not be known a priori, ii) the distribution of the interface can vary during the simulation, requiring the readaptation of the mesh partition, iii) in multi-physics simulations, the particular partition that is good for the VOF solver, may not be appropriate for other parts of the algorithm such as the momentum equation [3].

The new strategy here developed consist on dividing the work of the interface cells between all CPUs without taking into account the distribution given by the domain partition. CPUs exchange information necessary to perform the VOF operations in order to obtain a similar workload. Some aspects must be considered in this process, for example, the solution of the interface elements moved between CPUs has an additional cost, due to the communications, that must be taken into account to balance the work properly. Results show that the new parallelization strategy has a qualitatively better parallel performance than the standard domain decomposition approach. More details of this VOF parallelization strategy can be found in [4].

4.2.4 Parallel radial basis function interpolation methods for unstructured dynamic meshes

In a great amount of engineering applications it is necessary to resort to dynamically updated meshes. Some of these applications are moving boundary problems, bio-fluid mechanics problems (e.g. blood flow through veins and arteries), fluid-structure interaction (e.g. flutter simulation of wings) and optimization search, where the geometry of the object being studied is modified in order to find the best design. To attempt the numerical simulation of these phenomena, we need a robust, accurate and fast method, which redistributes the mesh in accordance with the movement of the domain. Roughly speaking there are two categories of methods: the methods based on mesh regeneration and on mesh deformation, respectively. Grid regeneration is generally much more time-consuming, and does not preserve the topological connectivity of the mesh elements. For this reason, numerous researchers have been interested in developing efficient moving mesh algorithms.

This study is focused on the moving grid techniques to be applied in the field of the computational fluid dynamics (CFD). Among all the possible strategies available to deform a mesh during a simulation, the method finally adopted often depends on the mesh type and the application. Thus, the main challenge would be finding a general technique, being suitable for all type of meshes and physical situations, which preserves the quality of the mesh and keeps the computational resources affordable. Furthermore, to ensure maximum efficiency this technique should be of easy implementation in a parallel environment.

In this work a radial basis function (RBF) interpolation method has been implemented to be applied in CFD problems with dynamic meshes. This method has been tested with two challenging examples of dynamic meshing: the deformation of a pitching airfoil and a three-dimensional movement of a sphere, both discretized over viscous grids of around 5 M control volumes. The work also includes a qualitative comparison of the RBF interpolation and the classical spring analogy formulations, which asserts that the new approach is far less costly and, besides, can achieve a good performance in preserving the mesh quality. In addition, the dynamic mesh adaptation has been coupled with a CFD solver, what has been validated on a benchmark problem consisting on a duct with a moving indentation. Finally, it has been analyzed the parallel performance of the algorithm for the case of the deformable sphere, pointing out some key aspects that must be considered in order to improve the parallelization. Further details can be found in [5, 6].

References

[1] R. Borrell, G. Colomer, and A. Oliva. Sweep based preconditioner for radiation transport. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[2] G. Oyarzun, R. Borrell, O. Lehmkuhl, and A. Oliva. Hybrid MPI-CUDA approximate inverse preconditioner. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[3] Ll. Jofre, O. Lehmkuhl, R. Borrell, J. Castro, and A. Oliva. Parallelization study of a VOF/Navier-Stokes model for 3D unstructured staggered meshes. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

[4] Ll. Jofre, R. Borrell, O. Lehmkuhl, and A. Oliva. Parallelization of the Volume-of-Fluid method for 3D unstructured meshes. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

[5] O. Estruch, R. Borrell O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. A parallel radial basis function interpolation method for unstructured dynamic meshes. *Computers and Fluids*, (in press).

[6] O. Estruch, R. Borrell, O. Lehmkuhl, and C.D. Pérez-Segarra. A Parallel Radial Basis Function Interpolation Method for Unstructured Dynamic Meshes. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

Appendix A

Circulant matrices and its Fourier diagonalization

Circulant matrices are diagonalizable in the Fourier space of the same dimension. Therefore linear systems with circulant matrices can be solved very easily in the corresponding Fourier space. Moreover, the change of basis necessary to transform the right-hand-side term to the Fourier space and, after solving the diagonal system, get the solution back to the Cartesian space, can be carried out by means of a Fast Fourier Transformation (FFT) algorithm, which has $O(N\log(N))$ instead of the $O(N^2)$ of a dense product. These good properties of circulant systems are used in this thesis, in order to accelerate the solution of the Poisson system in discretizations with one homogeneous periodic direction, because the couplings in such directions result in circulant matrices. In this appendix, these useful properties of circulant matrices are explicitly demonstrated (further details can be found in [1,2]).

A.1 Introductory Definitions and Properties

A.1.1 Circulant matrices

Definition A.1. A circulant matrix of order n, is a square matrix of the form

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & & \cdots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & c_2 & & \vdots \\ & c_{n-1} & c_0 & c_1 & \ddots & \\ \vdots & & \ddots & \ddots & \ddots & c_2 \\ & & & & & c_1 \\ c_1 & \cdots & & & c_{n-1} & c_0 \end{bmatrix},$$

where each row is a cyclic shift of the row above it. Using a more compact notation, C is referred as $C = circ(c_0, c_1, ..., c_{n-1})$.

Example: The *one - dimensional* second order discretization of the Poisson equation, in a mesh with constant step Δ_x, and with periodic boundary conditions, is a circulant matrix of the form

$$\mathsf{L}_x = \frac{1}{\Delta_x}\begin{bmatrix} 2 & -1 & 0 & & \cdots & -1 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ & & & & & -1 \\ -1 & \dots & & 0 & -1 & 2 \end{bmatrix}.$$

Thus $\mathsf{L}_x = \frac{1}{\Delta_x} circ(2, -1, 0, ...0, -1)$.

A.1.2 Fourier matrix

Definition A.2. Let $n \geq 1$ be a fixed integer and $w = \exp(-\dfrac{2\Pi i}{n})$ the primitive n^{th} root of unity. The Fourier matrix of order n is the matrix F_n defined as

$$\begin{aligned} \mathsf{F}_n &= \frac{1}{\sqrt{n}}\left(w^{ij}\right)_{i,j=0,\dots,n-1} = \\ &= \frac{1}{\sqrt{n}}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ 1 & w^3 & w^6 & \cdots & w^{3(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix}. \end{aligned}$$

Note that the matrix is normalized by $\frac{1}{\sqrt{n}}$ to make it unitary. The Fourier matrix is the matrix of the Discrete Fourier Transform (DFT).

Lemma A.1. If $x = (x_0, ..., x_{n-1}) \in \mathbb{R}^n$ and $\tilde{x} = (\tilde{x}_0, ..., \tilde{x}_{n-1}) \in \mathbb{C}^n$ such that

$$\tilde{x} = \mathsf{F}_n x$$

then

$$\tilde{x}_k = \tilde{x}^*_{n-k}, \qquad k = 1..., \left\lfloor \frac{n-1}{2} \right\rfloor,$$

where $\lfloor . \rfloor$ is the floor function.

Proof: For any $k \in \{1, ..., \lfloor \frac{n-1}{2} \rfloor\}$, the $k'th$ n-root of unity $w^k = \exp(-\dfrac{2\Pi ik}{n})$ satisfies the nest property:

$$w^k = (w^{n-k})^*$$

Then,

$$\tilde{x}_k = \sum_{j=0}^{n-1} x_j (w^k)^j = \sum_{j=0}^{n-1} x_j ((w^{n-k})^*)^j = \left(\sum_{j=0}^{n-1} x_j (w^{n-k})^j \right)^* = \tilde{x}^*_{n-k}.$$

□

A.2 Diagonalization

Theorem A.1. The eigenvectors of a circulant matrix $C = circ(c_0, ..., c_{n-1})$ of dimension n are the columns of F_n, the Fourier matrix of the same size. And the eigenvalues are

$$\lambda_k = [\mathsf{F}_n (c_0, ..., c_{n-1})^T]_k = \sum_{j=0}^{n-1} c_j (w^k)^j \qquad k = 0, ..., n-1. \tag{A.1}$$

Proof: The eigenvalues λ_k and the eigenvectors $v^{(k)}$ of $C = circ(c_0, ..., c_{n-1})$ are the solutions of the system

$$Cv = \lambda v \tag{A.2}$$

or, equivalently, of the difference equations

$$\sum_{k=0}^{m-1} c_{n-m+k} v_k + \sum_{k=m}^{n-1} c_{k-m} v_k = \lambda v_m; \qquad m = 0, 1, ..., n-1. \tag{A.3}$$

Changing the summation dummy variable results in

$$\sum_{k=0}^{n-1-m} c_k v_{k+m} + \sum_{k=n-m}^{n-1} c_k v_{k-(n-m)} = \lambda v_m; \qquad m = 0, 1, ..., n-1 \tag{A.4}$$

If the last expression evaluated in the vector $v = (1, \rho, \rho^2, ..., \rho^{n-1})$, where ρ is one of the n distinct complex n^{th} roots of the unity, the result is

$$\sum_{k=0}^{n-1-m} c_k \rho^{k+m} + \sum_{k=n-m}^{n-1} c_k \rho^{k-(n-m)} = \lambda \rho^m; \qquad m = 0, 1, ..., n-1. \tag{A.5}$$

Taking common factor ρ^m, results in

$$\left(\sum_{k=0}^{n-1-m} c_k \rho^k + \sum_{k=n-m}^{n-1} c_k \rho^{k-n} \right) \rho^m = \lambda \rho^m; \qquad m = 0, 1, ..., n-1. \tag{A.6}$$

Thus,

$$\begin{aligned}\lambda &= \sum_{k=0}^{n-1-m} c_k\rho^k + \sum_{k=n-m}^{n-1} c_k\rho^{k-n} = &\text{(A.7)}\\ &= \sum_{k=0}^{n-1-m} c_k\rho^k + \rho^{-n}\sum_{k=n-m}^{n-1} c_k\rho^k = &\text{(A.8)}\\ &= \sum_{k=0}^{n-1} c_k\rho^k &\text{(A.9)}\end{aligned}$$

Therefore, the vector $v = (1, \rho, \rho^2, ..., \rho^{n-1})$ is an eigenvector of C with eigenvalue $\lambda = \sum_{k=0}^{n-1} c_k\rho^k$. Replacing ρ by the different roots of the unity the theorem is proved. □

Corollary A.1. If $C = circ(c_0, ..., c_{n-1})$,

$$\mathsf{F}_n^* C \mathsf{F}_n = \Lambda$$

where $\Lambda = diag(\lambda_0, \lambda_1, ..., \lambda_{N_{per}-1})$ and $\lambda_k = [\mathsf{F}_n(c_0, ..., c_{n-1})^T]_k$. □

Corollary A.2. If $C = circ(c_0, ..., c_{n-1})$. The solution of the linear system of equations

$$Cx = b$$

is

$$x = \mathsf{F}_n \Lambda^{-1} \mathsf{F}_n^* b$$

where $\Lambda = diag(\lambda_0, \lambda_1, ..., \lambda_{N_{per}-1})$ and $\lambda_k = [F_n(c_0, ..., c_{n-1})^T]_k$.
Proof:

$$Cx = b \implies \mathsf{F}_n^* C \mathsf{F}_n \mathsf{F}_n^* x = \mathsf{F}_n^* b \implies \Lambda \mathsf{F}_n^* x = \mathsf{F}_n^* b \implies x = \mathsf{F}_n \Lambda^{-1} \mathsf{F}_n^* b$$

□

Fast Fourier Transform

Computing a DFT of dimension N in the naive way, as a matrix-vector product, takes $O(N^2)$ arithmetical operations, however using an FFT algorithm, which recursively decomposes the problem, the same result can be computed in only $O(Nlog(N))$ operations. Several general purpose libraries, such as [3], provide optimized implementations of FFT algorithms. This good property of the DFT is also of major importance to accelerate of the solution of cirtulant systems.

References

[1] R. M. Gray. Toeplitz and Circulant Matrices: A review. *Foundations and Trends in Communications and Information Theory*, 2:155–239, 2006.

[2] P. J. Davis. *Circulant Matrices.* Wiley-Interscience, New York, 1979.

[3] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Appendix B

Computing resources

This appendix lists different parallel computing systems where the codes developed in this thesis have been executed. The access to most of these equipments is obtained gaining competitive calls, in which the evaluated criteria are both the scientific relevance of the presented projects and the capability of the software to obtain a good parallel performance with the offered resources.

B.1 "Old" JFF cluster, Terrassa

The first "Beowulf cluster" at CTTC was configured in 1999. It was called JFF in memorial to Joan Francesc Fernández, a computer science professor at the Universitat Politècnica de Catalunya. He brought the first computer to our faculty and awake the interest for numerical simulation to many persons.

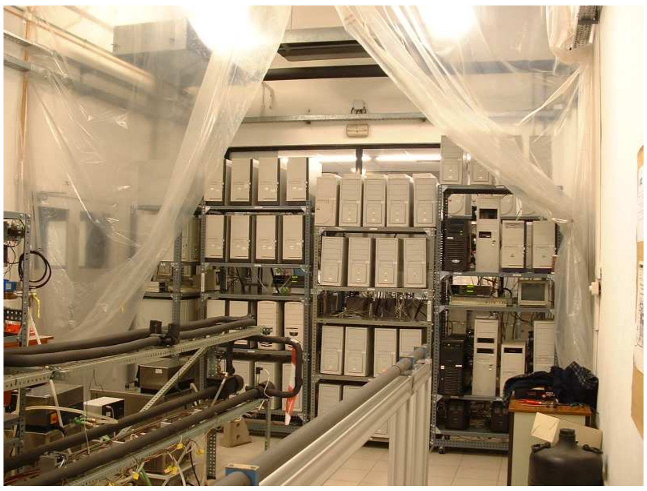


Figure B.1: Front view of the "old" JFF cluster

The initial JFF system, in 1999, was a 16 nodes machine, with AMD K7 processors at 600 MHz and a 100 Mbits/s switch. At that moment, it was one of the firsts Beowulf systems in Spain. It was updated several times and it grew up to a partition of 100 Core Duo CPUs at 2.2GHz. The nodes were interconnected via a gigabit Ethernet network and each node had 4Gb of RAM. From 2010 this cluster was retired, because it was replaced by the more efficient JFF supercomputer.

B.2 JFF supercomputer, Terrassa

Figure B.2: JFF supercomputer

JFF supercomputer consists in 168 computer nodes with 2304 cores and 4.6 TB of RAM in total. There are 128 nodes with 2 AMD Opteron 2376 quad-core processors at 2.3GHz and 16 GB of RAM linked with the infiniband DDR 4X network, and 40 nodes with 2 AMD Opteron 6272 16-core processors at 2.1 GHz and 64 GB RAM linked with the infiniband QDR 4X network.

B.3 MareNostrum supercomputer, Barcelona

MareNostrum supercomputer, from the Barcelona Supercomputing Center (BSC), is an IBM BladeCenter JS21 Cluster composed of 2560 nodes. Each node contains 2 dual-core PowerPC 970MP processors at 2.3 GHz with 8 GB of RAM. Nodes are linked by means of a high-performance Myrinet network. In total, the supercomputer has 10240 cores and 20 TB of RAM.

Figure B.3: MareNostrum supercomputer

B.4 Magerit supercomputer, Madrid

Figure B.4: Magerit supercomputer

Magerit supercomputer, from the Centro de Supercomputación y Visualización de Madrid (CeSViMa), is a cluster with 4160 cores and 9.2 TB of RAM installed in 260 computer nodes. There are 245 nodes eServer BladeCenter PS702 2S with 16 Power7 processors at 3.3 GHz and 32 GB of RAM, and 15 nodes eServer BladeCenter HS22 with 8 Intel Xeon 2.5 GHz processors and 96 GB of RAM.

B.5 Lomonosov supercomputer, Moscow

Lomonosov supercomputer, from the Research Computing Center in the Moscow State University, is a supercomputer with $35,360$ cores and 60 TB of RAM installed in 5100 nodes. There are 4420 nodes with 2 quad-core Intel EM64T Xeon 5570 at 2930 MHz and and 12 GB RAM, and 680 nodes with 2 6-core Intel EM64T Xeon 5670 at 2930 MHz and 12 GB of RAM. The interconnection network is an Infiniband QDR.

Figure B.5: Lomonosov supercomputer

B.6 K100 supercomputer, Moscow

K100 supercomputer, from the Keldysh Institute of Applied Mathematics of the Russian Academy of Science (KIAM RAS), is composed of 64 nodes with 2 6-core Intel Xeon X5670 at 2930 MHz, 3 NVIDIA Fermi 2050 and 96 GB of RAM each one. Implying in total 768 CPU cores, 192 GPUs and 6 TB of RAM. The interconnection of nodes is carried out by means of an Infiniband QDR networks and a PCI-Express.

Figure B.6: K100 supercomputer

B.7 Curie supercomputer, Paris

Figure B.7: Curie supercomputer

The Curie supercomputer, owned by the Grand Equipement National de Calcul Intensif (GENCI) and operated into the Très Grand Centre de Calcul (TGCC) by the Commissariat à l'énergie atomique (CEA), is the first French Tier0 system open to scientists through the French participation into the Partnership for Advanced Computing in Europe (PRACE) research infrastructure.

Curie is offering 3 different partitions of x86-64 computing resources, in the context of this thesis the “Curie fat nodes” and “Curie hybrid nodes” have been used. The fat nodes cluster is composed of 360 S6010 bullx nodes, with 4 eight-core Nehalem-EX X7560 CPUs at 2.26 GHz and 128 GB of RAM. The hybrid nodes cluster is composed of 16 bullx B chassis with 9 hybrid GPUs B505 blades, in each blade there are 2 quad-core Intel Westmere 2.66 GHz and 2 Nvidia M2090 T20A, in total 1152 CPU cores and 288 GPUs. In both cases and InfiniBand QDR Full Fat Tree network is used for the interconnection of nodes.

Appendix C

Main publications in the context of this thesis

International Journal Papers:

- G. Colomer, R. Borrell, F.X. Trias, and I. Rodríguez. Parallel algorithms for S_n transport sweeps on unstructured meshes. *Journal of Computational Physics*, (in press).
- O. Estruch, O. Lehmkuhl, R. Borrell, C.D. Pérez-Segarra, and A. Oliva. A parallel radial basis function interpolation method for unstructured dynamic meshes. *Computers and Fluids*, (in press).
- I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Flow dynamics in the turbulent wake of a sphere at sub-critical Reynolds numbers. *Computers and Fluids*, (in press).
- O. Lehmkuhl, R. Borrell, I. Rodríguez, C.D. Pérez-Segarra, and A. Oliva. Assessment of the symmetry-preserving regularization model on complex flows using unstructured grids. *Computers and Fluids*, 60:108–116, 2012.
- R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva. Parallel direct Poisson solver for discretisations with one Fourier diagonalisable direction. *Journal of Computational Physics*, 230(12):4723–4741, 2011.
- I. Rodríguez, R. Borrell, O. Lehmkuhl, C.D. Pérez-Segarra, and A. Oliva. Direct numerical simulation of the flow over a sphere at Re = 3700. *Journal of Fluid Mechanics*, 679:263–287, 2011.
- A. Gorobets, F. X. Trias, R. Borrell, O. Lehmkuhl, and A. Oliva. Hybrid MPI+OpenMP parallelization of an FFT-based 3D Poisson solver with one periodic direction. *Computers and Fluids*, 49(1):101–109, 2011.

International Conference Papers:

- J. Chiva, O. Lehmkuhl, R. Borrell, and A. Oliva. Direct Numerical Simulation of the turbulent natural convection flow in an open cavity of aspect ratio 4. In *7th Symposium on Turbulence, Heat and Mass Transfer, THMT-12*, Palermo (Italy), September 2012.
- O. Estruch, O. Lehmkuhl, R. Borrell, and C.D. Pérez-Segarra. Large-eddy simulation of turbulent FSI. In *7th Symposium on Turbulence, Heat and Mass Transfer, THMT-12*, Palermo (Italy), September 2012.
- I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Low-frequency unsteadiness in the vortex formation region of a circular cylinder. In *7th Symposium on Turbulence, Heat and Mass Transfer, THMT-12*, Palermo (Italy), September 2012.
- I. Rodríguez, O. Lehmkuhl, A. Baez, R. Borrell, and A. Oliva. Direct numerical simulation of a NACA 0012 in full stall. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.
- D. Aljure, O. Lehmkuhl, R. Borrell, and A. Oliva. Flow and turbulent structures around simplified car models. In *Conference on Modeling Fluid Flow*, Budapest (Hungary), September 2012.
- R. Borrell, G. Colomer, and A. Oliva. Sweep based preconditioner for radiation transport. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.
- G. Oyarzun, R. Borrell, O. Lehmkuhl, and A. Oliva. Hybrid MPI-CUDA approximate inverse preconditioner. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.
- Ll. Jofre, R. Borrell, O. Lehmkuhl, and A. Oliva. Parallelization of the Volume-of-Fluid method for 3D unstructured meshes. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.
- O. Estruch, R. Borrell, O. Lehmkuhl, and C.D. Pérez-Segarra. A Parallel Radial Basis Function Interpolation Method for Unstructured Dynamic Meshes. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.
- O. Lehmkuhl, I. Rodríguez, R. Borrell, and A. Oliva. High-Performance computing of flows with massive separation: flow past a NACA 0012. In *Parallel Computational Fluid Dynamics*, Atlanta (USA), May 2012.

- R. Borrell, O. Lehmkuhl, F. X. Trias, G. Oyarzun, and A. Oliva. FFT-based Poisson solver for large scale numerical simulations of incompressible flows. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- G. Colomer, R. Borrell, O. Lehmkuhl, and C.D. Pérez-Segarra. Parallel algorithm for the solution of the Boltzmann transport equation on unstructured meshes. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- Ll. Jofre, O. Lehmkuhl, R. Borrell, J. Castro, and A. Oliva. Parallelization study of a VOF/Navier-Stokes model for 3D unstructured staggered meshes. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- O. Estruch, O. Lehmkuhl, R. Borrell, and C.D. Pérez-Segarra. A parallel three-dimensional radial basis function interpolation method for unstructured dynamic meshes. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- A. Gorobets, F. X. Trias, R. Borrell, M. Soria, and A. Oliva. Hybrid MPI+OpenMP parallelization of an FFT-based 3D Poisson solver that can reach $100,000$ CPU cores. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva. Flow dynamics in the turbulent wake of a sphere at sub-critical Reynolds numbers. In *Parallel Computational Fluid Dynamics*, Barcelona (Spain), May 2011.

- O. Lehmkuhl, I. Rodríguez, R. Borrell, C.D. Pérez-Segarra, and A. Oliva. Low frequency variations in the wake of a circular cylinder at $Re = 3900$. In *13th European Turbulence Conference*, Varsaw (Poland), September 2011.

- A. Gorobets, F. X. Trias, R. Borrell, M. Soria, and A. Oliva. Hybrid MPI+OpenMP parallelization of a Navier-Stokes solver for large-scale DNS. In *7th International Conference on Computational Heat and Mass Transfer*, Istanbul (Turkey), July 2011.

- G. Colomer, R. Borrell, F. X. Trias, and A. Oliva. Effect of mesh partition on the scalability of the parallel solution of the radiative transfer equation. In *7th International Conference on Computational Heat and Mass Transfer*, Istanbul (Turkey), July 2011.

- G. Colomer, R. Borrell, O. Lehmkuhl, and A. Oliva. Parallelization of combined convection-radiation numerical simulations. In 14*th International Heat Transfer Conference*, Washington D.C. (USA), Agust 2010.

- G. Colomer, R. Borrell, and A. Oliva. Parallel solution of the radiative transfer equation on unstructured meshes using an explicit solver. In 6*th International Symposium on Radiative Transfer*, Antalya (Turkey), June 2010.

- O. Lehmkuhl, R. Borrell, I. Rodríguez, C.D. Pérez-Segarra, and A. Oliva. On the symmetry-preserving regularization model of complex flows using unstructured grids. In *Fifth European Conference on Computational Fluid Dynamics ECCOMAS CFD*, Lisbon (Portugal), June 2010.

- I. Rodríguez, O. Lehmkuhl, R. Borrell, A. Oliva, and C.D. Pérez-Segarra. Direct numerical simulation of turbulent wakes: flow past a sphere at Re=5000. In *Fifth European Conference on Computational Fluid Dynamics ECCOMAS CFD*, Lisbon (Portugal), June 2010.

- A. Gorobets, R. Borrell, F. X. Trias, T. Kozubskaya, and A. Oliva. Efficiency of large-scale CFD simulations on modern superocomputers using thousands of CPUs and hybrid MPI+OPENMP parallelization. In *Fifth European Conference on Computational Fluid Dynamics ECCOMAS CFD*. Lisbon (Portugal), June 2010.

- O. Lehmkuhl, R. Borrell, C.D. Pérez-Segarra, A. Oliva, and R.W.C.P. Verstappen. LES modeling of the turbulent flow over an Ahmed car. In *DLES8, workshop on Direct and Large-Eddy Simulation*, Eindhoven (The Netherlands), July 2010. Springer.

- I. Rodríguez, O. Lehmkuhl, R. Borrell, and C.D. Pérez-Segarra. On DNS and LES of natural convection of wall-confined flows: Rayleigh-Bénard convection. In *DLES8, workshop on Direct and Large-Eddy Simulation*, Eindhoven (The Netherlands), July 2010. Springer.

- R. Borrell, O. Lehmkuhl, I. Rodríguez, C.D. Pérez-Segarra, and A. Oliva. Parallel Poisson solver for revolved unstructured grids. DNS of the flow around a sphere at Re=3700. In *Parallel Computational Fluid Dynamics*, Moffett Field, California (USA), May 2009.

- G. Colomer, O. Lehmkuhl, R. Borrell, and A. Oliva. CFD simulation of the thermal behavior of a wind mill power generator nacelle. In 6*th International Symposium on Turbulence, Heat and Mass Transfer*, Rome (Italy), September 2009.

- I. Rodríguez, R. Borrell, O. Lehmkuhl, A. Oliva, and C.D. Pérez-Segarra. CFD simulation of the thermal behavior of a wind mill power generator nacelle. In 6*th International Symposium on Turbulence Heat and Mass Transfer*, Rome (Italy), September 2009.

- O. Lehmkuhl, R. Borrell, C.D. Pérez-Segarra, J. Chivas, and A. Oliva. Direct numerical simulations and symmetry-preserving regularization simulations of the flow over a circular cylinder at Reynolds number 3900. In 6*th International Symposium on Turbulence Heat and Mass Transfer*, Rome (Italy), September 2009.

- R. Borrell, O. Lehmkuhl, F. X. Trias, M.Soria, and A. Oliva. Parallel direct Poisson solver for DNS of complex turbulent flows using unstructured meshes. In *Parallel Computational Fluid Dynamics*, Lyon (France), May 2008.

- R. Borrell, O. Lehmkuhl, M. Soria, and G. Colomer. Parallel Schur-Fourier decomposition for the efficient solution of the Poisson equation on massive extruded unstructured meshes. In *Fifth European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS*, Venice (Italy), July 2008.

- O. Lehmkuhl, R. Borrell, F. X. Trias, and C.D. Pérez-Segarra. Assessment of large-eddy simulation models in complex flows using unstructured meshes. In *Fifth European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS*, Venice (Italy), July 2008.

- G. Colomer, O. Lehmkuhl, R. Borrell, and R. Capdevila. Coupling radiation and convection: Effect of radiation mesh on both results and performance. In *Fifth European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS*, Venice (Italy), July 2008.

- O. Lehmkuhl, F. X. Trias, R. Borrell, and C.D. Pérez-Segarra. Symmetry-preserving Regularization modelling of a turbulent plane impinging jet. In *DLES7, workshop on Direct and Large-Eddy Simulation*, Trieste (Italy), July 2008.

- O. Lehmkuhl, C.D. Pérez-Segarra, R. Borrell, M. Soria, and A. Oliva. TERMOFLUIDS: A new Parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC Clusters. In *Parallel Computational Fluid Dynamics*, Antayla (Turkey), May 2007. Elsevier.

- R. Borrell, O. Lehmkuhl, M. Soria, and A. Oliva. Schur Complement Methods for the solution of Poisson equation with unstructured meshes. In *Parallel Computational Fluid Dynamics*, Antayla (Turkey), May 2007. Elsevier.