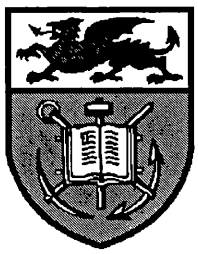
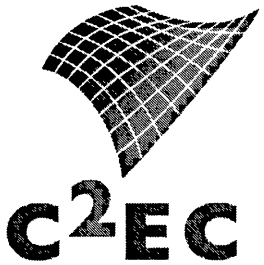
Unstructured parallel grid generation.

Said, Rajab





University of Wales Swansea
School of Engineering,
Civil & Computational Engineering Centre

Unstructured Parallel Grid Generation

by

Rajab Said

B.Sc., M.Sc.

Thesis submitted to the University of Wales in candidature for the degree of
Doctor of Philosophy

November 2003

Dedicated to

Zeinab and Sami,
who deserve the 'star' more than me

Acknowledgment

I would like to thank my supervisor, Professor Nigel Weatherill, for his expert advice and guidance throughout the course of this study; and 'extra' special thanks for his patience during the writing-up period. Also, many thanks are due to Dr. Richard Wood and Professor Javier Bonet for their great support and understanding during my work with them subsequent to completing the developmental work in this research. Help provided by support staff in the Civil & Computational Engineering Centre, Mrs A.E. Davies, Mrs. D. Cook and Mrs L. Jenkins, is highly appreciated.

I would like to thank the European Union for providing the funding for this research, through the JULIUS project (Esprit 25050). I also would like to gratefully acknowledge the partial financial support provided by the K.R.S. Foundation.

A special thanks is extended to all industrial partners who provided a number of exciting and challenging problems during this study, notably BAe Systems (UK), Dassault Aviation (France) and EADS (Germany). Also, I would like to express my deep gratitude to my friend Dr. Kaare Sorensen for providing the aerodynamic analysis results presented in this thesis.

I have a pleasure in thanking all those friends who have managed to 'brighten' Swansea up for us whenever it got 'gloomy'. Also, I wish to thank my 'old' friends back home and all over the world; your ever lasting friendship means a lot to me.

I would like to gratefully acknowledge the enormous support and love I have received from every body in my family: mum, dad, brothers and sisters and their families. I am equally grateful to my family-in-law for their love and kindness.

Finally, my deepest gratitude goes to my love Zeinab and my 'hero' Sami; though you have made things more difficult sometimes, this thesis could never be completed without your encouragement, understanding and endless love. I am, and will always be, indebted to you both.

Summary of the research

The ultimate goal of this study is *to develop a 'tool' by which large-scale unstructured grids for realistic engineering problems can be generated efficiently on any parallel computer platform.* The adopted strategy is based upon a geometrical partitioning concept, where the computational domain is sub-divided into a number of sub-domains which are then gridded independently in parallel. This study focuses on three-dimensional applications only, and it implements a Delaunay triangulation based generator to generate the sub-domain grids.

Two different approaches have been investigated, where the variations between them are limited to (i) the domain decomposition and (ii) the inter-domain boundary gridding algorithms only. In order to carry out the domain decomposition task, the first approach requires an initial tetrahedral grid to be constructed, whilst the second approach operates directly on the boundary triangular grid. Hence, this thesis will refer to the first approach as 'indirect decomposition method' and to the second as 'direct decomposition method'.

Work presented in this thesis also concerns the development of a framework in which all different sub-algorithms are integrated in combination with a specially designed parallel processing technique, termed as Dynamic Parallel Processing (DPP). The framework adopts the Message Passing Library (MPL) programming model and implements a Single Program Multiple Data (SPMD) structure with a Manager/Workers mechanism. The DPP provides great flexibility and efficiency in exploiting the available computing resources.

The framework has proved to be a very effective tool for generating large-scale grids. Grids of realistic engineering problems and to the order of 115 million elements, generated using one processor on an SGI Challenge machine with 512 MBytes of shared memory, will be presented.

Contents

Chapter 1

Introduction

1.1 Background

During the last two decades, computational engineering has proven to be a very cost effective tool for industry. A significant speed up factor has been achieved in the design-manufacturing cycle, and now *cheap* computational tests can be carried out instead of the traditional *expensive* prototype testing process. Integrating computer simulation effectively into the design process has been successfully achieved, particularly in industries such as automotive and aerospace. In return, as more complex realistic problems are addressed, new issues in the overall process can be identified. The challenge of maintaining computational procedures at a high level of performance, reliability, robustness and efficiency has been growing increasingly. Simulation of problems that involve large scale calculations is still considered to be problematic, particularly when access to advanced computing resources is limited. In short, the discipline of computational engineering in general is still an active area of research and development, and research presented in this thesis is just another small contribution.

Broadly speaking, computer simulation of an engineering problem requires the problem to be first defined in a form of mathematical equations, namely the governing equations. These equations are often expressed in the form of partial differential equations (PDEs), where achieving an analytical solution over the entire domain of interest is almost impossible. However, using a numerical approach these equations can be solved to produce an approximate solution and hence a simulation of the problem. Typically, numerical algorithms represent the continuous form of the governing equations in a simplified discretized form. The discretized form can be based on a set of points (finite difference method [39]) or cells (finite element and finite volume methods [153, 62]). These points or cells (elements) form a grid (mesh) which covers the domain of interest in the problem. In fact, the term *grid* signifies this form of domain representation,

thus, *a grid is a set of points (elements) distributed over a calculation field for a numerical solution of a set of equations* [61]. A diagram that demonstrates different aspects involved in the computer simulation of engineering problems is presented in Figure 1.1.

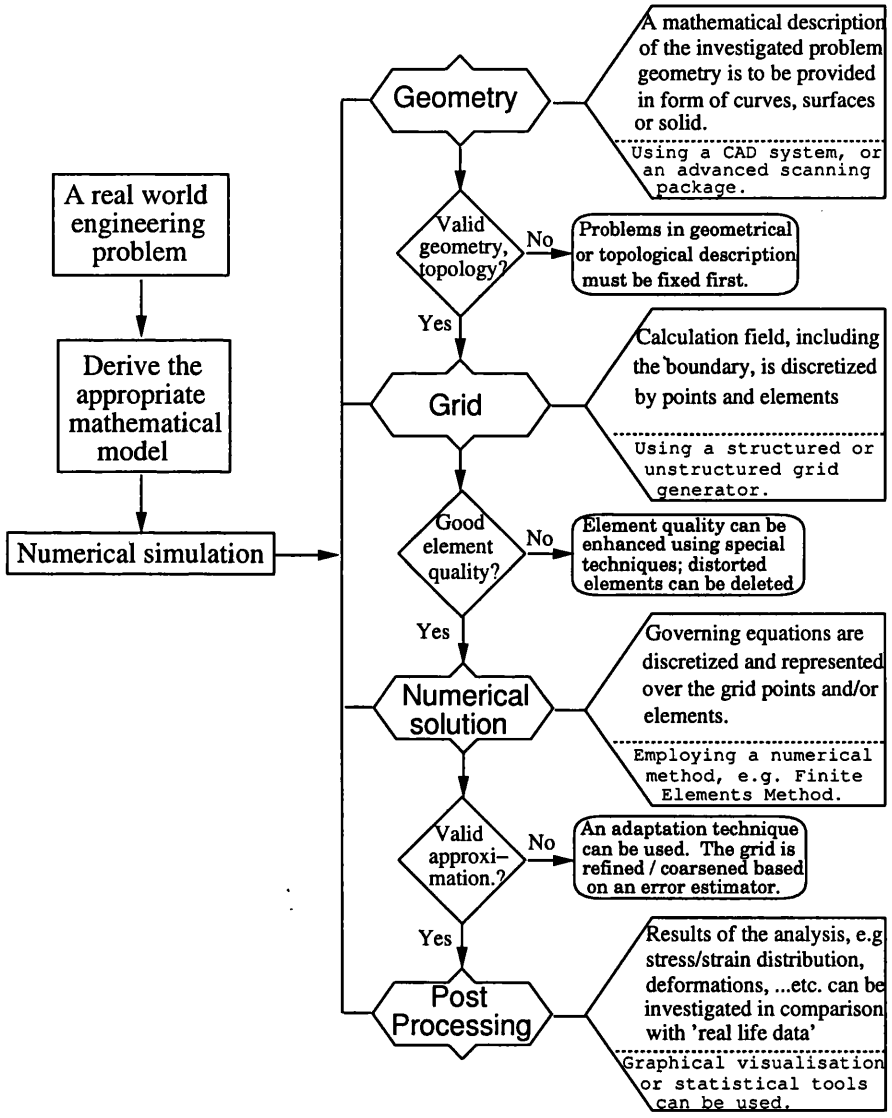


Figure 1.1: An overview of different aspects in computer simulation.

Grids of the order of few tens of elements, which can be constructed manually, used to be more than adequate for the problems attempted at the early days of numerical simulations. However, industry requires solutions to more challenging problems. With the rapid growth in computer power and the availability of computers, these challenging problems can now be addressed. A number of CAD systems which employ well established mathematical models that can

describe complex geometries [8, 19, 33, 64, 104, 34], have become widely available. Grid generation modules are central to such systems. Thus, generating grids for realistic problems with a sufficient number of points and high quality elements (i.e. not skewed) and within a reasonable time has become essential for an effective use of numerical simulation. Obviously, with such requirements manual grid generation is no longer feasible, and the search for a more efficient and reliable way to generate grids has began.

Unfortunately, there has not been any pre-established equations for automatic grid generation, furthermore, the grid generation by its own nature has got a significant element of an art. "The grid generation is not unique; rather it must be designed. Mathematics provides the essential foundation for moving the grid generation process from a user-intensive craft to an automated system" [124]. The roots of automatic grid generation can be traced back to early 1970s with the developments in the finite element method [152]. Interest in numerical grid generation grew enormously in the early 1980s, and since then it has become a worldwide active area of research. An extensive amount of literature has been published and a number of specialised international conferences are regularly organised. Probably the proceedings of the International Conference on Numerical Grid Generation[60, 111, 2, 138, 117, 23, 118], provide the best illustration of this. Nevertheless, although grid generation is still under development, a number of algorithms have matured and have already been implemented into widely available commercial packages.

In fact, having the capability of numerical grid generation has contributed significantly into the computational engineering discipline. A considerable success in extending the numerical methods into a wider range of applications has been achieved, and solving complex engineering problems by numerical simulation has therefore become a reality.

The classification of grids, and subsequently techniques associated with the generation procedure, is widely recognised as including structured or unstructured grids. The basic difference between the two types lies in the form of the data structure that most appropriately describes the grid. If the points are ordered as in a regular array, where a point in row i and column j is the immediate physical neighbour of the point in row $i+1$ and column j, the grid is described as *structured.* If such a form is not applicable then the grid is *unstructured.* Hence, the description of an unstructured grid must include an explicit definition of the point connectivity to define the elements, see Figure 1.2.

Structured grids can be generated algebraically by employing some form of interpolation from the boundary points (transfinite interpolation), or by solving some well-known partial differential equations (Laplacian operator). A structured grid, in two dimensional space, consists of quadrilaterals which are formed by a network of two intersecting sets of lines (not necessarily orthog-

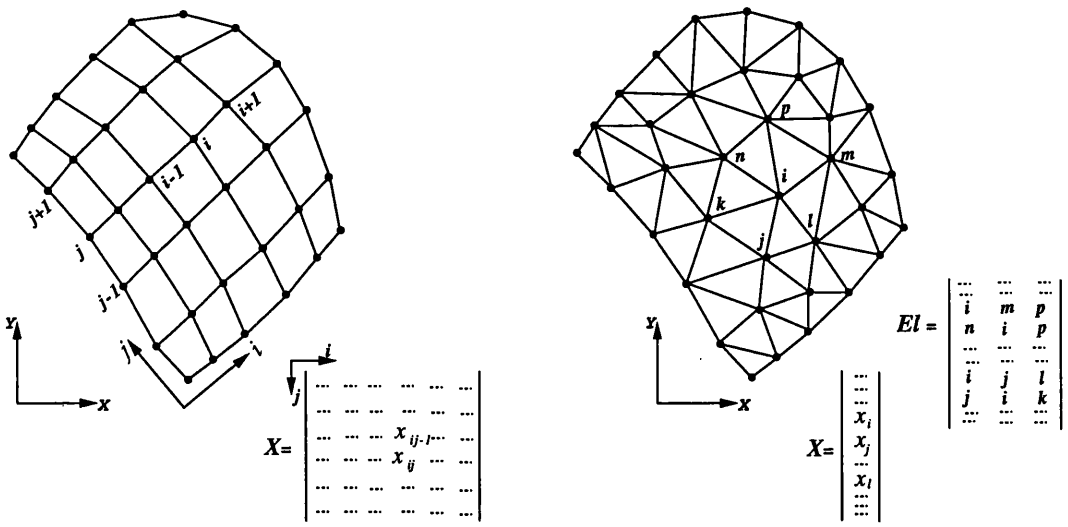


Figure 1.2: Illustration of a structured grid (left) and unstructured grid (right).

onal nor regularly spaced) called a curvilinear coordinate system. The same principle is extendible into three-dimensional space. Of course, for a realistic engineering problem it can be extremely difficult to construct one global curvilinear system which is able to represent all important features accurately. Alternatively, a multi-block technique is often used, in which the domain is subdivided into 'blocks' and a local curvilinear coordinate system is created inside each block [123]. Although the grid is structured within each block, the blocks fit together in an unstructured manner. Such subdivision provides a greater flexibility to construct structured grids for complex geometrical shapes. In fact, the main interest of this research is in unstructured grids. A good source for details on structured grid generation can be found in [122], whilst for more recent developments in this area, the reader is advised to consult relevant chapters in [124].

Whilst creating the position of points is the only concern in structured grid generation unstructured grid algorithms always have an additional task, which is to provide a list that defines explicitly the connections between points (i.e. the element connectivity). Although the construction of such a list may appear as a straightforward task, in practice, implementing it in an efficient computer program can be a real challenge. Unstructured grid generation algorithms depend heavily on geometrical concepts that often involve an extensive search for neighbouring points and elements. Hence, adopting an advanced data structure with a sophisticated search procedure is the corner stone for any successful unstructured grid generator. Since points and connectivities in the unstructured grids embrace no global structure, it is possible to add or delete points and elements *locally* any where in the domain. Such flexibility has made

unstructured grids an ideal approach for discretizing complicated geometrical domains as well as for adaptive solution algorithms (where the grid is modified continuously throughout the simulation with respect to an error estimator).

A number of unstructured generation techniques are already well established and widely used, particularly in the of area CFD (Computational Fluid Dynamic) applications. A broad discussion about unstructured grids, with a detailed description of a Delaunay triangulation algorithm, is presented in the next chapter, see section 2.2. An appendix which is devoted to a discussion on various aspects related to the Delaunay triangulation technique is available at the end of this thesis, see Appendix A. However, principles of different unstructured grid generation techniques are presented in an early book by P. George [48], whilst for a wider and more recent discussion the reader is advised to consult relevant chapters in [124].

Hybrid and *Chimera* are another two different types of grid generation techniques, but they both use concepts from structured and unstructured grid methods. The former simply allows structured and unstructured grids to coexist in one global grid. In a typical hybrid grid, a structured grid is used to discretize some regions, e.g. the regions near a boundary, whilst an unstructured grid connects them all and fills the rest of the domain. In the Chimera grid approach, separate structured grids are generated around various components in the computational domain and then overlaid on a background grid or may be on each other.

Grid generation algorithms, in general, have a common starting point and that is *preparing* the geometrical description of the problem to be gridded. Unfortunately, very often, such geometrical description (normally obtained from a CAD system) is not *ready* to be used by grid generators directly. A number of different issues can be encountered, such as: incompatibility with the mathematical model used in the parametric description of curves and surfaces [1], geometrical discontinuity represented by gaps between adjacent surfaces or curves, poor accuracy in defining intersections between different entities, ...etc. In fact, developing tools to detect and fix such problems is an active area of research, known as 'geometry repair' [106]. Furthermore, even when the 'valid geometry' is achieved some extra preliminary work is still needed such that a *topological description* is constructed on the top of the geometrical entities. In short, a considerable amount of human intervention time is required before a valid geometrical and topological description of the problem domain is obtained [113]. Thus, the 'fully automated' grid generation environment is still a goal more than as a reality. "Grid generation remains one of the most time consuming aspects of numerical simulation of complex configuration" [98].

[1] A number of different models have been proposed during the last few decades, [8, 19, 33, 64, 104], though more recently the NURBS (Non-Uniform Rational B-Splines) model format has been adopted as a standard [28].

1.2 Motivation Behind the Research

Aerospace companies have employed unstructured tetrahedral grids in their CFD simulations for more than a decade. Such simulations have demonstrated that an acceptable level of accuracy for steady compressible inviscid flows over a complex geometries can often be achieved by employing grids of the order of few million elements [91, 93]. Generating a grid of this magnitude can be accomplished nowadays, using a Delaunay based grid generator [136, 137, 139], on a regular workstation within 1-2 hours (i.e. including all aspects of the generation procedure and some other post processing operations such as enhancing the grid quality by collapsing severely distorted elements). Thus, the obvious question one may ask is then; "is there a need for further research and developments in the area of unstructured grid generation?!".

Computational fluid dynamic has, without doubt, been a major driving force behind significant developments in unstructured grid generation. Whilst the inviscid flow can be accurately simulated on a 'small' size grid, simulation of more complex types of flow such as viscous, turbulent, high Reynolds number flows or even transient inviscid flows with moving boundaries has shown a need for much larger size grids. "Due to the inherent complexities of high-lift flows (both geometrical and flow physical), the accurate analysis of such flows requires highly resolved grids. Current estimates place the requirements for accurate Reynolds-averaged Navier-Stokes high-lift analysis of a complete transport aircraft configuration in the range of 10^7 to 10^8 grid points" [89].

In the field of CEM (Computational Electro-Magnetics), in order to achieve a good level of accuracy in simulating the electromagnetic wave propagation around 15 nodes per wavelength can be required [94]. In other words, the size of grid required would be a function of the wavelength, therefore, in the 2D and 3D applications it would be a function of the square and the cube of the wavelength, respectively. Thus, if the wavelength is to be halved in a typical CEM simulation in 3D, then to maintain the same level of resolution the associated grid will require eight times the number of points that were originally needed. Given this acknowledged requirement, the simulation of high frequency electro-magnetic scattering around a complete aircraft will necessitate grids of the order of 10's of millions of elements [94, 145].

Present trends in computational engineering point to a requirement for more multi-disciplinary analysis. The coupling of fluids and structures, or thermal with fluids and structures are good examples. Furthermore, the inter-dependency of many engineering designs on such factors as fluids, structures and possibly electro-magnetics also points to more challenging computation simulations. We are possibly not far away from a requirement to generate a grid outside, inside and through an aircraft for multi-disciplinary simulation. Such requirements will place considerable demands on mesh generators and,

inevitably, very large grids will be required [108].

The above discussion shows that to solve realistic practical engineering problems there is already a requirement for generating grids of the order of 10 to 100 million elements and may be, very soon, for much more beyond that. Obviously, this represents a significant challenge not only in the numerical solution process, but also in the grid generation procedure. In fact, the extensive research in the area of parallel processing of numerical algorithms, with the rapid advances in computer technology, have reached the point were the calculations of such large scale problems can now be carried out in a practical sense [90, 94, 59, 119, 146, 149]. For example, on a machine such as the CRAY T3D, the implementation of an equation solver for CFD is such that approximately 400,000 elements can be accommodated on a single processor. This means that grids of around 200 million elements can be handled if 512 processors can be simultaneously accessed [91, 146, 148]. However, the size of such grid is much larger than can be considered as practical for traditional sequential unstructured grid methods, operating on current computing platforms. A typical sequential Delaunay generator would require 100 MBytes of memory to generate a grid in the order of 1 million tetrahedra, thus to accomplish the generation of a grid of the order of 200 million elements a platform with approximately 20 GBytes of main memory would be needed [2]. Thus, the grid generation procedure has become a bottle neck in the field of computational engineering of large-scale problems.

Of course, the option of adopting a sequential grid generation approach and just seeking access to more and more computing power to generate larger and larger grids has got its own hardware limitations issue. However, several attempts of employing sequential generators to meet the challenge have been reported [94, 90]. A standard h-refinement technique is exploited in order to refine an existing grid. In such a technique, a generated grid can be refined globally by introducing a new point on each edge, then new elements created by appropriate reconnection of the nodes. Two different approaches for implementing h-refinement are addressed here:

First approach. In this approach, after generating the largest grid that can be accommodated on the available resources, a standard h-refinement technique is applied on the entire grid. Then, in order to prepare the *new* grid to be used by a parallel numerical algorithm, a typical grid partitioning technique is implemented. Such techniques subdivide the grid (computational domain) into a number of partitions (sub-domains) which are then distributed as one sub-domain per processor such that the numerical simulation is carried out in parallel. A large number of grid partitioning algorithms have been

[2]These approximate figures are based on the performance of a sequential Delaunay algorithm adopted in this research work, and it has been developed originally by Weatherill [136, 139].

developed over the last two decades [30, 112, 6, 73, 132]. Broadly speaking, most of these algorithms have shown a considerable limitation in handling large size grids from the view point of memory requirements.

Table 1.1 illustrates a comparison of the memory requirements of a sequential Delaunay grid generator [139], the Recursive Spectral Bisection (RSB) grid partitioning algorithm [112] and an Euler flow solver [142]. It is clear that the domain decomposition is the most expensive procedure. The situation is not much better with other grid partitioning algorithms. For example, Metis demands 240 MBytes for a tetrahedral grid with 1 million vertices [74]. Nevertheless, this approach has been implemented in generating and partitioning (using the RSB technique) a grid of the order of 8 million tetrahedra, and another grid of 16 million tetrahedra (using the Recursive Bandwidth Minimisation RBM instead of RSB). Both grids have been constructed on a CRAY YMP/EL machine, which has 256 MWords of main memory [95, 145]. Another attempt is reported in [89], where the issue of memory limitations is clear as well. "... the access to a large central memory provided by the cc-NUMA shared memory architecture of the SGI ORIGIN 2000 is a key enabling feature for the refinement of large unstructured grid ..." [89].

No. of Elements	Memory Requirements (MWords)		
	Grid Generator	RSB	CFD Solver
2.0E+6	38	52	34
4.0E+6	76	104	68
6.0E+6	114	156	102
8.0E+6	152	208	136
10.0E+6	190	260	170

Table 1.1: Memory requirements for a Delaunay grid generator, RSB domain decomposition and Euler flow solver.

Second approach. There are clear limitations in handling refined grids in the first approach. A second approach proposes implementing the h-refinement procedure after the domain has been partitioned. First, an initial grid is generated and partitioned into the desired number of sub-domains (i.e. as required by the solver, so no further partitioning is needed) within the limits imposed by the available resources. Subsequently, the sub-domains are refined separately until the desired grid size is achieved. Ideally, the refinement procedure should be implemented in parallel on the same supercomputer where the solver is to operate. Since the continuity of the grid across the sub-domains interface must be reserved, developing a parallel version of the h-refinement procedure is not a trivial task. However, although this approach may have the potential for generating much larger grids than the first one, in fact, the h-refinement

procedure itself has got its own drawbacks. Also, the issue of memory requirements is still present in the partitioning of the initial grid, which in turn may effect the size of the final grid that can be obtained.

The major drawback of the h-refinement procedure is that, when boundary edges are considered, the newly generated points have to lie exactly on the boundary surface. This is usually an expensive task, and access to the original geometrical description of the model is required. Also, the quality in some of the resulting elements may become very poor, particularly if further refinement was needed after the first cycle. An additional disadvantage is that, since the refinement is applied globally, introducing some extra 'redundant' elements in some parts of the domain is inevitable.

It is clear that in both these approaches access is required to 'advanced' computing resources. In fact, the number of supercomputers available nationwide is still very limited, and accessibility to them for most of universities, research centres, small and medium size companies is restricted and it can be very expensive [27]. Therefore, developing grid generation tools which can produce large size grids without the need for any highly sophisticated computing resources must be a very cost effective strategy.

Obviously, relying on the traditional sequential grid generation methods only to generate large scale grids can not provide a practical solution. "... Unstructured grid generation for complex high-lift geometries can be accomplished in a matter of days ..." [89]. So, alternative approaches had to be investigated, aiming to make the time scale more practical and, if possible, to preclude the bottle neck of memory requirements. A considerable number of algorithms that implement parallel processing technology in various ways have been developed recently, some of them are reviewed in detail in the next chapter (see section 2.4.2).

To conclude, the answer to the question "is there a need for further research and developments in the area of unstructured grid generation?!" is: Yes, there is still a demand for further developments in the unstructured grid generation field. And the priority, from our point of view, is to find a solution to the generation of large size grids. However, parallel grid generation is still very young and certainly more research is justified. "... because the development of efficient scalable parallel techniques takes much more time than their sequential counterparts, it may take a while before parallel mesh generation comes to a state of maturity ..." [22].

1.3 Aim and Overview of the Research

The ultimate aim of this project is *to develop a 'tool' by which large-scale unstructured grids for realistic engineering problems can be generated efficiently*

on any parallel computer platform. The adopted strategy is based upon a geometrical partitioning concept, where the computational domain is subdivided into a number of sub-domains which are then gridded independently in parallel. The final grid of the entire domain can be constructed by merging grids in the sub-domains or, based on the user choice, it may remain as a distributed grid. In the later case, all the nodes that exist on more than one sub-domain boundary must be clearly identified, so the inter-domain communication lists needed in parallel simulations can be established. The approach is applicable in both two and three dimensions and, in general, to different types of grid generators. This study focuses on three-dimensional applications only, and it implements a Delaunay triangulation based generator [136, 139] to generate the sub-domain grids.

The entire procedure can be accomplished by employing four different algorithms as in the following order:

1. **A domain decomposition algorithm**, in which the volume enclosed inside the computational domain boundary is partitioned into an arbitrary number (defined by the user) of sub-domains.

2. **An inter-domain boundary gridding algorithm**, in which the 'new' internal boundaries created between adjacent sub-domains are extracted, and a smooth triangular grid is generated on each one.

3. **A sub-domain gridding algorithm**, in which a closed and properly oriented triangular grid is constructed on each sub-domain boundary, such that a grid can be generated using a sequential grid generator.

4. **A post processing algorithm**, so that the user has the opportunity to choose a procedure that most suits the future use of the generated grid, e.g. constructing one global grid, ensuring a well-balanced distribution of the elements, ...etc.

Two different approaches have been investigated, where the variations between them are limited to the domain decomposition and inter-domain boundary gridding algorithms only. Both approaches assume that a triangular grid has already been generated on the domain boundary. However, in order to carry out the domain decomposition task, the first approach requires an initial tetrahedral grid to be constructed (using Delaunay triangulation of the boundary points), whilst the second approach operates directly on the boundary triangular grid. Hence, this thesis will refer to the first approach as 'indirect decomposition method' and to the second as 'direct decomposition method'.

Indirect decomposition method. In fact, this method is an extension of the 2D original algorithm developed by Weatherill and Verhoeven [128, 129]

into three dimensions. In this algorithm, following the triangulation of the boundary points, the points are connected together by a set of tetrahedra using Delaunay and then a greedy-type algorithm is employed based upon an equal volume (area) criterion [30]. The total volume of the entire domain is subdivided 'equally' into a number of sub-domains. This is a simple, fast and memory efficient procedure for partitioning unstructured grids but, unfortunately, it may produce highly unbalanced workload in the sub-domains. Enhancing this algorithm so that it accounts for the effects of grid control parameters, which define the distribution of grid point density, has been investigated.

Having the sub-domains formed as a group of clustered tetrahedra in the initial grid, the inter-domain boundaries are then formed by extracting all triangular planar faces that are shared between adjacent sub-domains. Thus, each internal boundary is represented by a surface, which is already defined in a discretized form. A typical surface will have a set of triangles which are extremely distorted and irregular. A topology correction and smoothing algorithm has to be implemented before any of these surfaces can be gridded. Eventually, a point insertion with edge swapping technique is employed to generate a smooth triangular grid on each surface.

Despite achieving a considerable success in generating standard quality and smooth triangular grids on the internal boundaries, the indirect decomposition method overall has failed to demonstrate a steady robust performance in gridding complex geometries. In addition, the demand for generating an initial tetrahedral grid has proved to be difficult. Hence, an alternative approach to partitioning the computational domain had to be investigated, which subsequently has led to the development of the direct decomposition method.

Direct decomposition method. In this method, the volume enclosed inside the triangular grid on the boundary is subdivided *directly* by a number of planar cuts. In general, these cuts can be imposed in various ways, such as a set of parallel planes distributed along one axis, a Cartesian network of planes which can be imposed in one step or in a recursive manner or even by employing an octree decomposition procedure. However, the general shape of the inter-domain boundary created by any of the proposed procedures must be simple planar surface. Thus, the problematic irregular form of the internal boundary surfaces in the previous method has been excluded and, furthermore, the decomposition technique allows for a better control over the workload distribution.

The technique adopted in this study uses the unidirectional planar cut approach with various options for assigning a criterion for the planes location. The available criteria are: equal spacing, equal number of boundary triangles and interactively, where the exact location of each cut is defined explicitly by the user. However, after the planes have been defined, every set of trian-

gles bounded by two plans is assigned to be in a sub-domain. Consequently, internal boundaries are defined by extracting edges that are shared between adjacent sub-domains at the interface regions. In order to create a surface grid on such a boundary, a closed contour of 2D edges is constructed by mapping the extracted 3D edges onto the surface of the cut plane. A coarse triangular grid is generated on this surface in the 2D space which is then mapped back into the three dimensional space. The same point insertion and edge swapping technique used in the indirect decomposition method can be employed to generate the final fine smooth grid. Alternatively, the fine grid can be generated in the 2D space before it is mapped back, however, the former way has shown a smoother final grid and a better consistency with the original boundary grid point spacing. In general, high quality grids on the internal boundary are always obtained in the direct decomposition method, which in turn has made the method very reliable.

Clearly, after applying the first two algorithms in the global procedure, whatever method from the above is used, the *big* task of generating one global grid for the entire domain becomes a collection of *smaller* similar tasks. An independent closed triangular grid is constructed for each sub-domain, by merging the relevant parts of internal and original boundary grids. Thus, a typical sequential grid generation algorithm can now be employed to generate a tetrahedral grid inside each sub-domain independently. A Delaunay triangulation based generator developed by Weatherill [136, 139] has been implemented in this research. Such volume grid generators are sensitive to the orientation of the boundary faces. Therefore, a technique to ensure all boundary faces are correctly oriented before the generation procedure starts has been integrated.

In fact, at least one post-processing operation has to be applied before the grid obtained is ready to be used. Various options have been made available, so the user can choose according to the application and the expected employment of the grid. However, it is very likely for grids generated in such an approach to be employed in distributed parallel simulations, therefore, having a balanced distribution of the total workload on the sub-domains is vital. A post-processing procedure that shuffles elements between adjacent sub-domains, in order to achieve a highly balanced workload, has been developed. Other procedures such as constructing one global grid by merging sub-domain grids, establishing inter-domain communication tables, partitioning a sub-domain locally and inter-domain relaxation are available as well (see chapters 5 and 6).

Work presented in this thesis also concerns the development of a framework in which the four algorithms introduced above are integrated in combination with a specially designed parallel processing technique. The Message Passing Library (MPL) programming model has been adopted throughout the parallelisation work in this research. The framework implements a Single Program Multiple Data (SPMD) structure with a Manager/Workers mechanism. In

such mechanism, one of the processors is assigned to be a manager, which 'administers' the processing only, while the rest are workers where the 'real work' is carried out in parallel. The designed technique, termed as Dynamic Parallel Processing (DPP), uses this mechanism in a way where processing an arbitrary number of tasks becomes always possible whatever the number of processors is available.

The administrative role of the manager in the DPP technique involves: identifying the jobs that are due to be processed in parallel, sorting them in descending order according to an estimated workload, preparing all data required to process each job independently and finally synchronising the job processing on the workers efficiently. On the other hand, when the workers receive these jobs, the same list of instructions must be applied by all workers while each acts on its own set of data (e.g. applying the volume Delaunay grid generator on the sub-domains). Every worker must notify the manager on the completion of an assigned task and go back into the 'stand by' mode. In the DPP technique, there is no communication among the workers themselves, and the only type of communication that takes place is when and only when the manager wants to assign a job to a worker or receives a result. The MPICH, which is an implementation of the Message Passing Interface (MPI) library specifications [53], has been employed to carry out all the communication (message passing) operations. Benefiting from the portability of a such library, in addition to the careful design of the parallelisation work, the framework after all can operate on a wide range of computing platforms: a single workstation, cluster of networked workstations, shared memory multiprocessors machines and massively parallel supercomputers.

The DPP technique has been implemented to process various tasks in the algorithms above, including: generating the internal boundary surface grids, generating the sub-domain tetrahedral grids and another two different tasks involved in the element re-distribution procedure (i.e. an option in the post-processing algorithm). Only a few simple tasks of the procedure overall, particularly in the direct decomposition method, remain to be processed sequentially. However, the developed framework has proved to be a very effective tool for generating large size grids, also the enhancement by the DPP technique has provided a great flexibility and efficiency in exploiting the available computing resources. Grids to the order of more than 100 million elements have been generated using one processor on an SGI Challenge machine which has 512 MBytes of main memory. A number of realistic engineering problems with complex configuration are presented in this thesis.

1.4 Layout of the Thesis

Chapter 2: Before discussing any technical details of the developed algorithms, and in order to put this research in perspective, some essential topics are introduced. The general algorithm of a Delaunay based tetrahedral grid generator is discussed, and an overview of some relevant parallel processing technology is presented. The chapter then presents a brief survey of previous research in the parallel grid generation field, and highlights the advantages of the geometrical partitioning approach in general.

Chapter 3: Both direct and indirect decomposition methods are discussed in detail. The entire procedure for each method is introduced in a simple 2D format first, and then followed by further discussions about the 3D implementation. Illustrated examples are presented, and issues associated with each method are addressed.

Chapter 4: This chapter is devoted to the discussion of the parallelisation work in this research. The Dynamic Parallel Processing technique is presented in great detail. The main steps in the general procedure are examined individually in order to estimate the benefit of, or the possibility for, implementing the DPP technique on each step. An example to illustrate the DPP role in the parallel frame work is presented as well.

Chapter 5: This chapter is divided into three main sections, each one is devoted to a discussion on a post-processing algorithm. Every section starts by addressing the main reasons behind developing the relevant algorithm, then followed by further technical details and an illustrated example. The three sections are: inter-domain communication, grid quality and load balancing.

Chapter 6: This chapter simply demonstrates the developments and how they can be employed in the 'real world'. Various options are demonstrated, including the construction of one global grid and applying a local partitioning procedure on individual sub-domains. Grids of realistic engineering problems to the order of 100 million elements are presented, also results obtained from typical parallel CFD simulations using some of generated grids are shown. The chapter also demonstrates the high efficiency and scalability of the developed programs by examining thoroughly a number of well known parallel performance measurements.

Chapter 7: A conclusion from the overall research is presented, and areas for further investigation are indicated.

Two appendices are introduced in this thesis, the first one provides an insight into the Delaunay triangulation technique, including a review of some point creation procedures. The second appendix introduces some basic features in the Message Passing Interface library, with emphasis on the data communication functions which are related to the parallelisation work. Also, a summary

of parallel performance tools, which is used during the performance analysis study, is presented.

Chapter 2

An Overview of Unstructured Grid Generation and Parallel Processing

2.1 Introduction

In order to put the work carried out during this research into perspective, it is appropriate to review some of the essential and relevant concepts. Therefore, this chapter attempts to review, very briefly, various topics in the unstructured grid generation and the parallel processing fields. First, a description of the most well known techniques in the unstructured grid generation area is introduced. Then, an algorithm based on the Delaunay triangulation technique, which has been adopted in this study, is outlined. Fundamental concepts and other technical issues related to the Delaunay triangulation approach are discussed in Appendix A. Discussion then focuses upon parallel processing issues as a very compact review of parallel machines and parallel programming models is presented. A wider discussion of the Message Passing Library model, which is the adopted strategy in this study, is presented in Appendix B.

Having introduced the basic concepts of unstructured grid generation and parallel processing, the chapter then moves on to discuss how researchers have used them in order to develop new techniques that can generate unstructured grids in parallel. Different types of approaches are addressed, advantages and disadvantages of each are discussed. A number of algorithms that have been developed over the last decade are reviewed briefly.

2.2 Unstructured Grid Generation

Recalling the description of unstructured grids as in Section 1.1, particularly their flexibility for complex geometries, grid adaptation and relative ease and speed of use, it is clear why there is a great interest in using unstructured grids in numerical simulation. In fact, since the introduction of unstructured grids into the CFD world early 1980s, unstructured grid generation has proved to be a very effective tool and a key component in simulating realistic engineering problems. However, ".... disadvantages following from adopting the unstructured grid approach are that the number of alternative solution algorithms is currently rather limited and that their computational implementation places large demands on both computer memory and CPU. Further, these algorithms are rather sensitive to the quality of the grid being employed and so great care has to be taken in the generation process. ..." [58].

The process for constructing an unstructured grid inside a closed domain (volume grid) requires the boundary of the domain to be represented in a discretized form. In other words, volume grid generation requires the generation of grids on boundary curves and surfaces. The boundary surface grid has a direct effect on the volume grid inside the domain, in particular on the quality of volume elements near the boundary. Hence, surface grid generation is considered as a very important step in the unstructured grid generation process [100].

In general, unstructured grid generation algorithms use geometrical search operations extensively. The selection of optimal data structures and very sophisticated search algorithms, [9, 38, 77], is essential otherwise the efficiency of the grid generator will drop dramatically. Elements in unstructured grids, in principles, can have any geometrical shape, however, triangles on surfaces and tetrahedra in volume are still the most widely used shapes. Quadrilaterals and hexahedra are also used and favoured in some applications.

The last two decades have seen a wide and intense research activity in the field of automatic grid generation [60, 111, 2, 138, 117, 23, 118, 124], and a large number of unstructured grid generation algorithms have been developed. Obviously, it is beyond the main theme of this thesis to get involved in reviewing such a long list of algorithms. However, the majority of these algorithms depend on one of the two approaches, namely the Advancing Front and the Delaunay Triangulation [86]. The Delaunay method is based on an elegant and simple concept which can be traced back to the nineteenth century, with a paper by Dirichlet appearing in 1850 [26]. Given a set of points, a diagram known as the Voronoi diagram can be constructed. This diagram consists of a set of polygons in which each is associated with a point, and defines the region that is closer to this point than to any other point. Every segment of a polygon in the Voronoi diagram is equidistant from the two adjacent points that it

separates. If points with a common boundary are connected, then a triangulation of the points is formed. This type of triangulation is known as Delaunay triangulation. The Delaunay-Voronoi construction can be constructed on a set of points in n dimensional space, however, the resulting triangulation would consist of a set of triangles in two dimensions and a set of tetrahedra in three dimensions. Nevertheless, since generating unstructured tetrahedral grids using the Delaunay triangulation method is essential to this research, it is appropriate to have a wider discussion. An overview of the general procedure in a typical Delaunay grid generator is presented in the next section, and details of relevant technical aspects are introduced in Appendix A [5, 48, 136, 139]

The advancing front method [81, 84, 99, 100, 103], consists of marching into the domain creating elements. The region separating the gridded portion of space from the as yet ungridded one is called the front. Given an initial front, which in two dimensions is the set of edges that represent the boundary curves and in three dimensions is the set of triangular faces that represent the boundary surfaces, a node is created and connected to the front so a new element is formed. The process continues by advancing and updating the front until the entire domain is filled with elements. Clearly, while a new element is being formed, it is essential to ensure that no intersection with any existing elements occur, as well as try to satisfy element quality criteria. Hence, performing a considerable amount of *local checks* before a new element can be formed is inevitable. Of course, this highlights the important role of the advanced data structures and search techniques [9].

Another less known approach for unstructured grid generation is based on spatial tree subdivision, which makes full use of tree data structures to decompose the domain into elements. It is often referred to as the Octree technique, or Quad-tree in the two dimensions case. However, the basic concept consists of placing the entire geometry, as represented by the boundary grid, in a cube. Then to subdivide the cube into its eight octants which are then recursively subdivided a number of times until the length scale in the terminal octants become consistent with the length scale on the boundary grid. When no further subdivision is needed, the final step requires the subdivided grid to be connected to the boundary grid [114, 151].

A number of methods based on a 'combination' of two of the approaches mentioned above have emerged. Schroeder and Shephard have developed a method that combines the Octree method with Delaunay. "The basic concept is to use an Octree building procedure to generate octant geometries that can then be tetrahedronized using Watson's Delaunay algorithm" [110]. Another algorithm, which combines the advantages of efficiency and nice mathematical properties of the Delaunay approach with the Advancing Front high-quality point-placement strategy, has been developed by Frey *et al* [42]. In this algorithm, and after a boundary grid is generated, the internal points are created

using advancing-front point placement and then inserted using the Delaunay triangulation method.

2.2.1 Delaunay Grid Generation Algorithm

In general, there are three different problems that a Delaunay based grid generator must be able to solve. Firstly, the problem of how to connect a set of points already defined in k dimensional space (where $k \geq 2$), secondly, how to generate the position of new points such that a desired density of grid points is satisfied, and thirdly, how to ensure that the resulting triangulation is boundary conforming. Whilst reliable and robust techniques are well established for the first two problems, the last problem is still a subject of further research [3, 4, 49, 143]. However, here we focus on illustrating the general procedure of a Delaunay based generator developed by Weatherill [136, 137, 139], which has been adopted in this research. This grid generator implements Bowyer's algorithm for constructing the Delaunay triangulation [12], in conjunction with an advanced technique of point generation and point density control; details of these technical aspects are available in Appendix A. The general procedure is presented in association with an illustrated example of a simple 2D triangular grid, see Figure 2.1 and reference [147].

Consider a circle with its boundary already discretized into a set of edges and points. See (I) in Figure 2.1; notice the 'point source' located at the centre, which is used by the mechanism that controls the grid point spacing [1].

- The first step is to define a convex hull that encloses all the boundary points. A simple construction can be imposed, which consists of four points (two triangles) in the 2D applications case and eight points (five tetrahedra) in the 3D case. See (II) in Figure 2.1.

- Given the initial Delaunay construction of the four points in the convex hull, points on the circle boundary can then be inserted one at a time and connected into an already existing Delaunay triangulation structure. Figure 2.1 (III) shows the resulting grid after all the boundary points have been inserted.

- To create the grid inside the circle, it is then necessary to systematically refine the triangles inside the circle. There are several methods for performing this task, however, the proposed algorithm implements a simple interpolation to insert points at the centroid of elements. The grid point density is controlled by a background grid and any 'sources' have been specified, see Appendix A. Points are created by looping over elements

[1] Details about this mechanism and the 'sources' can be found in Appendix A.

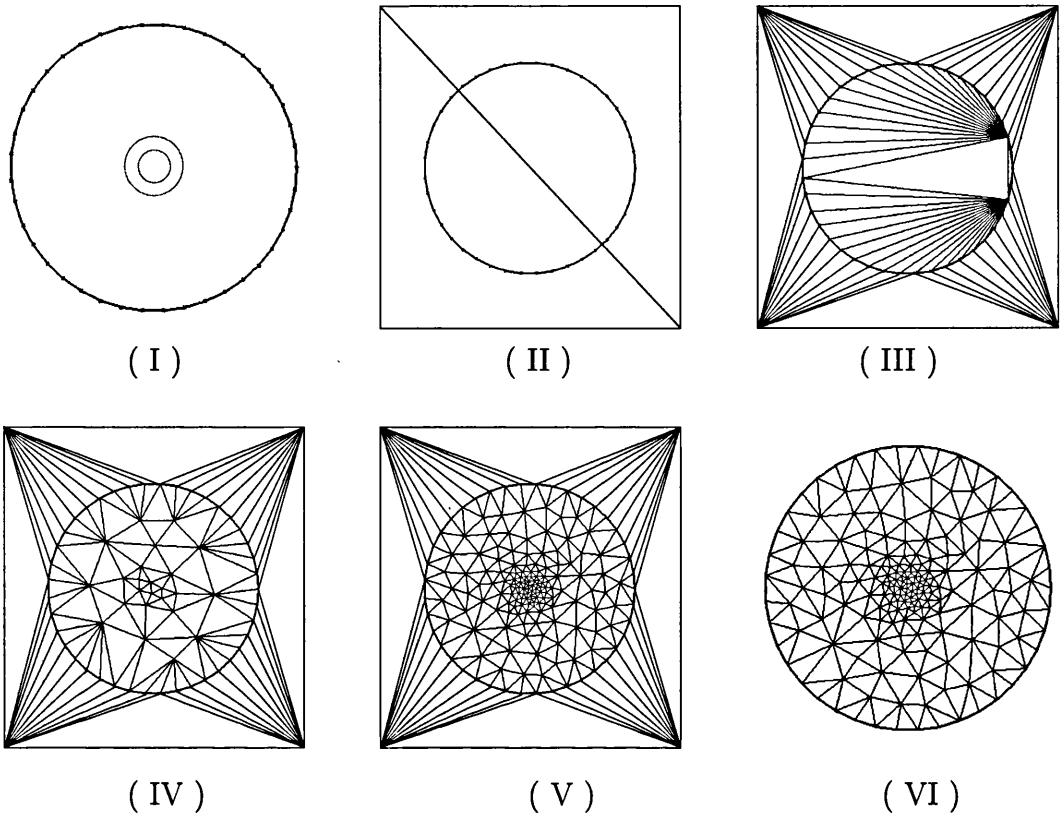


Figure 2.1: Illustration of the general procedure of a Delaunay based grid generator. The boundary as discretized into a set of edges in (I), with a 'point source' that controls points density. Four points with their prescribed triangulation are imposed, i.e. convex hull, such that all boundary points are enclosed (II). The triangulation of all boundary points (III), performed using Delaunay. In (IV) and (V) two different grids during the point insertion phase. The final grid after all 'redundant' elements been deleted (VI); notice the effect of the point spacing source at the centre.

within the domain and inserting a point when element refinement is required, and subsequently connected into an existing triangulation using the Delaunay based algorithm. (IV) and (V) in Figure 2.1 show the grid at two different stages during this process.

- Once the grid point density has been achieved, a post-processing step deletes all triangles that are not inside the domain of interest. In fact, although such a step looks rather easy in this example, it may involve the use of a complex and expensive procedure for complex shape boundaries. The reason behind this step is that to ensure the initial boundary edges are preserved in the final grid. This is the so-called *boundary integrity* requirement which can be a challenging problem, particularly in the tetrahedral grids [143]. However, (VI) in Figure 2.1 shows the final

grid, where the effect of the point spacing source at the centre is clear.

2.3 Parallel Processing

"Parallelism has sometimes been viewed as a rare and exotic sub-area of computing, interesting but of little relevance to the average programmer. A study of trends in applications, computer architecture, and networking shows that this view is no longer tenable. Parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise", [40]. It is impossible to cover such an active area of developments in an introductory review. However, a brief description of some relevant aspects and clarification of some special terms is still feasible and appropriate.

Parallel Machine Models

There are two different major architectures of parallel machines, namely the Single Instruction Multiple Data (SIMD) machines and the Multiple Instruction Multiple Data (MIMD) machines [2]. In SIMD machines, processors execute the same instruction stream simultaneously while each processor is acting on a different piece of data. This type is also known as *vector computers*, the Cray YMP machine series is a typical example of this type. In fact, the SIMD type is irrelevant to the parallelisation work developed in this research, thus no further reference to it is to be made; an interested reader can consult [126].

The MIMD machines are parallel computers which contain a number of interconnected processors, each of which is programmable and can execute its own instructions. The instructions for each processor can be the same or different. The processors can operate on a *distributed memory* or *shared memory.* In the former, which is referred to sometimes as multicomputers, every processor has its own memory which no other processor has a direct access to. A typical example of this type is massively parallel machines like the Cray T3D and IBM-SP2. Clusters of networked workstations is another form of this architecture as well. In the shared memory case all processors have read and write access to any address of the available memory, the type is known also as multiprocessors system. A typical example of this type is the SGI Challenge and the Origin 3000.

Diversity in the literature and commercial articles that evaluate the technical developments and features of high-performance computing hardware is very wide. However, for a reader interested in finding out more about parallel machine models, we recommend the two following references: A.K. Noor in [96] presents a wide review of miscellaneous types of high-performance computing

[2]The standard serial computers in this respect can be referred to as the SISD type. Also, though there are few machines in the MISD architecture category, none that have been commercially successful or had any impact on computational science [96].

technology, with a brief assessment of the current and projected advances in distributed computing and networking technology. A. Trew and G. Wilson in 1991 reported extensively on parallel computers which were available commercially at the time [126]. In fact, the survey can be considered as old and out of date by now, but the book still provides a valuable comprehensive technical insight of parallel machines. Also I. Foster in [40] devotes a section for *Parallel Machine Model.*

Parallel Programs Structure
The two terms MPMD (Multiple Program Multiple Data) and SPMD (Single Program Multiple Data), appear frequently in the parallel processing literature. Thus, we intend to clarify their meaning here since any inconsistent interpretation may lead to some confusion in this thesis context. First of all, not to over confuse them with the other two popular terms MIMD and SIMD that introduced above. Whilst the MIMD and SIMD represent two categories of computer architectures, the MPMD and SPMD describe two different types of *parallel program structure.* "The MPMD programs are usually specified by the *parallel block* construct or the *multiple-code* approach, while the SPMD programs are usually specified by the *parallel loop* construct or the *single-code* approach", [68]. Although both structures adopt the concept of different data domains per processor, in the SPMD programs the *same* code is executed by all involved processors whilst in the MPMD programs processors may execute *different* codes. The second point which, more importantly, to be clarified in here is that both of the SPMD and MPMD structures are associated with the MIMD type machines. Thus, both SPMD and MPMD are MIMD, and in this thesis they will be used in conjunction with the program structure only, i.e. regardless if the used platform is a MIMD machine with shared memory or distributed memory. It is interesting to mention that employing the two structures (i.e. SPMD and MPMD) within one parallel program is totally accepted.

2.3.1 Parallel Programming Models

Traditional parallel software designs have attempted to match the architecture of the underlying hardware systems, so as a result application development was problematic as hardware evolved [27]. However, broadly speaking two types of parallel programming models are widely recognised nowadays, they are the 'explicit model' and the 'implicit model'. In the former model, the parallelisation is *explicitly* specified by the programmer using special parallel programming languages, or calling special library functions. Whilst in the implicit model, the programmer lets the compiler and the run-time support system automatically exploit all the possibilities for parallelisation. This model is irrelevant to the work done in this thesis and very little attention will be

given, however, an interested reader can consult [68].

Although several explicit models have been developed, the *data-parallel* model and the *message passing library*, which are the most popular models, are to be considered. The term *data parallelism* refers to the concurrency that is obtained when the same operation is applied to some or all elements of a data ensemble. A data-parallel program is a sequence of such operations. A 'special' programming language is needed to write such program, though the control flow is just like any typical sequential program. Fortran 90 is a data-parallel programming language in its own right, but the data distribution directives are available within an extension known as High Performance Fortran (HPF). "... F90 provides constructs for specifying concurrent execution but not domain decomposition. HPF augments F90 with additional parallel constructs and data placement directives, ...", [40]. HPF is the most common data-parallel programming language, and it will represent the data-parallel model from now on in this thesis. The mechanism employed in HPF model for distributing data is based on a systematic style of array decomposition, whereby series of columns is directed to each processor. Compiling an HPF program on a distributed memory machine produces an SPMD program, where data is partitioned into sub-groups and each one is allocated to a processor. The compiler will launch the communication operations when they are needed.

In the Message Passing Library (MPL) model, processors execute program(s) written in standard sequential programming languages and utilise special library functions to interact by sending and receiving messages. Message passing libraries provide a wide range of functions to manipulate and transfer data. Operations like, deriving new data structure, packing and unpacking data, sending and receiving data between two known processors, broadcasting from one processor, ... etc. are the basics which almost every message passing library provides. Also, Message passing libraries usually provide various versions of its functions that cover all known stander sequential programming languages, which is known as a language binding. In Appendix B, a typical message passing library (i.e. an implementation of the MPI specification known as MPICH) is reviewed, examples of some basic functions with illustration of a standard 'MPI program' are also discussed.

Unlike the HPF model, the MPL model does not provide an automated way to decompose or distribute data. Instead the programmer has to take care of and control all these activities. However, although that the mechanism used for data decomposition and distribution in the HPF model may appear to be very attractive, in practice, it has narrowed its applications area considerably. It is a well known fact that the HPF model has not managed to become as popular as the MPL model. "A problematic feature of HPF is the limited range of parallel algorithms that can be expressed in HPF and compiled efficiently for large parallel computers.", [40].

Most unstructured grid generation techniques use a geometrically localised data, e.g. inserting a point in a Delaunay based generator (see Appendix A). Therefore, the way that data is distributed within the HPF model is not very helpful, and a large amount of irregular communication among processors would be inevitable. Although the programmer does not have to introduce the communication tasks explicitly (since it is done in an automated and optimised way by the compiler) it is very unlikely to achieve an acceptable performance after all. K. Hwang and Z. Xu in [68] report "It is doubtful whether HPF can efficiently support parallel algorithms with general data structures and irregular communication patterns". I. Foster, in his famous book *Designing and Building Parallel Programs*, [40] says: "The performance of an HPF program depends not only on the skills of the programmer but also on the capabilities of the compiler". However, just to conclude, HPF might be an ideal parallel programming model in some cases, but certainly it is not the appropriate model to be used with the unstructured grid generation. To the best of our knowledge that the work of Chen *et al*, presented in [16], is the only reported attempt for using HPF to develop a parallel Delaunay grid generator.

2.4 Parallel Processing and Unstructured Grid Generation

In this section, we will address some of the research activities that have been reported over the last decade in the field of parallel processing of unstructured grid generation. The field is still very young, with the first relevant paper published by Löhner in 1992 [82] [3]. In fact, it is very likely that some of the algorithms discussed in here are still a subject of further developments, including the one developed in this research [149], and new versions might be already emerging. Nevertheless, it is appropriate to emphasise that algorithms related to parallel grid *refinement* are not considered, since they do not really belong to the area of parallel grid *generation* [4].

In general, two options are available for developing a parallel grid generator, one is to parallelise an existing algorithm and the other is to parallelise the problem. The later option is widely known as the geometrical partitioning approach, in which the domain is subdivided into a number of sub-domains that are then gridded separately.

[3] For information about the parallel processing in structured grids field see [61].

[4] In fact, a confusion between the two terms has been observed in some of the literature

2.4.1 Using an Existing Algorithm Approach

Reviewing the brief description of unstructured grid generation methods in section 2.2 and the Delaunay based algorithm in Section 2.2.1; it is clear that unstructured grid generation procedures are essentially scalar. Most of the methods presented to date construct grids by introducing a new element, point, at a time. "Unstructured grid generators belong to a wide class of problems that are characterised by: (i) being scalar; (ii) having a large variation in the number of operations required during each step of the calculation; (iii) achieving parallelism by some sort of distance" [115]. Thus, unsurprisingly a very limited number of attempts have been made to parallelise an existing algorithm. However, a list of advantages and disadvantages of this approach is presented in Table 2.1, followed by a brief discussion of a typical algorithm uses this approach. Other algorithms that are, in a way or another, related to this approach can be found in [105, 80, 25, 76]. Also, an early attempt to parallelise the 'searching' task by developing a vectorised algorithm for determining the nearest neighbours is presented in [10].

Disadvantages	Size of generated grids is limited by the available memory. There is a need to use a domain decomposition algorithm on. the final grid before it can be used by a parallel simulator. The performance depends on number of available processors. Cost of inter-processor communication is very high.
Advantages	No new algorithms are to be developed. A possibility for improvements on the speedup factor. Grid quality should not be effected at all.

Table 2.1: Advantages and disadvantages of parallelising the grid generation task by developing a parallel version of an existing algorithm.

A parallel implementation of the Bowyer-Watson (BW) algorithm, which is the base for most of the available Delaunay grid generators, see Appendix A, has been presented by Chrisochoids and Sukup in [18]. "The element creation step of the BW algorithm takes place into two phases: (i) cavity computation, for the newly inserted point, and (ii) reconnection of the new point with the vertices of the cavity." The paper is concerned about the efficient implementation of the element creation step on distributed memory computers. However, obviously two new points can not be inserted concurrently if their corresponding cavities overlap. In such a case synchronisation among involved processors is essential in order to produce a unique and valid Delaunay triangulation. The cost of this synchronisation can be very high, and it is indeed the main source of the inefficiency in this type of approach. The paper demonstrates the algorithm using a very simple and small size planar triangular grid. Very little

information about the parallel implementation and performance is given, however, the paper reports that "... indeed preliminary performance data indicate linear speedups; ... The slow down occurs mainly due to 28.7 % in polling for remote service requests and 39 % in testing for global pointers". In fact, the same algorithm has been enhanced later on by a form of *manual* domain partitioning procedure and presented in another paper [17], which makes the new version falls under the second approach category. Nevertheless, the modified algorithm is supposed to minimise the interprocess communication required, though this has not been clearly demonstrated.

2.4.2 Using a Geometrical Partitioning Approach

As mentioned earlier, the main concept in this approach is to split one *big* grid generation task into a set of *smaller* grid generation tasks by partitioning the domain of interest into a number of sub-domains. Although details between available algorithms may vary dramatically they all share the same starting point, i.e. partitioning an *empty* space defined by a closed boundary which can be described as a geometric model or as a discretized surface. In fact, such a task can be rather difficult "... Partitioning in the context of parallel grid generation is hard ... It means one is trying to partition 3D domain having only the knowledge of its boundary ... Proper evaluation of the work load is also a challenge ... It is problematic to accurately predict the number of elements to be generated in a given sub-domain, or how much computation per element will be required ... " [22].

Grid partitioning algorithms available to date, [30, 112, 6, 73, 132], are designed to partition an already existing grid and they can not be applied directly on an *empty* space. Thus, researchers were left to choose between either developing their own *new* partitioning technique or to employ an existing partitioner acting on a *coarse* grid or on a background grid. Algorithms that operate on the geometrical definition of a domain have been reported as well. Lammer and Burghaedt presented in [78] an algorithm which applies a partitioning procedure based on calculating the centre of gravity and the principle axes of the domain. Saxena and Perucchio in [109], employ a typical tree decomposition procedure to divide the domain into a number of octants which then are distributed among available processors.

Shepherd and de Cougny in [22] categorise the parallel grid generators available to date using the order of gridding the internal boundary within the overall procedure as main criteria. Thus, the following three categories can be identified:

1. Algorithms that discretize the internal boundary while the sub-domains grid are been processed.

2. Algorithms that carry out the gridding of the internal boundary as a post processing procedure.

3. Algorithms that discretize the internal boundary first and then build a set of sub-domains ready to be gridded independently.

A typical algorithm in each category will be discussed briefly, but first of all an overview of advantages and disadvantages of the geometrical partitioning approach in general is presented in Table 2.2, see Table 2.1 for comparison with the 'using an existing algorithm' approach.

Disadvantages	Extra tasks are introduced to the general algorithm beside the grid generation task. Gridding the internal boundary can be a challenging task. Load redistribution among the sub-domains may be required during the gridding process or as a post-processing step. Global grid quality may become effected by the quality of the internal boundary grid.
Advantages	The generated grid is already been partitioned, ready to be used by parallel simulators. Very large size grids can be generated. A good speedup factor can still be achieved. Inter-process communications can be negligible. With a *good* quality grid on the internal boundary, the global grid quality is likewise any grid generated sequentially.

Table 2.2: Advantages and disadvantages of parallelising the grid generation task using the geometrical partitioning approach.

Gridding the sub-domains and internal boundaries simultaneously

Okusanya and Peraire present in [97] an algorithm for generating triangular grids in parallel using the Delaunay triangulation technique. The process is based on a cycle of point insertion and load balancing operations. Various techniques of point insertion have been considered. The partitioning of the domain is done using planes perpendicular to the Cartesian axes, operating on a priori discretized boundary and a background grid. Each sub-domain is assigned to a processor, and elements that are shared between adjacent sub-domains are *duplicated* on relevant processors. Points are inserted within each sub-domain until a prescribed number of elements have been generated. The grid is then balanced to ensure a better distribution of the workload employing

an element migration procedure. The load balancing is achieved based on solving a linear system of equations, derived from a spring analogy, in order to determine the position of the partitioning planes.

The element migration procedure itself is carried out in three separate stages. In the first stage, elements that are to be migrated are checked if they can be transferred to their destination processor or not, which requires a *lock* to be made on all processors, excluding the sender and receiver. The second stage consists of the data transfer itself, which involves packing the relevant information, sending and receiving operations between two processors. Update procedure is carried out in the third stage, and then an 'appropriate' message is sent out to *unlock* processors that were prohibited from sending or receiving. The paper presents one example of a triangular grid consists of 1 million element for the NACA-0012 airfoil. Operating on IBM SP2 system, and using the Active Messages [5], the grid was generated on 8 processors within 10 minutes. The CPU time spent on the generation procedure itself is almost one third of the total time, while the rest is spent on the load balancing and communications.

Apparently, algorithms such as the above would involve a considerable amount of inter-processors communication and a high risk of having too many idle processors. In fact, overlapping between computation and communication is a must for algorithms in this category. On the other hand, element quality can be as good as in any other grid generated sequentially. The overall scalability is very questionable, particularly in the three dimensional space applications. The cost of the dynamic load balancing procedure, including the three stages of the element migration scheme, is expected to grow substantially. However, although the idea of extending the algorithm above into the 3D space was proposed in [97], no further reporting could be found.

Post-gridding of the internal boundary

Clearly, algorithms under this category would split the computational domain into a number of sub-domains first, produce a grid *inside* each sub-domain and then deal with the sub-domains interface region. Löhner *et al* in 1992 have presented an algorithm in which the computational domain is subdivided using a background grid, resulted sub-domains are then gridded using the Advancing Front Method. Each sub-domain is gridded separately following the 'In-Out' strategy, in which the 'Front' starts from the inside of a sub-domain and stops just before reaching the adjacent sub-domains. Regions formed between 'uncompleted' sub-domains grid are called the interface regions, and they are

[5] Active Message is a low-latency communication mechanism that minimises overheads and allows communication and computation to overlap. It is used by MPI as its underlying communications layer, [15].

gridded at the end as a post-processing step also using the Advancing Front Method. The algorithm has been demonstrated originally for 2D applications in [82] and then extended into the three dimensions in [115].

"When generating the inter-sub-domain regions, one cannot generate at the same time the interfaces of all neighbours for a certain sub-domain. Therefore, a contingency list was implemented that avoids these conflicts....", [82]. This necessary contingency test implies that if a sub-domain is used to grid an interface, it cannot be used to grid another one. Therefore, the number of interfaces that can be processed in parallel will always be very small, which leads to having plenty of 'idle processors' and thus a significant impact on the overall performance. In fact, although an *improved* way to grid the interfaces has been developed and implemented later on in the 3D algorithm [115], the effect of this problem remains very clear. Another disadvantage in this algorithm is the need to generate and partition a background grid of a size that depends upon the input surface mesh. In another words, the algorithm has a serious issue in respect to the scalability with grid size. In addition, elements quality in the interface regions may deteriorate, particularly when number of sub-domains or the surface to volume ratio per sub-domain grows.

Another algorithm in this category is the one developed by de-Cougny and Shephard and presented initially in 1994, [20] and later on in 1999 after introducing some further developments into the parallelisation of the partitioning procedure, [21]. The algorithm, in general, represents a parallel implementation of a typical octree-based grid generator such as the ones presented in [114, 151], see Section 2.2 for a brief description. In fact, this algorithm can be seen as a modified version of the one discussed above, [82, 115], considering that the octree in here represents the background grid in the other algorithm. However, early version of this algorithm had a very serious problem in terms of scalability, where a serial octree data structure (duplicated over all processors) was used. The issue has been addressed in the later version where the tree partitioning is performed by a parallel recursive inertial bisection procedure.

Clearly, the octree structure supports the concept of distributing the workload onto a set of processors. In fact, the algorithm benefits substantially from the concept of 'processor sets', which is a feature exist in MPI where a number of processors can be grouped and defined as an independent set. Processor sets are split into sub-sets and current terminal octants are assigned to subsets based on the associated estimated workload. Once all processor sets are reduced to single processors, each processor stores a 'coarse' tree. Although this may sound as an ideal condition for achieving an optimum speed up factor a rather disappointing performance was recorded in [21], e.g. a speedup factor in the order of 1.6 using up to 32 processors.

In general, this algorithm suffers badly from the constant need for repartitioning due to the unknown workload associated with evolving octree. Impact of

the partitioning procedure on the element quality was not addressed, though it is highly likely to be severely effected. The largest generated grid reported in [21] is in the order of 2-3 million elements, generated on SGI Onyx machine with 8 processors. No information about the memory available on the machine was reported. A target for producing grids in the order of 10 million elements on the same mentioned machine was reported.

Pre-gridding of the internal boundary

In this category, the internal boundaries are gridded first and at least one *closed* sub-domain is constructed before any discretization of the original domain space can start. In fact, since the sub-domains are first prepared as a set of standard independent grid generation tasks a number of advantages of this strategy in particular can be identified:

- Since all the sub-domain grids are accomplished while operating on a local scale per sub-domain, grids of much larger size can be generated. That is unlike the previous categories where some procedures are carried out acting either on the global grid or on a few sub-domains simultaneously, e.g. the gridding of the interface regions in [115].
- There is more flexibility in enhancing the grid quality on the internal boundary prior to gridding the sub-domains themselves, therefore there will be a better opportunity for improving the quality in the global grid.
- Cost of inter-processor communication can be negligible, particularly if no load redistribution technique is integrated into the sub-domains gridding procedure.
- Algorithms in such a category can accept any type of existing sequential grid generators, e.g. Delaunay or Advancing Front based generators, to be integrated for generating the sub-domains grid.

The earliest algorithm in this category was developed by Weatherill and Verhoeven [128, 129] [6], upon which one of the algorithms developed in this research is built. The algorithm will be discussed thoroughly in the next chapter therefore there is no need to emphasise it here. However, it is still appropriate to report that this algorithm carries out the domain decomposition by employing a greedy type grid partitioner [30], acting on a coarse grid constructed by Delaunay triangulation of the boundary points. Having the internal boundary discretized and the closed boundary for each sub-domain built, a standard

[6] An early form of this algorithm is presented in another paper in the proceeding of the 5th international conference on numerical grid generation [43].

sequential Delaunay grid generator is then employed to grid individual sub-domains independently. In fact, whilst the planar applications of this algorithm showed a high level of grid quality and robustness, unfortunately, the extension into the three dimensional space has proved to be a rather problematic approach. The original algorithm and the extension into the three dimensions with the associated issues are to be discussed thoroughly next chapter.

Hodgson and Jimack have demonstrated in [63] an algorithm for producing triangular planar grids using a Delaunay based technique. The domain partitioning is carried out in terms of the 'graph theory' acting on a weighted dual graph which is constructed from the background grid elements. In fact, every internal boundary in this algorithm is gridded twice, independently with each sub-domain. The algorithm incorrectly considers that: "There is no problem with vertices generated by two different processors not coinciding along the same background edge, since the method used to position the nodes creates a unique discretization of that edge...."!, [63]. Of course, the implications of such an assumption would become more problematic when the program portability is considered, particularly if the algorithm is extended to cover the three dimensional applications. Nevertheless, triangular grids of size in the order of less than 1 million elements, partitioned into up to 8 sub-domains, have been presented.

Galtier and George subdivide the computational domain by employing a set of 'separators', i.e. ".... an imaginary surface that separates the set of points. Such a surface can be a plane, a sphere or even something much more complex ..." [44]. The technique used to discretize the separators is based on a concept referred to as a *projective* Delaunay, in which the Voronoi diagram is constructed on the separator surface in the 3D space. In fact, the paper identifies this procedure as a rather problematic one, mainly due to the difficulties in reserving edges on the original boundary (i.e. boundary recovery problem). "Another practical problem was encountered because of the non-conformity of the original skin (i.e. the surface boundary)..... This proved to be a very hard problem, therefore the corresponding difficulties in projective Delaunay triangulation needs some attention. .... "

In fact, another practical problem has been identified in the paper, which is also due to the way used in defining and gridding the separators. "Indeed, since the generated separator is triangulated by Delaunay, it is not possible that two faces of it intersect. But some faces of the separator may intersect with faces that describe the boundary of the original domain." [44].

Grids of size in the order of 3 million elements generated within two sub-domains are presented. A detailed breakdown of time required in the partition and gridding procedures is available in the paper, however, ".... some examples lead to disastrous CPU time when being pre-partitioned. This is due to the fact that when a projective Delaunay triangulation that fits to the origi-

nal boundary cannot be found, the separator generation process is restarted from zero." Other issues such as the element quality and load balancing are mentioned very briefly in the paper, but without any solution been proposed.

2.5 Concluding Remarks

Essential concepts in the fields of unstructured grid generation and parallel processing technology have been discussed very briefly in this chapter. An algorithm for generating unstructured grids based on the Delaunay triangulation technique has been illustrated, also different types of parallel machines and parallel programming models have been introduced.

An overview of the literature that reported on the parallel unstructured grid generation research activities over the last two decads has been presented. Two different strategies are identified whilst only one of them, known as the geometrical partitioning approach, has proved to be more effective and favourite. A number of different algorithms that adopt this approach have been discussed after been grouped in three different categories. The method used to carry out the partitioning of the computational domain, particularly the order of gridding the internal boundary within the overall procedure, was considered as main criterion for the classifications. However, the chapter concludes: the category defined as 'pre-gridding of the internal boundary' has some advantages in comparison to the others (see the list presented in page 33). In fact, both algorithms developed in this research, which are discussed in details next chapter, fall under this category.

In general, the outcome from most of the algorithms reported in the literature so far has not managed to maintain a satisfactory level of performance. The size of grids generated is still in the order of a few million elements only, and speedup factor achieved is less than 2 in some cases. In fact, size of grids generated in parallel and presented in some of the reviewed literature has been less than what already could be generated sequentially on normal computers!. Thus, apparently, the challenge for generating large size grids efficiently remains to be solved, and further investigation is certainly needed.

Chapter 3

A Geometrical Approach for Unstructured Parallel Grid Generation

3.1 Introduction

The geometrical partitioning approach has been adopted as our strategy to address the issue of generating large size grids. Two methods fall under within this category, but differ at the domain decomposition and the internal boundary discretization procedures, are presented in this chapter. Both methods assume that a triangular grid has already been generated on the computational domain boundary. In order to carry out the domain decomposition task in the first method an 'initial' tetrahedral grid is needed, which is constructed using Delaunay technique to connect the boundary grid points. Whilst in the second method the decomposition procedure operates directly on the boundary triangular grid. Therefore, we will refer to the first method by '*the indirect decomposition method*' and to the second as '*direct decomposition method*'.

The chapter is divided into two major sections, which are associated with the two methods mentioned above. Each section starts by a brief review of the algorithm using a simple triangular grid for illustration. Details about individual procedures are then presented in separate sub-sections. The construction of a tetrahedral grid for a real-world engineering problem is demonstrated after the method has been discussed. The implementation of these two methods using the parallel processing technology is not covered in this chapter, since it will be discussed in depth in the next chapter.

3.2 The Indirect Decomposition Method

This method had already been developed and implemented on triangular grids by Weatherill and Verhoeven when this study began [43, 128, 129]. Most of the work carried out on this method during this study consist of extending the original algorithm such that it covers both triangular and tetrahedral grids. However, a triangular grid generated by the original algorithm is presented here for illustration purposes.

3.2.1 An Overview of the General Algorithm

The key concept of this method is to partition the computational domain by exploiting an 'initial grid', which is an unstructured grid built by using the Delaunay triangulation technique on the boundary points. No points are introduced inside the domain during the construction of this grid, see (c) in Figure 3.1. The partitioning procedure is carried out using a greedy-type algorithm [30], in which the total area/volume of the computational domain is divided equally into a number of sub-domains. This procedure starts from an element in the initial grid and then moves to its immediate neighbours while accumulating the individual elements area/volume, and so on until every element in the grid is assigned to a sub-domain, see (d) in Figure 3.1.

A set of new 'internal boundaries' is formed by extracting all edges/triangles that have two adjacent elements that belong to two different sub-domains. An appropriate discretization algorithm has to be applied on these internal boundaries, which must consider the grid point spacing sources and their effects[1], see (e) in Figure 3.1. Thus, after having the internal boundary grids attached back to the original boundary grid, every sub-domain becomes an independent grid generation task, see (f) in Figure 3.1. Any normal sequential grid generator can now be used in order to generate the sub-domain grids.

3.2.2 Domain Decomposition

The domain decomposition procedure in the indirect decomposition method comprises a straightforward implementation of a grid partitioning algorithm operating on the initial volume grid. A considerable number of grid partitioning algorithms has been developed over the last two decades [6, 30, 73, 83, 112, 132]. Differences between these algorithms include various aspects such as: the basic approach, requirements of computing resources, quality of the partitioned grid, performance of the algorithm overall, ... etc. It is certainly

[1] For more information on the grid points spacing sources, the reader is advised to consult Appendix A.

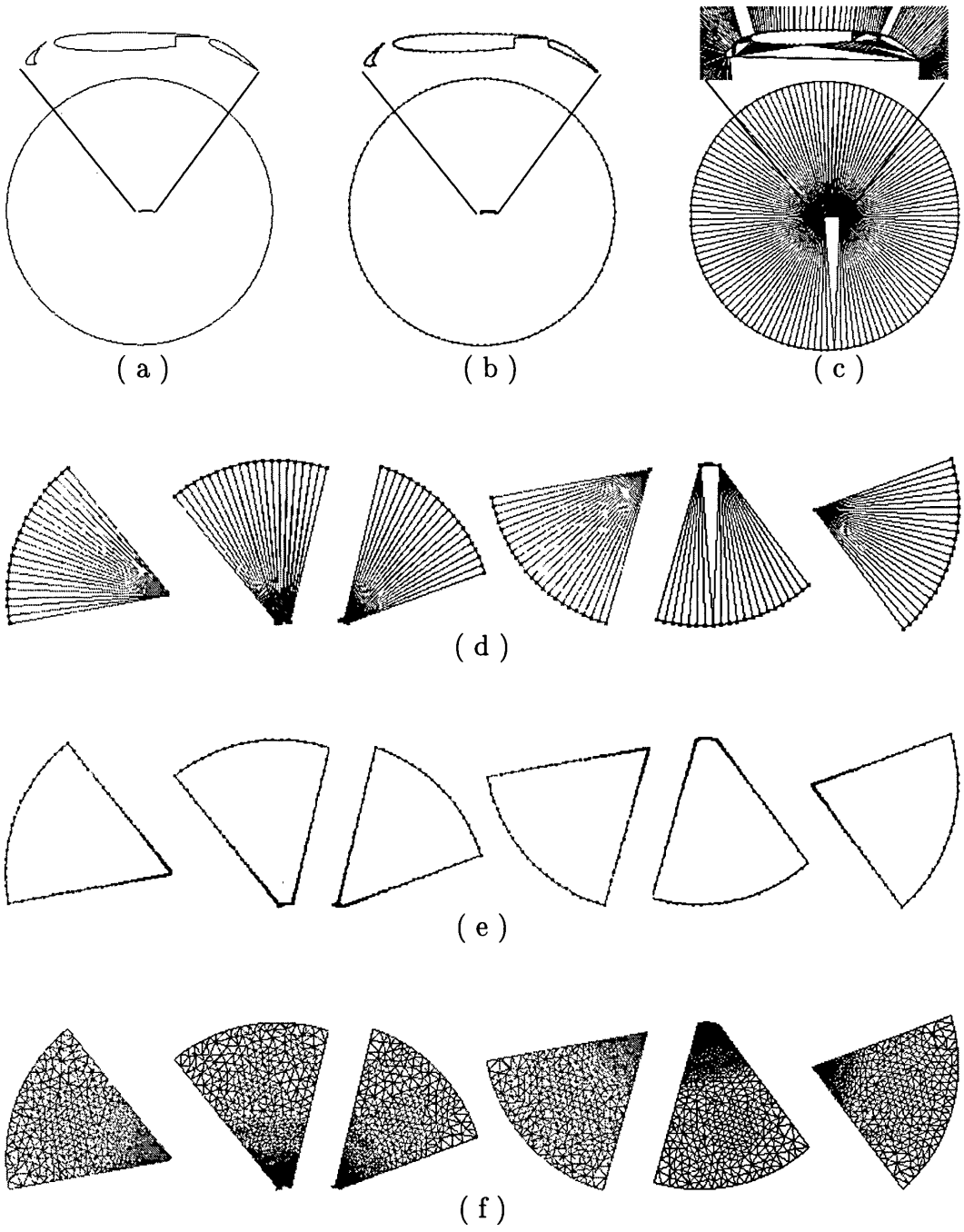


Figure 3.1: Illustration of the major steps in the indirect decomposition method with multi-elements airfoil configuration: (a) Geometrical definition of the configuration, (b) Discretized boundary as a set of points and edges, (c) Delaunay triangulation for the boundary points and edges, (d) Result of domain decomposition, sub-domains with ungridded internal boundary, (e) Closed boundary of individual sub-domains, internal boundary gridded considering the effects of grid point density sources , (f) individual sub-domains grid.

beyond our intention here to get involved in any form of comprehensive discussion or comparison of these algorithms; however, some brief comments are still appropriate. A reader who is interested in such comparisons can consult one of the references [65, 83, 134].

In general, a high quality partitioning of a grid is identified by: (a) well balanced distribution of the total workload among the sub-domains and (b) minimization of the inter-domain communication cost. Grid partitioning algorithms based on the multilevel partitioning approach in general are considered to be very good in producing well balanced load distributions with a minimized inter-domain communication cost. In this approach, a hierarchy of coarse grids is produced by merging together groups of neighboring vertices in the previous grid. Thus, the partitioning procedure is applied on a rather coarse grid; the final sub-domains are then obtained by mapping results of the partitioned coarse grid back to the original fine grid. Metis [73, 74, 75] and Jostel [132, 133, 134] are well known algorithms in this category. Graphical based algorithms are also available such as the Recursive Spectral Bisection (RSB) algorithm [112], and some of its derivatives as in [6]. Indeed, the grid partitioning discipline can be considered as an independent active area of research.

In fact, most of the algorithms mentioned above are considered to be very expensive in respect to their requirements of computing resources, see Table 1.1 in page 11 for typical figures of the Recursive Spectral Bisection (RSB) algorithm [112]. The situation is far worse in the multilevel partitioning algorithms case, for example Metis demands about 240 MBytes to partition a tetrahedral grid with 1 million vertices [74]. Obviously, introducing a grid partitioning procedure that turns the domain decomposition step into a bottleneck of the parallel grid generator is not appreciated. With the interest of minimizing the amount of computing memory needed at every step throughout the general algorithm, not surprisingly, none of the algorithms above was viewed favorably. Furthermore, due to the nature of the initial tetrahedral grid many of the advantages associated with these algorithms become not applicable. For example, advantages gained from the grid coarsening procedure in the multilevel approach will not be very effective since there are no internal points to gather inside the initial grid.

A rather simple, fast and very memory efficient algorithm, known as the Greedy algorithm, has been adopted to carry out the partitioning of the initial tetrahedral grid. "It is referred to as 'Greedy' algorithm because it essentially 'bites' into the mesh in order to construct the sub-domains" [32]. This algorithm is much less expensive computationally than any other algorithm. It only requires an extended form of the grid connectivity data, which consists of a list of all neighbouring elements in addition to the original elements connectivity matrix, [30, 31, 32].

Pseudo code for the Greedy domain decomposition algorithm

```
(1) Construct the element-neighbours tree of the initial
    volume grid (Ne_tree);
(2) Calculate the total volume of the computational domain
    (Vol_tot);
(3) Calculate volume per sub-domain
    (Vol_sub = Vol_tot / Nsub_tot), where (Nsub_tot) is the
    total number of sub-domains

Initialise the 'Front' by choosing an element and make
     In_sub = 1;

while(In_sub < Nsub_tot){

Acc_vol = 0;
 (4) Loop over elements in the 'Front';
     and for every element:

   (4-1) using the (Ne_tree) access each adjacent element (ie);
   (4-2) if (ie) has not been assigned to a sub-domain already
         then attach it to the current on;
   (4-3) add the volume of (ie) to Acc_vol;
   (4-4) update the 'Front' by considering the (ie)
         elements in (4-3) only;
   (4-5) if (Acc_vol >= Vol_sub) then break the loop,
         otherwise back to (4);

 (5) A new sub-domain has been established; ++In_sub;

 (6) If the 'Front' is empty and the decomposition is not
     complete yet then a 're-start' procedure is called.
}
```

The 're-start' procedure mentioned in step (6) may have to compromise on the equal volume criterion in order to ensure that no disjoint sub-domains are created. Also, it must ensure that no element has been left out, and the desired number of sub-domains has been achieved.

Whilst this algorithm always showed a very reasonable performance in the two dimensional applications, it has been far from ideal in the three dimensional case. An acceptable level of load balance could not be achieved very often,

despite several attempts to enhance the overall performance by introducing a load redistribution procedure. It has been noticed that the main reason behind the poor performance of the Greedy algorithm in three dimensions is due to the nature of the initial tetrahedral grid itself. The extreme irregularity in the elements shape and volume plays a major role in this issue. A typical example of the element volume distribution in an initial tetrahedral grid is presented in Table 3.1. Clearly, more than 99% of the entire domain volume is filled by a very small number of elements (i.e. less than 1.4% of total number of elements), see columns 2 and 3 in the first line in the Table 3.1.

Domain decomposition with 'weighted' elements

Unfortunately, the Greedy algorithm presented above may produce a highly unbalanced distribution of the total workload among the sub-domains. It does not take into consideration the impact of the background grid and point spacing sources on the workload associated with each sub-domain. For example, if two sub-domains with exactly the same volume have different grid point spacing parameters they will certainly have different 'workload' from the grid generation procedure view point. However, an attempt to improve the above algorithm performance has been made. Particularly, by replacing the element volume by a 'function' such as $\tilde{v}$, which represents the real workload associated with the total number of elements that is expected to be generated inside each element.

$$\tilde{v} = \int_v \frac{1}{df} \; dv \tag{3.1}$$

where df is the value of the grid points spacing function. In fact, several other functions have also been investigated but, unfortunately, due to the extreme heterogeneous distribution in the elemental volumes, the impact of introducing a function instead of the volume has been very limited. Very little improvements on the final workload distribution has been encountered. For comparison and further illustration purposes details about the above function are presented next to the statistics of the pure element volume case, see Table 3.1. Clearly, the impact on the overall balance is very limited, where it remains that more than 99% of the total workload is still associated with less than 7% of the total number of elements in the initial grid (see columns 5 and 6 in the first row of Table 3.1. In other words, it is still a very high possibility to have the work load of a sub-domain doubling up by just simply adding one single element to the sub-domain during the partitioning.

$v_p < v_e/V\%$	$n_e/N_e\%$	$\sum_{i=1}^{n_e} v_{p,i}$	$\tilde{v}_p < \tilde{v}_e/\tilde{V}\%$	$n_e/N_e\%$	$\sum_{i=1}^{n_e} \tilde{v}_{p,i}$
0.029	99.150	1.399	0.027	99.130	6.847
0.058	0.373	15.700	0.054	0.384	15.170
0.087	0.166	14.490	0.082	0.167	13.670
0.116	0.052	5.941	0.109	0.053	5.667
0.145	0.064	9.951	0.136	0.064	9.341
0.174	0.082	14.88	0.163	0.082	13.960
0.203	0.030	6.447	0.190	0.030	6.052
0.232	0.017	4.321	0.217	0.017	4.056
0.260	0.009	2.443	0.245	0.009	2.294
0.289	0.009	2.763	0.272	0.009	2.594
0.318	0.025	8.650	0.300	0.025	8.120
0.347	0.003	1.344	0.326	0.003	1.261
0.376	0.013	5.398	0.353	0.013	5.067
0.405	0.007	3.105	0.380	0.007	2.914
0.434	0.002	0.862	0.408	0.002	0.809
0.463	0.000	0.000	0.435	0.000	0.000
0.492	0.000	0.000	0.462	0.000	0.000
0.521	0.000	0.000	0.489	0.000	0.000
0.585	0.003	2.312	0.549	0.003	2.170
1.000	0.000	0.000	1.000	0.000	0.000

Table 3.1: Details about the volume distribution in the initial tetrahedral grid of a CFD model (i.e. B60 inside a box) compared with 'weighted volume' distribution. Notice that: v_e is an element volume, V is the total volume of the domain, n_e number of elements in the associated category, N_e is the total number of elements. Same fields are repeated for the weighted volume case in the columns on the right hand side.

3.2.3 Discretizing the Internal Boundary

Having the sub-domains formed as a group of clustered tetrahedra in the initial grid, the inter-domain boundaries are then formed by extracting all triangular planar faces that are shared between adjacent sub-domains. In sharp contrast to the two dimensional applications, discretizing the new internal boundaries is not an easy task. Whilst it is a simple problem of discretizing a straight line in the 2D applications, in the 3D case, it involves generating a smooth triangular grid on a surface which is already defined in a discretized form. Such a surface is represented by a set of triangles which are extremely distorted as individuals and, in addition, gathered in a very irregular manner; see Figure 3.2. "Perhaps the most technically challenging aspect of the parallel mesh generation for three- dimensional applications is in the construction of the inter-domain boundary grid on these highly irregular surfaces" [148]. A typical example

of an internal boundary is presented in Figure 3.2, where a general view of the initial grid and the final grid is presented in (a); a close up of each grid is presented in (b) and (c) respectively. In order to generate such a smooth triangular grid, a number of sub-tasks have to be carried out systematically; a brief discussion about each sub-task is presented in this section:

Validation of the internal boundary topology

The initial set of triangular faces that forms the internal boundaries is, in general, very irregular and has a high number of triangles with very small angles. The main reason behind this is the nature of the tetrahedral grid itself, as it is been constructed on the boundary points. Statistics of the dihedral angle in the initial grid of three different examples are presented in Figure 3.3. Two of these examples involve a complex configuration of a civilian aircraft (i.e. Falcon and B60), whilst the third one consists of a simple configuration of a box. Although, the complexity of the configuration plays a major role in the quality of the initial grid, clearly, the situation in a simple configuration is not that much better[2]. A few attempts have been made to improve the quality of the initial grid, mainly by allowing for different levels of point insertion inside the domain. It was observed that a relatively 'fine' grid is needed before any noticeable improvements are achieved. Of course, such an option can not be considered since it may cause a serious problem in respect to the scalability . In short, having a poor quality tetrahedral grid as an 'initial grid' seems to be inevitable.

In some cases, according to the results of the domain decomposition procedure, the initial definition of the internal boundary surfaces may have edges shared among more than two triangles and hence form so-called multi-shared edges. The initial surfaces are considered as valid topology as long as they do not have any intersected faces (i,e. a multi-shared edge). Therefore any internal boundary that has a surface triangulation with a multi-shared edge is considered as an invalid boundary.

A smoothing technique is applied on the interface tetrahedra. Any tetrahedron which belongs to domain j and neighboured by three tetrahedra from domain i is moved from j to i. Such a smoothing scheme improves the configuration of the initial set of triangular faces and, in turn, helps to improve the quality of the final grid. An example of the inter-domain boundary smoothing procedure on the tetrahedra at the interface regions on the final triangular grid is illustrated in Figure 3.4. Such a smoothing procedure on the internal boundary cannot always eliminate the multi-shared edges. Therefore, another scheme depends on re-arranging the cluster of tetrahedra connected to these multi-shared edges

[2]The reader can compare the statistics of the initial grid as presented in here with the same statistics in a typical tetrahedral grid as presented in Figure 5.5 page 109.

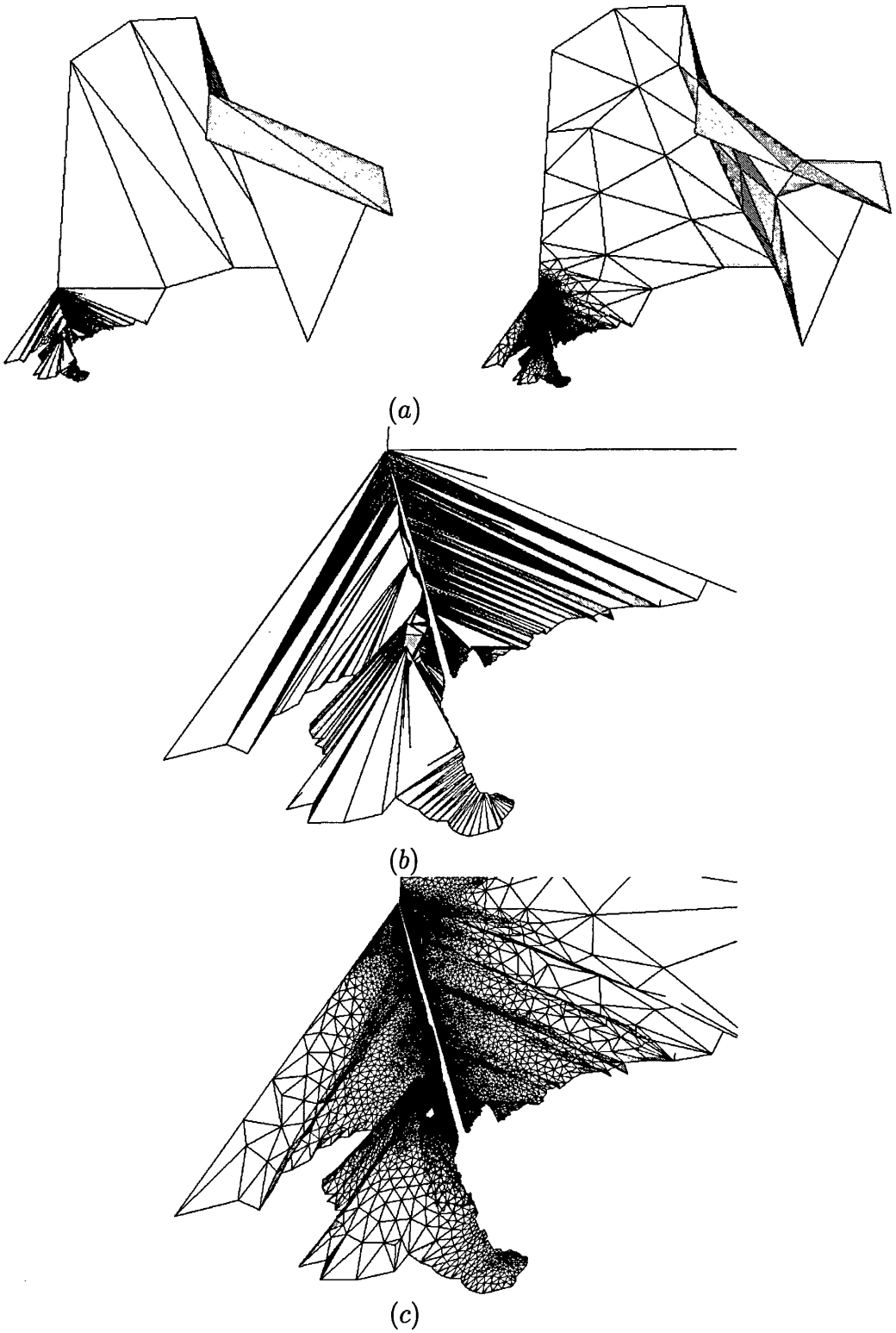


Figure 3.2: An example of generating smooth triangular grids on the internal boundary. (a) The initial form of the discretized surface (left) and final triangular grid (right), (b) and (c) are close-ups from the above respectively.

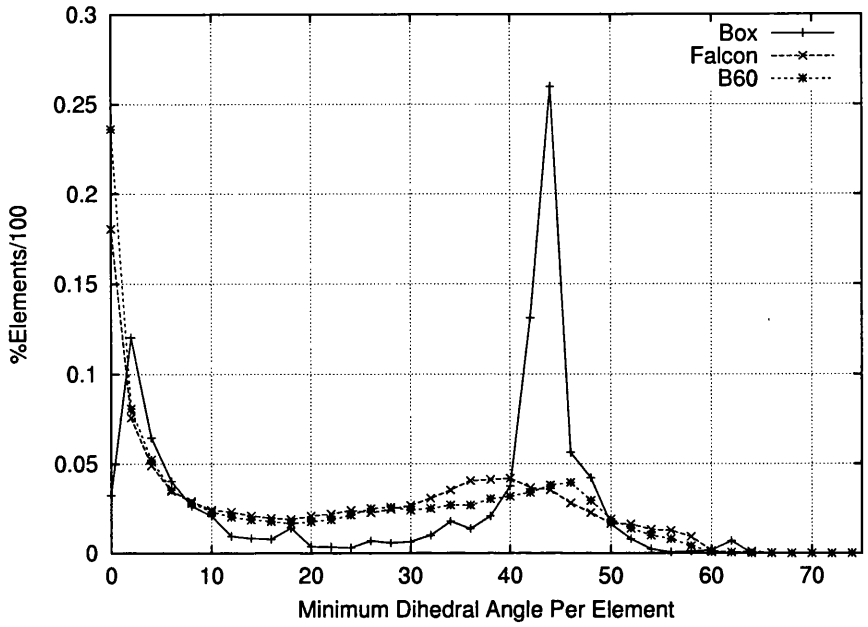


Figure 3.3: Statistics of the minimum dihedral angle per element in the initial tetrahedra grid for three different configurations.

is introduced. Figure 3.5 and Figure 3.6 illustrate an example of the multi-shared edge problem and how it can be solved.

An edge on the internal boundary surface between sub-domains i and j is shared between three faces, see (II) in Figure 3.5. The target is to rebuild the internal boundary such that the problematic edge is eliminated. The process consists of reorganizing the cluster of tetrahedra around the edge in a certain manner. Starting from a tetrahedron in the cluster which belongs to domain j and interfaces with another tetrahedron belonging to domain i, then by looping over the cluster of tetrahedra in a fixed direction (see (I) in Figure 3.5) any tetrahedron that belongs to j and interfaces with i is defined as a new artificial sub-domain. This is then given a new number M, where $M = N + l$; N is the total number of original sub-domains and l is the total number of artificial sub-domains introduced so far, see (I) in Figure 3.6.

The new construction of tetrahedra gives new internal boundary surfaces. The old internal boundary surface between domain i and j now has a different triangulation where the previous multi-shared edge is shared between two faces only, see (II) in Figure3.6. The internal boundary surfaces between domain j and the new artificial sub-domain (i.e. M) is not gridded at all. The new artificial sub-domains are linked back to their mother sub-domain before the start of the volume grid generation procedure. In other words, the total number sub-domains remains exactly the same at the end of the topology correction

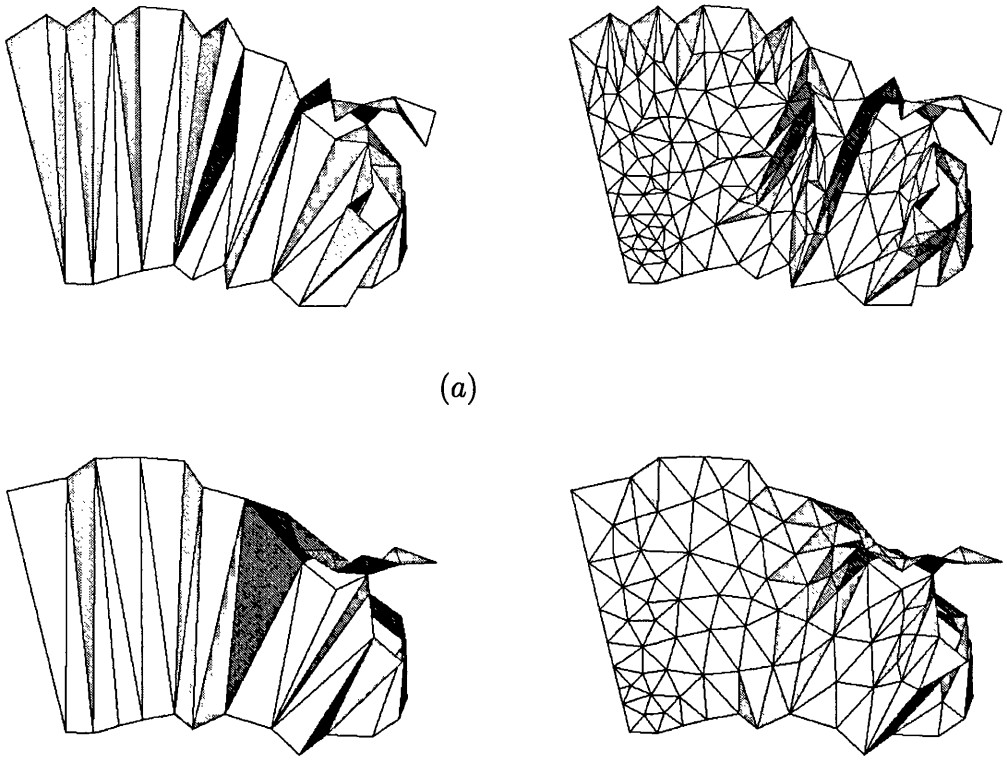


Figure 3.4: Impact of the smoothing procedure applied on the tetrahedra at the interface regions on the final triangular grid. Initial surface (left) and final triangular grid(right), (a) without the smoothing procedure while in (b) with.

procedure, and none of the 'artificial' sub-domains is recognized outside this procedure.

Discretizing the 'global edges'

Edges on internal boundaries might be shared among several sub-domains. Such edges will be referred to as global edges. Since, by definition, more than one surface meets along a global edge, it is necessary, so as to ensure coincident points, to generate nodes on a global edge once and once only. The set of points $\mathbf{X}_k$ along an edge are computed in a recursive manner. Starting from an edge with two initial boundary points at the two ends, one new point is computed using a linear interpolation scheme, see Figure 3.7. In turn, these edges are then sub-divided such that

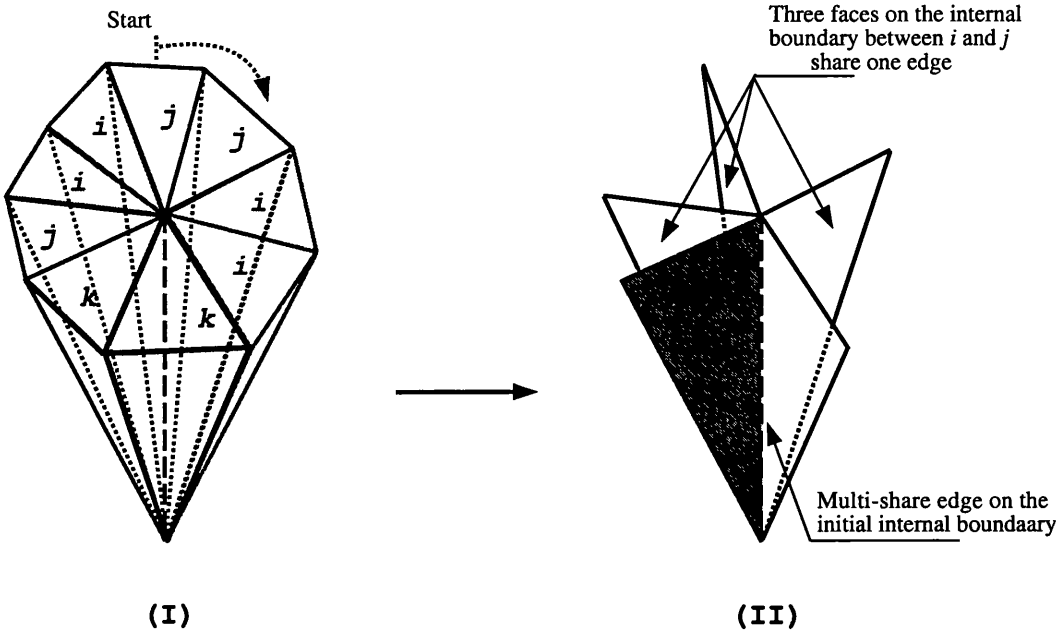


Figure 3.5: Invalid topology of the internal boundary surface between sub-domains i and j, an edge shared among three triangles on the same surface (that is equivalent to the surface-surface intersection problem). Cluster of tetrahedra in (I) and the internal boundary surface(s) in (II).

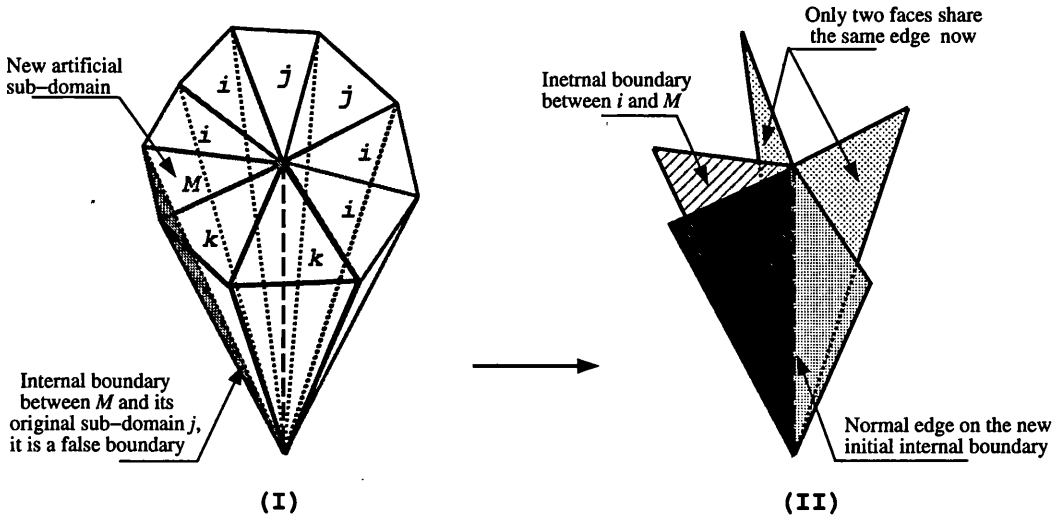


Figure 3.6: A valid surface definition of the internal boundary between sub-domains i and j is obtained by introducing an artificial sub-domain (i.e. M). 'New' internal boundaries associated with the artificial sub-domain are defined and gridded separately, except the boundary between shared with the 'mother' sub-domain which should never be recognized as internal boundary.

$$\mathbf{X}_k = (1-u) * \mathbf{X}_i + u * \mathbf{X}_j \tag{3.2}$$

$$u = \mathbf{df}_i / (\mathbf{df}_i + \mathbf{df}_j) \tag{3.3}$$

where $\mathbf{df}_i$, $\mathbf{df}_j$ are the point distribution functions at the two end points which define the edge. As already mentioned, initially both i and j will be points from the initial surface grid. The point distribution function is computed at the new point also using a linear interpolation scheme.

$$\mathbf{df}_k = (1-u) * \mathbf{df}_i + u * \mathbf{df}_j \tag{3.4}$$

A new point along an edge is rejected when its distance from any of the two end points $\mathbf{D}_{i,j}$ violates the local point distribution function. The value of the point distribution function is controlled by the background grid and the grid point spacing parameters and sources, see Appendix A.

$$\mathbf{D}_{i,j} < min\ (\mathbf{df}_{i,j}) \tag{3.5}$$

The set of new points on every global edge is stored together with the internal boundary surfaces connected to the edge. When the grids on the internal boundary surfaces are generated the computed points on the global edges ensure that the surface grids along the common edge are topologically consistent.

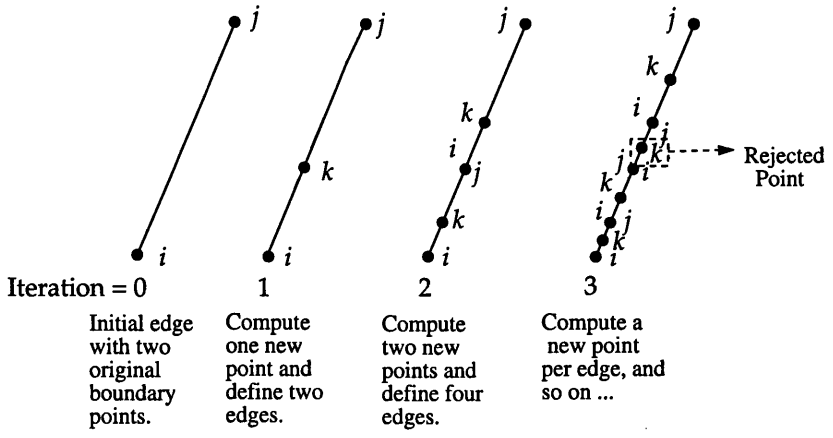


Figure 3.7: Computing new points on a global edge using linear interpolation scheme. Notice that points which violate the local point distribution function are rejected.

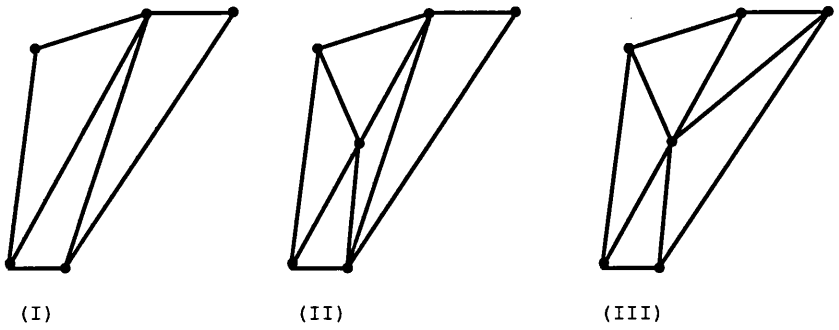


Figure 3.8: An illustration of the point insertion, i.e. (*II*), and edge swapping,i.e. (*III*), technique.

Grid generation on discretized surfaces

Unfortunately, all traditional surface grid generation techniques are designed to operate on surfaces defined by advanced geometrical models which always provide a high level of continuity and smoothness [100]. In general, generating a smooth triangular grid on a surface defined in a discretized form is a rather difficult task, particularly in the three dimensions. This can get far more complicated when the quality of the 'original' discretization is as bad as shown above. The number of algorithms that address the issue of 're-gridding' discretized surfaces is very limited; the author is aware of one algorithm only which is developed by Löhner [85]. In fact, this algorithm has been found to be computationally very expensive, its performance also was found to be heavily dependent on the nature and quality of the original discretized surface. Thus, this algorithm does not really address the main issue in our case, which is the extreme irregularity and poor quality in the initial triangulation. On the other hand a considerable number of efficient and simple grid refinement procedures have been developed over the last two decades [140], upon which our approach is built.

The process been adopted for generating a smooth surface grid on an internal boundary combines three different techniques namely: point insertion, edge swapping and points relaxation [107, 108]. The first technique, i.e. point insertion, introduces new points into an existing triangulation, using linear interpolation on edges wherever the grid point density distribution allows to do so. In fact, this technique has already been discussed when the discretization of the global edges procedure was reviewed, see page 46. The only difference here is that the point interpolation scheme is integrated into the surface gridding procedure itself. When a new point is introduced on an edge, the two neighbouring elements are split into two triangles each by connecting the new point to the third node (i.e. not on the same edge where the interpolated point is) in each element, see (II) in Figure 3.8. It is appropriate to re-emphasis here the importance of considering the effects of the background grid and point spacing

sources. Density and distribution of the grid points on the internal boundary should be consistent with the rest of the original boundary and volume grids.

The element creation procedure associated with the point insertion technique is carried out with very little consideration of geometrical factors. Elements of a rather poor quality can be generated; many triangles with an angle in the order of 1 - 5 degrees may exist. Obviously, such poor quality triangulation on the internal boundary will have a severe impact on the overall volume grid. In order to improve the quality of individual triangles as well as the global smoothness of the internal boundary, two different procedures have been integrated. The first procedure is edge based, in which the quality of every two adjacent elements is inspected and a decision about to swap or not to swap the shared edge is made, see (III) in Figure 3.8. The second procedure is point based, which is known as the Laplacian point relaxation technique as presented in equation 3.6, [35, 41].

$$\mathbf{r}_0^{p+1} = \mathbf{r}_0^p + \frac{\omega}{N_0} \sum_{i=1}^{N_0} (\mathbf{r}_0 - \mathbf{r}_i) \qquad i = 1, \ldots N_0 \tag{3.6}$$

Where $\mathbf{r}_0^p$ is the new position (x, y, z) of the point $\mathbf{r}_0$ after p iterations, N_0 is the number of neighbours points with coordinated $\mathbf{r}_i$ and ω is the relaxation parameter. This technique has proved to be extremely effective in improving element quality and the overall smoothness of internal boundary. For a typical example of such a surface grid on an internal boundary see Figure 3.2 in page 44. Also, another illustrated example is presented in Figure 3.9, where in addition to the initial and final grids on the surface a couple of interim grids are Displayed.

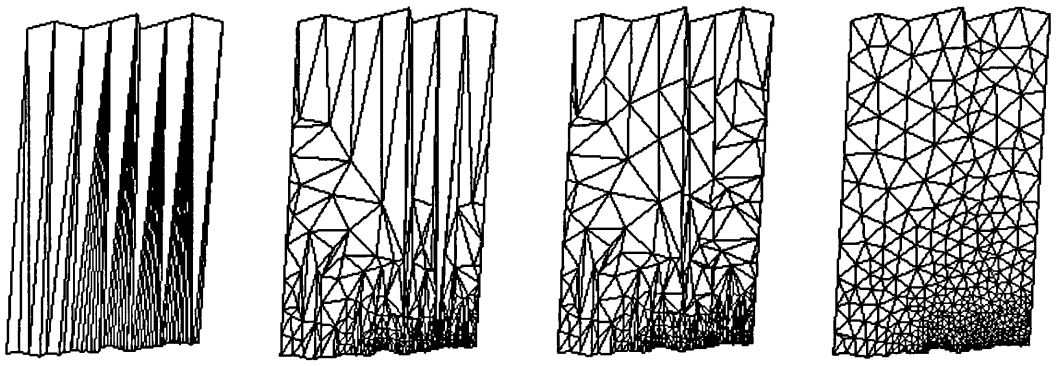


Figure 3.9: A typical example of internal boundary surface grid, snapshots of different stages throughout the procedure.

General algorithm for generating triangular grids on the internal boundary

A flowchart that summaries the general procedure followed in generating the internal boundary surface grids is presented in Figure 3.10. Notice that the grid point relaxation (smoothing) procedure is introduced once only, i.e. at the end when all points have been introduced. In practice the smoothing procedure can be activated more regularly, but not before a considerable number of points is introduced. It has been observed that early implementation of the smoothing procedure may lead into intersection between different internal boundaries if they were too close.

3.2.4 Gridding the Individual Sub-domains

The Delaunay volume grid generator is sensitive to the orientation of the boundary faces, in order to determine the domain to be gridded. Therefore, the boundary faces of all sub-domains have to be correctly oriented. A technique based on a very simple concept can be applied. Two adjacent triangles are considered to have the same orientation if the shared edge exists in order ij on the first triangle, and the same edge ordered ji on the other triangle (See Figure 3.11).

Sub-domain boundaries consist, in general, of a set of original boundary faces and another set of internal boundary faces. Original boundary faces always point in the correct direction, because their orientation is preserved after generating the initial volume grid [107].

The orientation procedure starts from an internal boundary face adjacent to an original boundary face. The internal boundary face is oriented based on the order of the shared edge with the original boundary face. Having this first face oriented correctly, a neighbouring internal boundary face can be then oriented using the same technique (See Figure 3.11). The procedure cycles from one internal boundary triangle to a neighbouring internal boundary face until all internal boundary faces are correctly oriented. The technique is quite reliable whatever the combination of internal boundary faces. The same technique can be used for sub-domains bounded by internal boundary faces only, where any triangle can be chosen as a start, and then a check can be performed to find if the chosen direction is correct or not. Orientation of internal boundary faces is carried out by workers.

Another point which a Delaunay volume grid generator would be very sensitive to is the quality and smoothness of the boundary triangulation; since it plays a major role in the success rate and efficiency of maintaining the boundary integrity at the end of the volume grid generation procedure[3]. This task

[3]For more details about the Delaunay grid generation the reader can consult AppendixA.

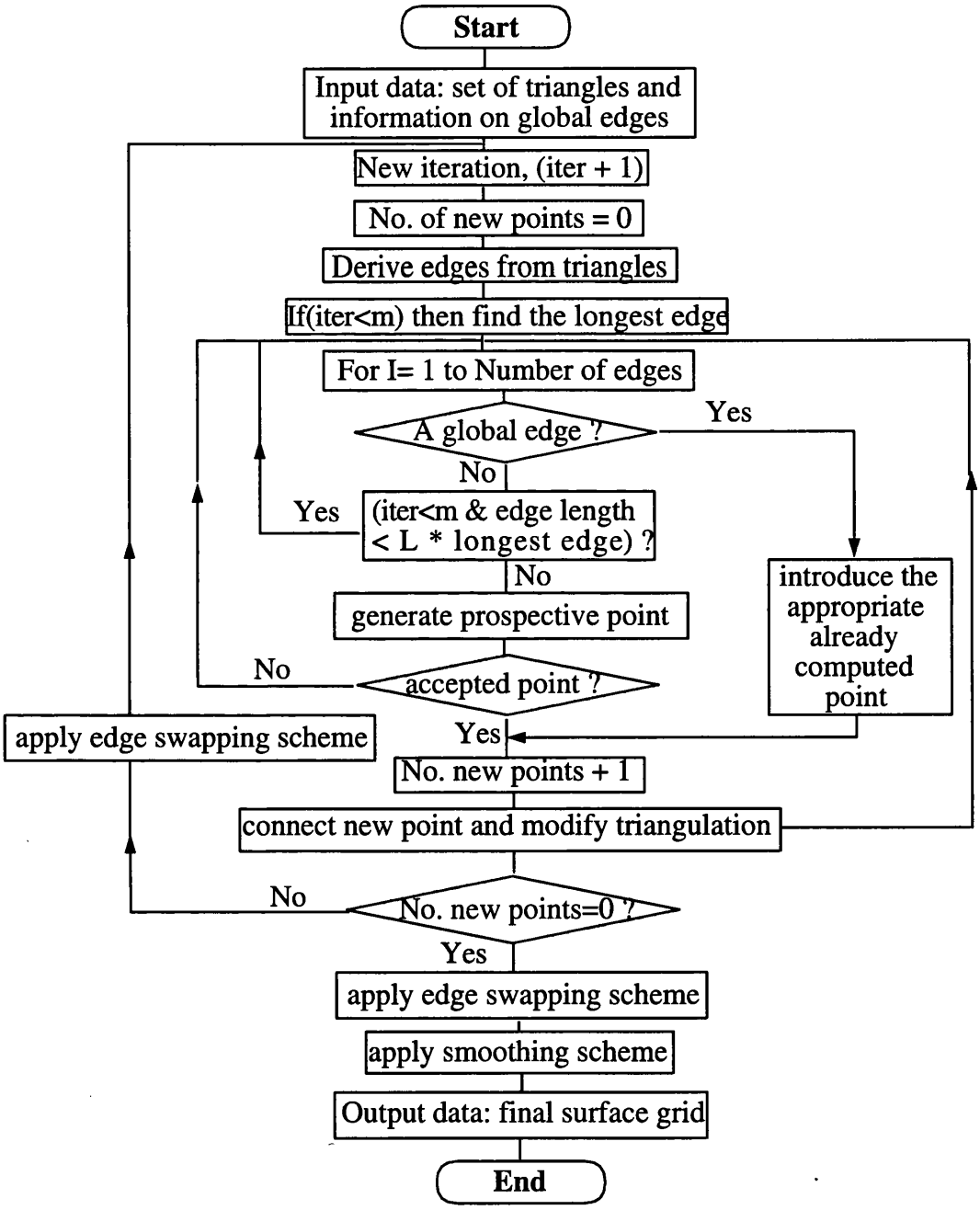


Figure 3.10: Flow chart of the surface grid generation on the inter-domain boundary.

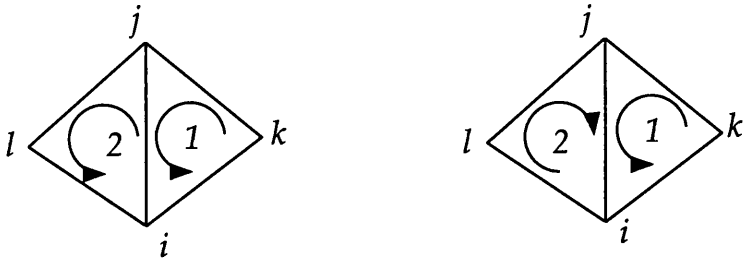


Figure 3.11: Two adjacent faces oriented correctly in (A). Notice the order of the shared edge in every face.

of recovering the boundary triangular grid has been the subject of active research for a number of years [143]. Though a considerable improvement has been achieved, in practice, some problems may still occur whenever a poor quality grid or a complex surface exist on the computational domain boundary. Recalling the extreme irregularity in the shape of the internal boundary obtained in the indirect decomposition method may explain why this method has failed to become a reliable approach for generating tetrahedral grids in parallel [107, 108].

A flowchart of the general algorithm with an illustration example

To conclude the discussions on the indirect decomposition method we present a flowchart summarising all the steps involved in the general algorithm as discussed above, see Figure 3.12. All tasks that need to be carried out after the completion of the sub-domain grid generation are considered as 'post processing' activities, which will be discussed extensively in Chapter 5. In addition, as a *visual* illustration of the algorithm developed, a small size grid for a realistic engineering problem is presented. The configuration consists of an experiment prepared by a research group in Loughborough University in order to carry out some experimental investigation of the air flow around the outlet-intake region of a Harrier Jet. Overview of the geometrical description, the grid points spacing sources and the surface grid on the original boundary are presented respectively in (*a*) and (*b*) in Figure 3.13; the four sub-domains obtained by applying the Greedy algorithm on the initial volume grid are presented in (*c*). Also, a typical example of the initial and final triangulation on an internal boundary is presented in (*d*) in the same Figure. The final surface grids on the four sub-domain boundaries are presented in (*a*) in Figure 3.14, whilst a planar cross section in each sub-domain volume grid is presented in Figure 3.14, (*b*).

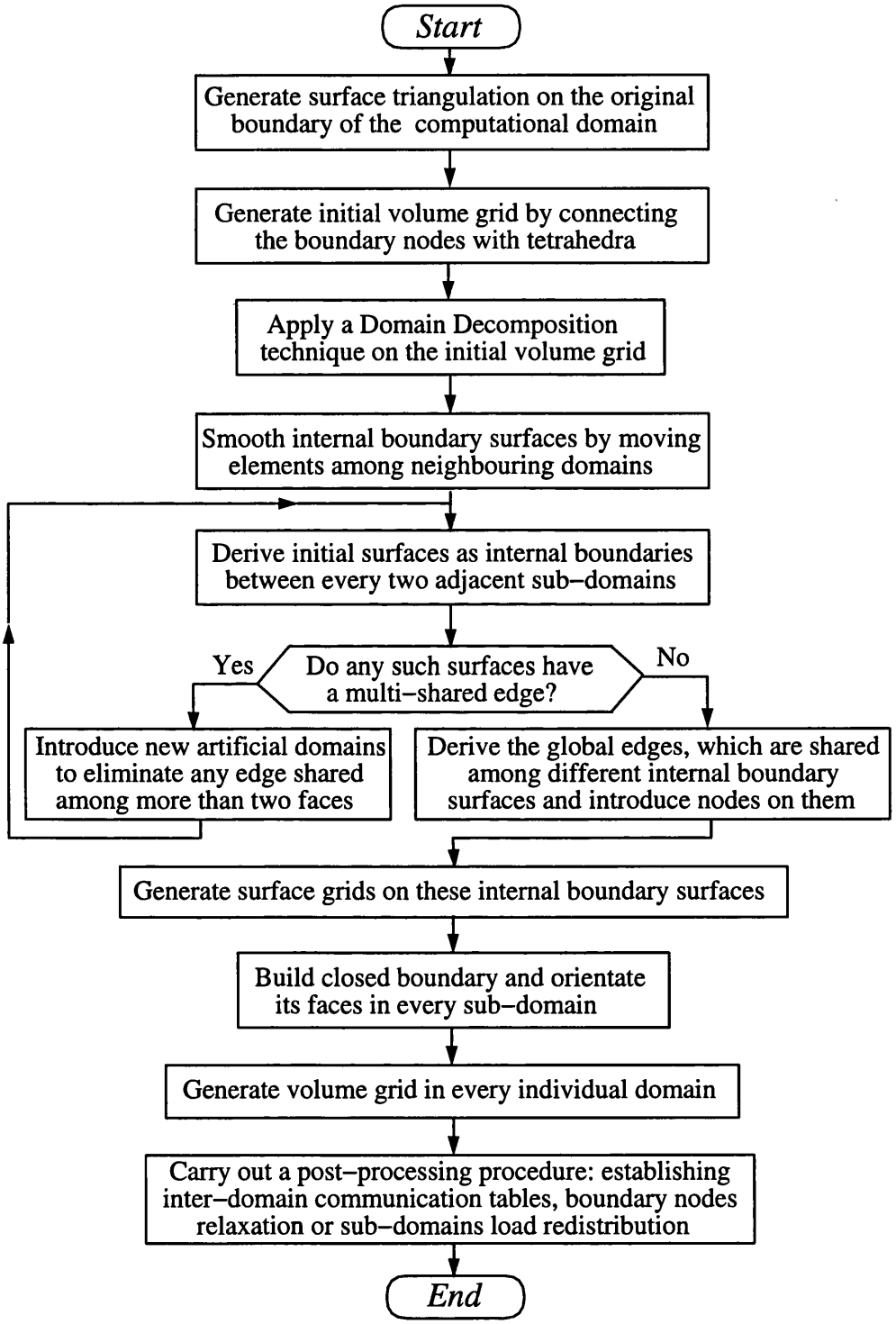


Figure 3.12: Overview of the general algorithm using the Indirect Decomposition Method.

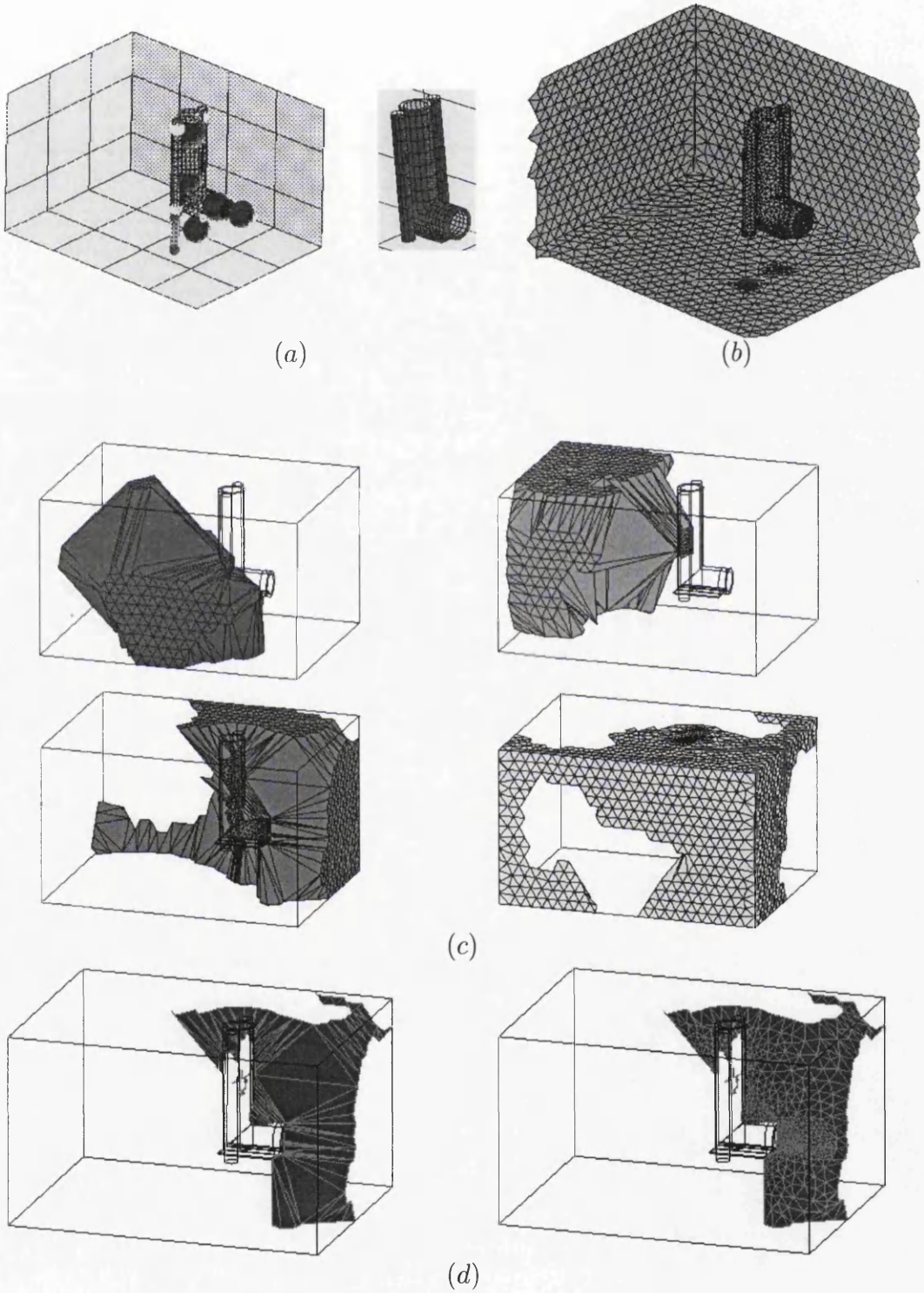


Figure 3.13: Partitioning the computational domain into sub-domains: (a) Geometrical definition with the point spacing sources, (b) Triangular grid on the *original* boundary, (c) Four sub-domains created by applying Greedy algorithm on the *initial* tetrahedral grid. (d) Initial form of an internal boundary and the final triangular grid.

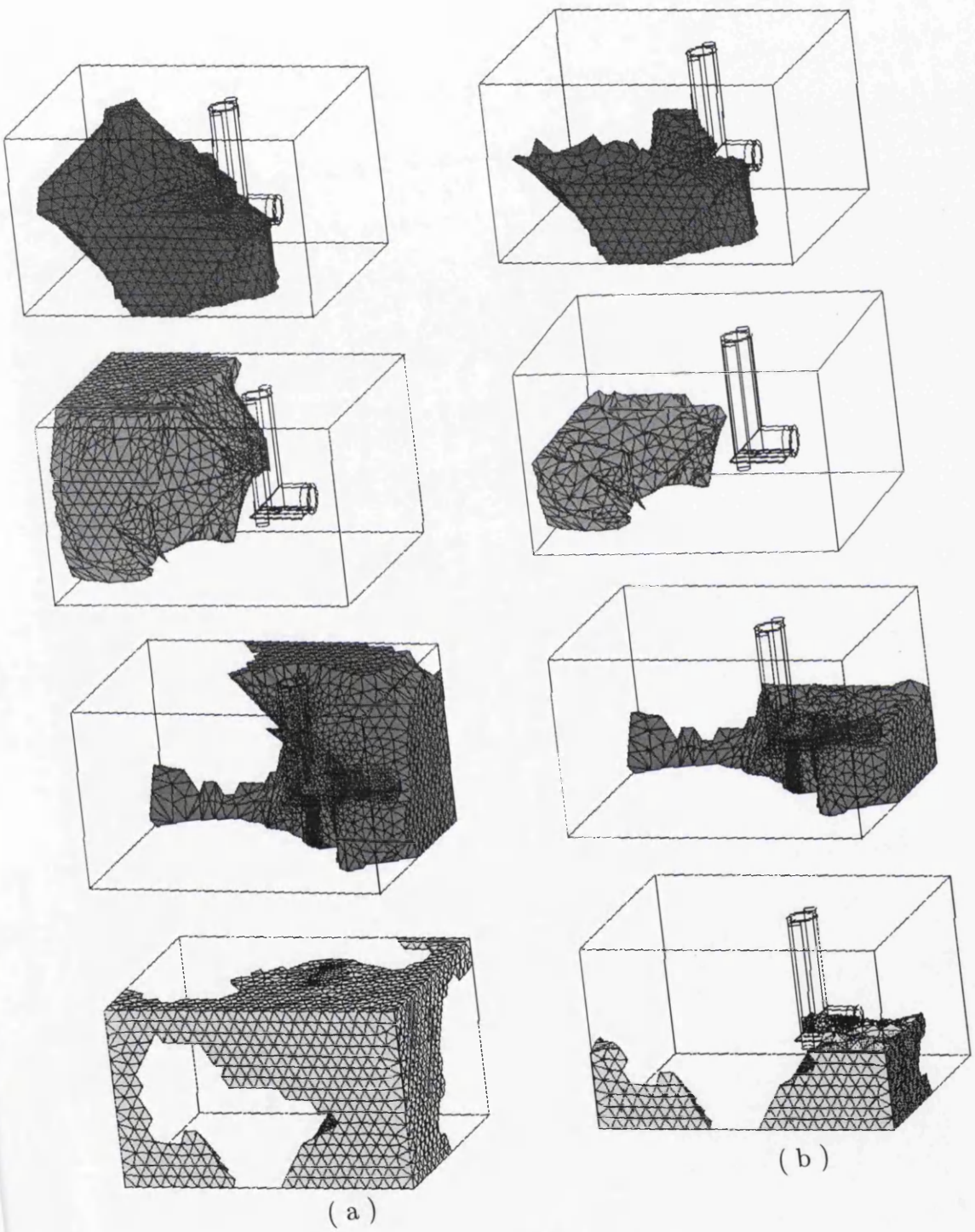


Figure 3.14: Surface and volume grids for the independent sub-domains: (a) The closed and oriented surface grid of the sub-domains, (b) A planner cut in the volume grids.

3.3 The Direct Decomposition Method

Despite of the success in using the indirect decomposition method to generate a considerable number of grids for complex configurations, the method in general has got a number of problems:

- The overall shape of sub-domain boundaries is usually very complicated, which makes maintaining the boundary integrity after Delaunay triangulation rather difficult and time consuming.
- The demand for generating an initial tetrahedral grid can become a real bottle neck in the process. Scalability and efficiency of the algorithm overall can get badly effected by such an inevitable sequential step.
- Achieving an acceptable level of workload balance among sub-domains at the domain decomposition step has proved to be very difficult if not impossible.
- The inter-domain communication cost is very likely to be high as a result of the extreme shape of the sub-domain boundary.

In addition to the issues listed above it is appropriate to mention another practical problem that has been encountered. Though it is not caused directly by the indirect decomposition method itself, it can have a massive impact on the efficiency of the entire procedure. It is related to the fact that some Delaunay grid generators may allow the point insertion technique to refine elements that lie completely outside the domain boundary[4]. An illustrated sketch of such a case is presented in Figure 3.15, where the initial triangulation constructed on a domain boundary and its convex hull points is shown (*b*); elements that are referred to in the above are presented in dotted line format in (*c*).

In general, the indirect decomposition method has failed to demonstrate a steady and robust performance in gridding complex geometries, as well as to be a scalable and an efficient method for generating tetrahedral grids in parallel. Thus, almost about the end of the second year of this study, it became clear that more revolutionary changes are required, and it was the time to search for an alternative method. A number of points were considered as the key issues while investigating new method such as: improving the quality of the internal boundary grid, maintaining the overall scalability by avoiding any unscalable procedure and enhancing the efficiency in general by improving the workload distribution among the sub-domains.

[4]This happens only *during* the gridding procedure itself, since all 'external' elements will disappear after recovering the boundary grid.

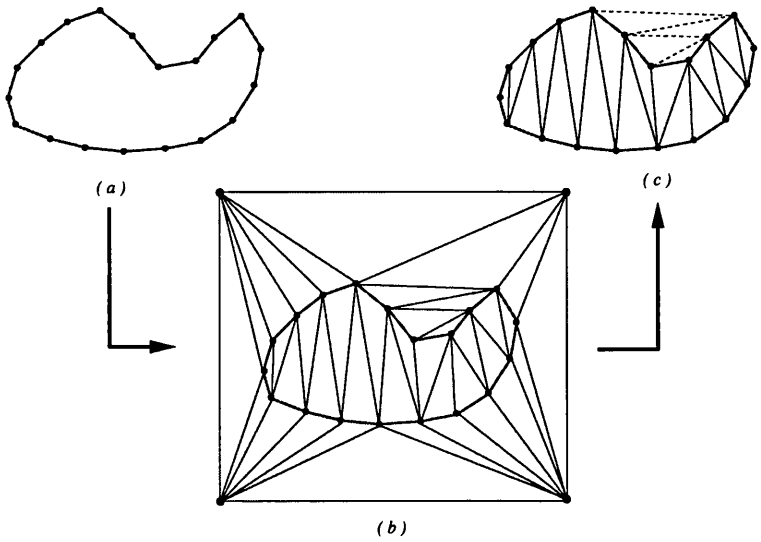


Figure 3.15: An illustration sketch of a case where some 'external' elements can be refined during the Delaunay grid generation procedure.

3.3.1 An Overview of the General Algorithm

The general algorithm in this method, as in the previous method, starts also from the discretized boundary of the domain (see (*b*) in Figure 3.16), whilst the enclosed area/ volume in here is subdivided by acting *directly* on the boundary grid. It is carried out by a set of planar cuts that can be imposed in various ways such as: a set of parallel planes distributed along one axis, a Cartesian network of planes or by employing an octree decomposition procedure. An example of a typical case based on an octree decomposition approach is presented in (*c*) in Figure 3.16. Clearly, the inter-domain boundary in the two dimensional space would always consist of a set of straight lines. This simply means that the same technique used in the indirect decomposition method for discretizing the internal boundary can still be applied in this method, see (*d*) in Figure 3.16. However, the story is totally different in the three dimensions, despite of having some of the techniques used in both methods there are some variations in the overall procedure; which will be discussed in details shortly in Section 3.3.3.

Having completed the domain decomposition and the discretization of the inter-domain boundaries, the closed boundary of each sub-domain can then be constructed and the internal triangular (tetrahedral) grids generated exactly the same way as in the previous method. See Figure 3.17 for an illustration of a set of sub-domain boundary grids in (*a*), and the final grids in (*b*). In general, this method produces a very different shape of sub-domains to the ones produced in the previous method, particularly in three dimensions. The

overall shape is more regular and surface grids on the internal boundary are much smoother and have far better quality elements. In fact, such differences have introduced massive improvements into various aspects of the parallel grid generation procedure.

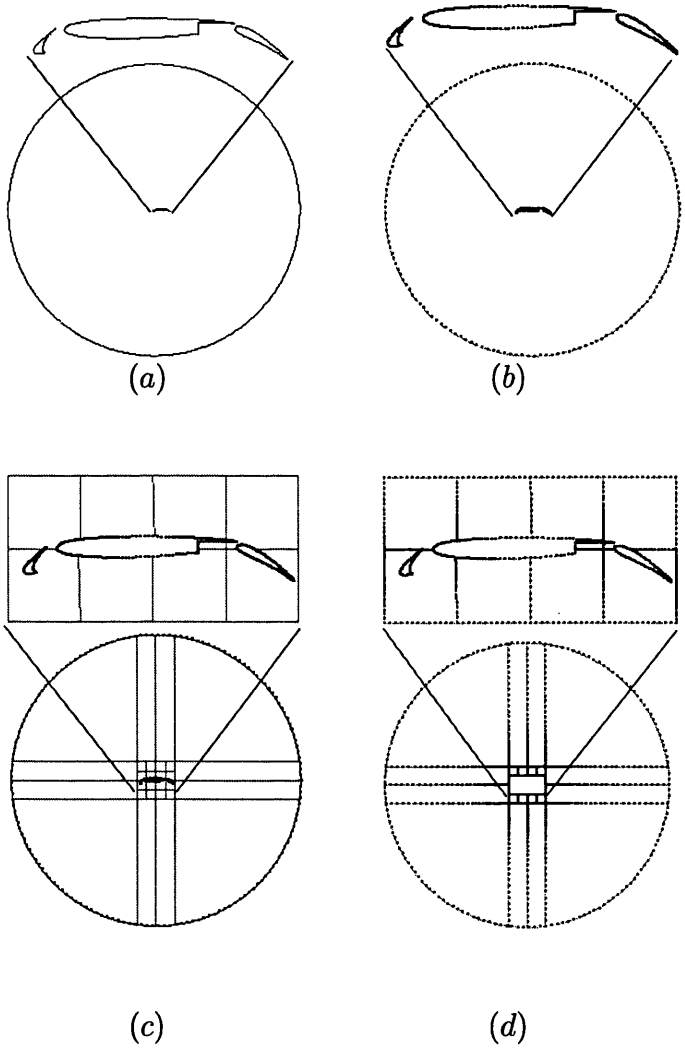
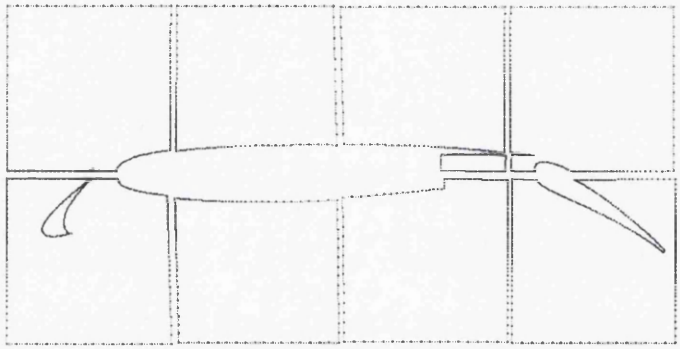
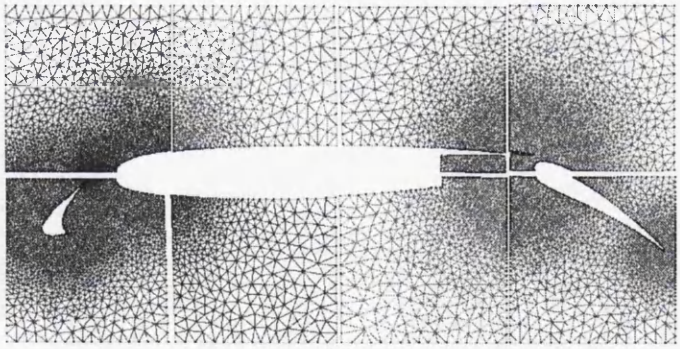


Figure 3.16: Illustration of the major steps in the direct decomposition method with multi-elements airfoil configuration: (a) Geometrical definition of the configuration, (b) Discretized boundary as a set of points and edges, (c) Domain decomposition Cartesian or Quad-tree, (d) Internal boundary gridded with the sources effect.



(a)



(b)

Figure 3.17: Illustration of the major steps in the direct decomposition method with multi-elements airfoil configuration: (a) Closed boundary of individual sub-domains, (b) individual sub-domains grid.

3.3.2 Domain Decomposition

As mentioned above, there are different ways for implementing the planar cut technique in order to carry out the 'direct partitioning' of the computational domain. Our main interest at this stage of the study was first to prove the various concepts associated with the direct decomposition method and demonstrate its advantages, then all the further tuning and refining of the implementation procedure could follow in the due course. Therefore, it was appropriate to adopt a very basic option in developing the partitioning technique such as distributing the planar cuts along one of the main axis in the domain. More advanced domain partitioning techniques, based on the concept of multi-directional planar cuts or even more sophisticated algorithms such as RSB [112] and MRSB [6], are also applicable and may produce better results. However, broadly speaking, the *best* domain decomposition algorithm should consider the influence of the background grid and other point spacing parameters more effectively, such that a well balanced workload distribution can be achieved.

Different criteria for choosing the cutting plane positions along one axis have been made available in the program. Currently, there are three different options: equal distance for each interval, equal number of boundary elements between every two planes and interactively (where the exact location of each cut is defined explicitly by the user in advance). A validation procedure is implemented in order to check if any cutting plane is too close to any parts of original boundary. If the normal distance between a cutting plane and the nearest point on the original boundary is less than the *local* minimum point spacing, a location re-adjustment procedure is carried out. Such a procedure would incrementally move the position of the cutting plane from its original problematic position until a new valid location is found. However, after all the cutting planes are finally located, every set of triangles that lie between two planes are assigned to be a sub-domain, and numbered incrementally starting from one end of the partitioning axis.

In order to improve the smoothness at the interface region, every triangle that is neighboured by two triangles from the adjacent domain is moved from its original domain to the neighbouring one. An illustration of such a general smoothing technique is presented in Figure 3.18. In fact, the same 'concept' has been employed at other places in this study, such as at smoothing the internal boundary in the indirect decomposition method, see Figure 3.4 in page 46, and at the load redistribution procedure, presented in Chapter 5. However, the real benefit from this procedure is in improving the smoothness of the internal boundary, as well as reducing the inter-domain communication cost.

Pseudo code for the Direct domain decomposition algorithm

```
(1) Construct the element-neighbours tree of the original
    boundary surface grid (Ne_tree);

(2) For every boundary point, establish a list of all
    connected triangles (Np_conn);

(3) Project the boundary nodes on the main axis of the
    planar cuts and then sort them accordingly (Np_sort);

(4) Based on the user choice of the decomposition criterion
    and the total number of sub-domains (Nsub_tot),
    establish the position of (Nsub_tot - 1) planes;

(5) Check if all plane positions are valid, then carry out
    the location adjustment procedure if it is needed;

(6) Using both lists (Np_conn and Np_sort), and starting
    from one end of the axis give every triangle on the
    original boundary grid the right sub-domain number;

(7) Carry out the smoothing procedure on triangles at the
    interface regions;
```

Recall that one of the key advantages of the direct decomposition method is the improvements in the scalability of the overall algorithm in general and the domain partitioning procedure in particular. Thus, it is appropriate to demonstrate this by inspecting the behavior of the algorithm in respect to a change in the boundary surface grid size, see Figures 3.19, and in the total number of sub-domains, see Figure 3.20. It is clear from both figures that the time needed to complete the partitioning procedure is a linear function of the number of elements /sub-domains. In fact, the time presented includes, in addition to the time spent on the decomposition procedure itself, the time needed to derive the 'rim' associated with every cutting plane [5]. However, such a linear relationship confirms that the direct decomposition method in general should always be scalable, at least at the domain partitioning step.

[5]More details about these rims and their role in discretizing the internal boundary are to follow in the next section.

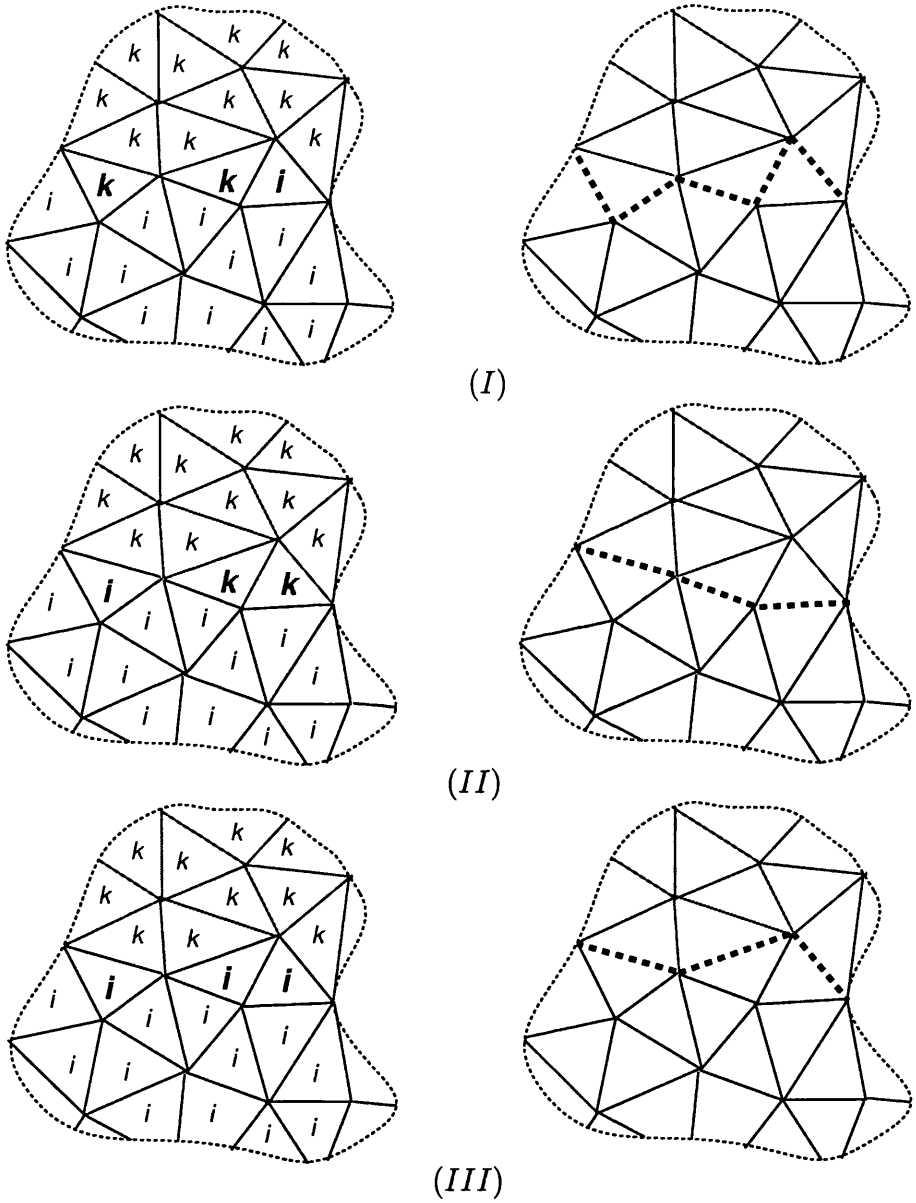


Figure 3.18: An illustration of a general smoothing procedure, which is integrated into various sub-algorithms in this research. The original configuration of a set of triangles is presented in (I), which can similarly be a set of tetrahedra. Elements considered to be the interface region between sub-domain `i` and `k` are presented in bold. The internal boundary associated with each configuration is presented in the dashed line on the right hand side. Two different possibilities for employing the smoothing procedure are presented in (II) and (III), where the difference is due only to the order of adjacent sub-domains.

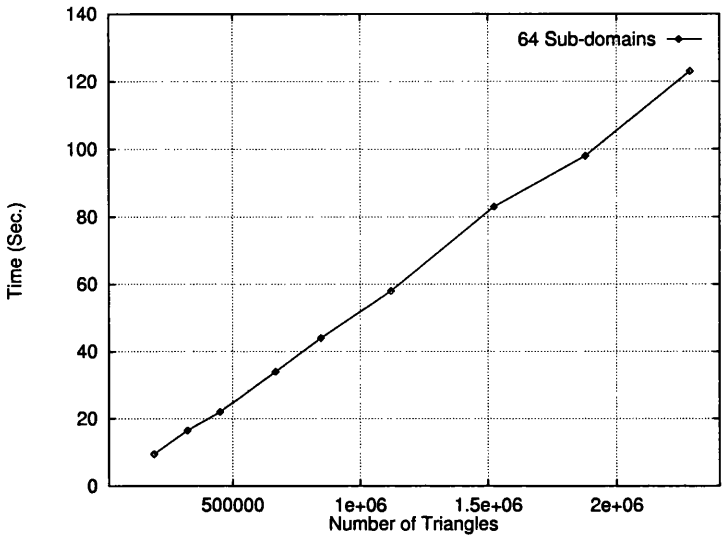


Figure 3.19: Time required for the domain decomposition, 64 Sub-domains where surface gird on the original boundary varies between 0.2 and 0.3 million triangles.

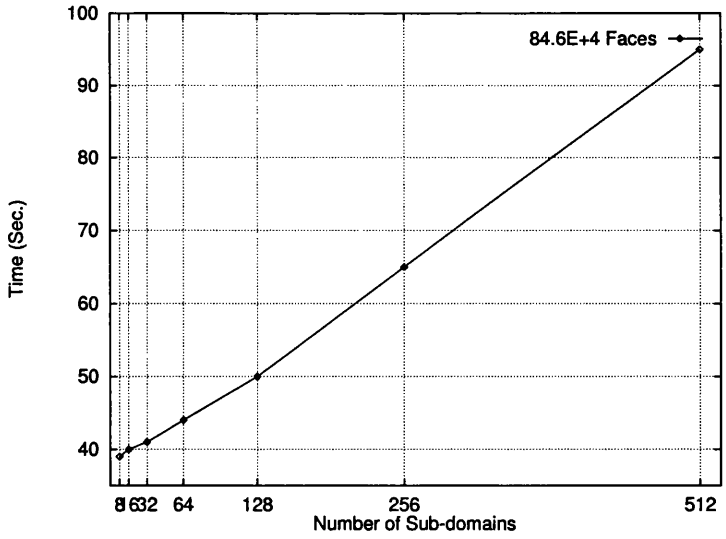


Figure 3.20: Time required for the domain decomposition, 845948 triangles on the original boundary with the number of sub-domains varying in the range $[8 - 512]$.

3.3.3 Discretizing the Internal Boundary

Obviously, internal boundaries in the direct decomposition method are associated with the planar cuts as they are used in partitioning the domain; the question now is how to create a smooth high quality triangular grid on such an 'empty plane'. A multi-step procedure has been developed for this purpose, which starts by deriving all edges shared between the two adjacent sub-domains at every cutting plane. Such a set of edges forms a closed contour(s) of 3D edges so called the 'rim'; which is then mapped into the cutting plane using an *enhanced* form of orthogonal projection scheme. A coarse 2D triangular grid is constructed on the cutting plane, which in turn is mapped back to the three dimensional space where the final internal boundary surface grid is constructed. The same point insertion and edge swapping technique employed in the indirect decomposition method, see Figure 3.10, is used at this last step.

The construction of 'valid' rims on the projection planes

Whilst extracting the rim edges from the original boundary surface grid is a rather easy task, mapping them onto the cutting plane can be a bit more subtle. It has been observed that a straightforward mapping using an orthogonal projection of the rim edges directly onto the cutting plane may cause some problems. An illustrated example is presented in Figure 3.21; where clearly though the two edges (i.e. 1-3 and 2-4) do indeed appear in the projected rim they simply do not exist in the 3D original rim!. Therefore, a 'correction procedure' has been developed and integrated into the projection procedure such that only *valid* rims are constructed on the projection planes.

The correction procedure is based on a few simple concepts. First of all, as the rim edges are extracted from the surface grid in *random* order, a re-ordering scheme is implemented such that every rim consists of *continuous* contour(s) of edges [6]. A point is chosen somewhere in the rim and projected onto the cutting plane, then, moving in one direction, edges are mapped incrementally within 'steps'. Every step may include a different number of edges but the procedure is always the same. Such a procedure starts by evaluating the 'weight' of each edge included in the step, which is based on the ratio of the individual edge lengths to the total accumulative length of all other edges in the step. Then, just after the second point of the last edge in the step is projected directly onto the cutting plane, all other points (edges) in the step are mapped into the same plane using linear interpolation between the two projected points considering the weight associated with each edge. Now, if we revisit the illustrated example presented in Figure 3.21, and consider the edges 1-2, 2-3, and 3-4 as in one step; according to the correction procedure presented above: point 1 will be

[6] In order to maintain an efficient search scheme at this procedure, the elements neighbouring tree and the connectivity matrix of the surface grid are exploited.

projected first, followed by point 4, and then points 2 and 3 will be mapped using a linear interpolation scheme based on the weight associated with each edge.

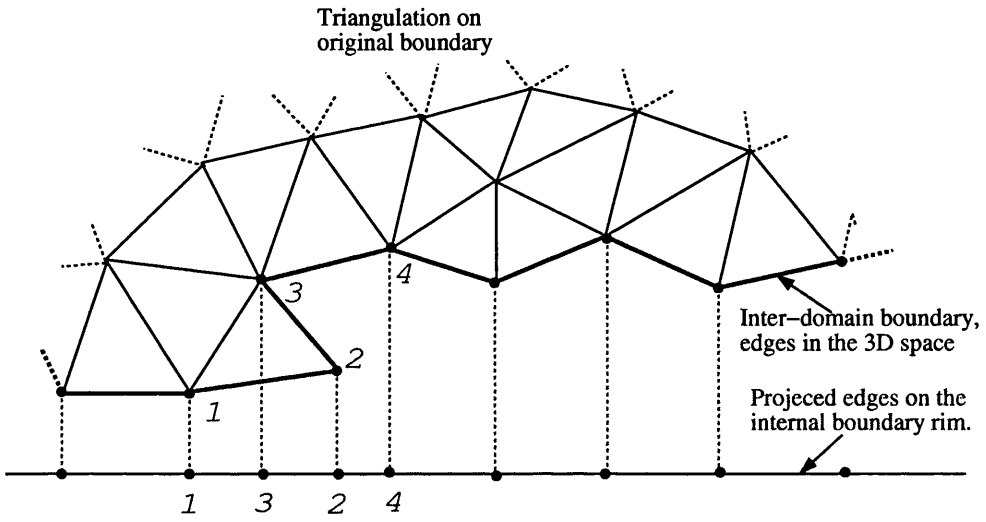


Figure 3.21: The process of mapping the rim edges from the original boundary surface grid into the cutting plane, in order to construct the 'basic' triangulation of the internal boundary in the two dimensional space.

Clearly, some rims may have more than one closed contour of edges. Hence, every time the mapping of a closed contour is complete, a checking procedure must be carried out in order to determine if any edge is still left in the three-dimensional space or not. In the case of finding such an edge, the same procedure as described above will be repeated again and so on. A flowchart summarising all steps involved in the procedure is presented in Figure 3.22.

The 'accuracy' in representing the rim on the projection plane using such a procedure is indeed questionable. It seems that there is a great potential for introducing dramatic changes into the shape of the rim or even loosing some of its important features completely!. The consequences of such issues can also be very serious and it may destroy the reliability of the direct decomposition method. The key issue lies with the mechanism for deciding how many edges are to be included in the step (i.e. when to stop collecting edges).

First of all, the maximum number of edges to be collected in one step is limited up to 5, and it can be as small as 1. Secondly, other geometrical constraints such as the angle between normals on the two boundary faces (i.e. the ones that have the edge that shares the relevant point in the rim) are always considered. In other words, the mapping procedure would maintain the 'curvature'

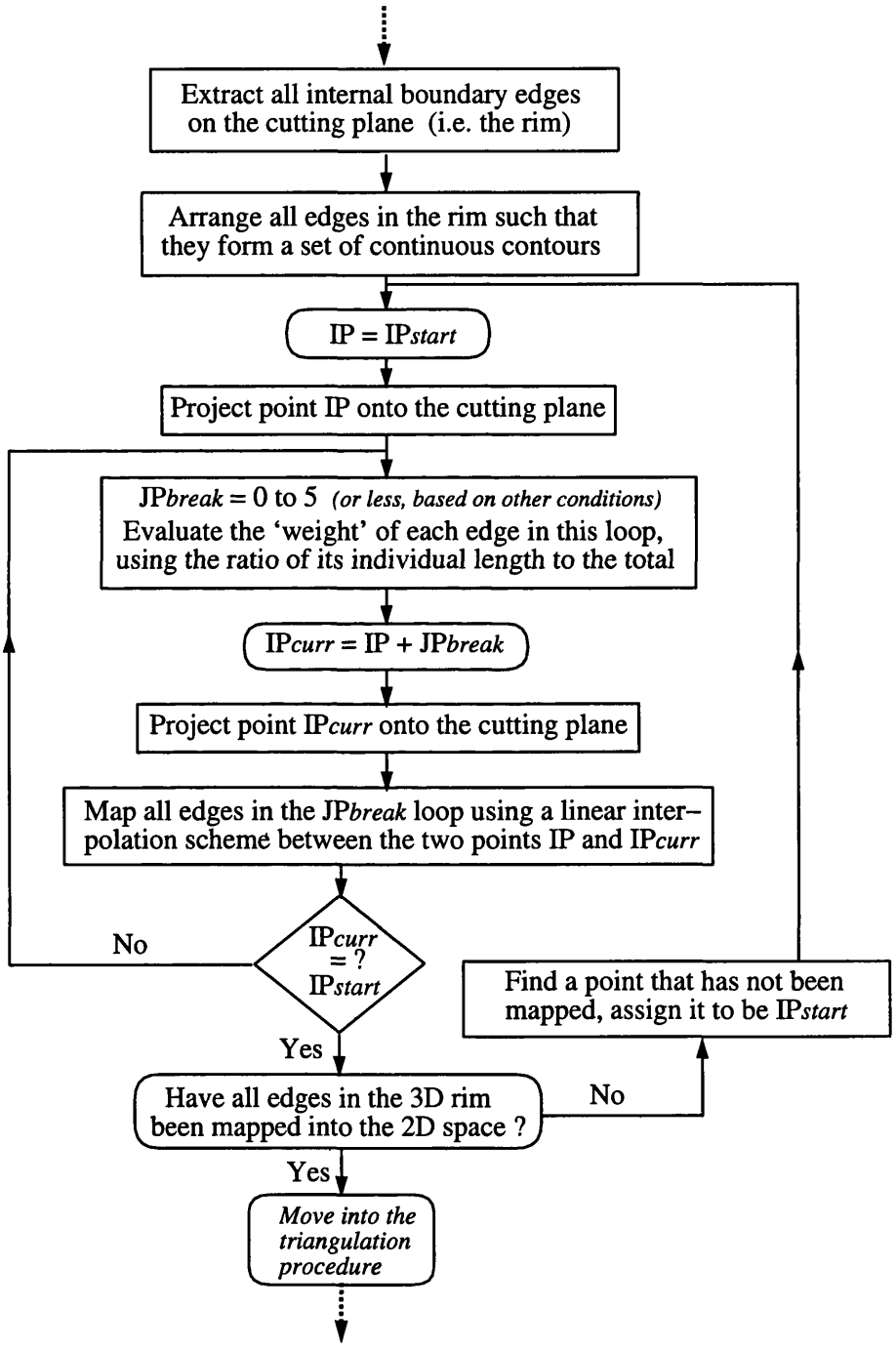


Figure 3.22: Overview of the procedure used in mapping the rim edges from the 3D space onto the cutting plane.

of the rim such that every noticeable change in the surface grid normals is well represented. A very good example that demonstrates the effectiveness of the mapping procedure is presented in Figure 3.23. The result of direct orthogonal projection is presented on the left hand side, whilst on the right hand side is the result after introducing the correction procedure. Notice how all geometrical features are reserved and simultaneously all problematic edges have been avoided (see where the two arrows are pointing in the last row images).

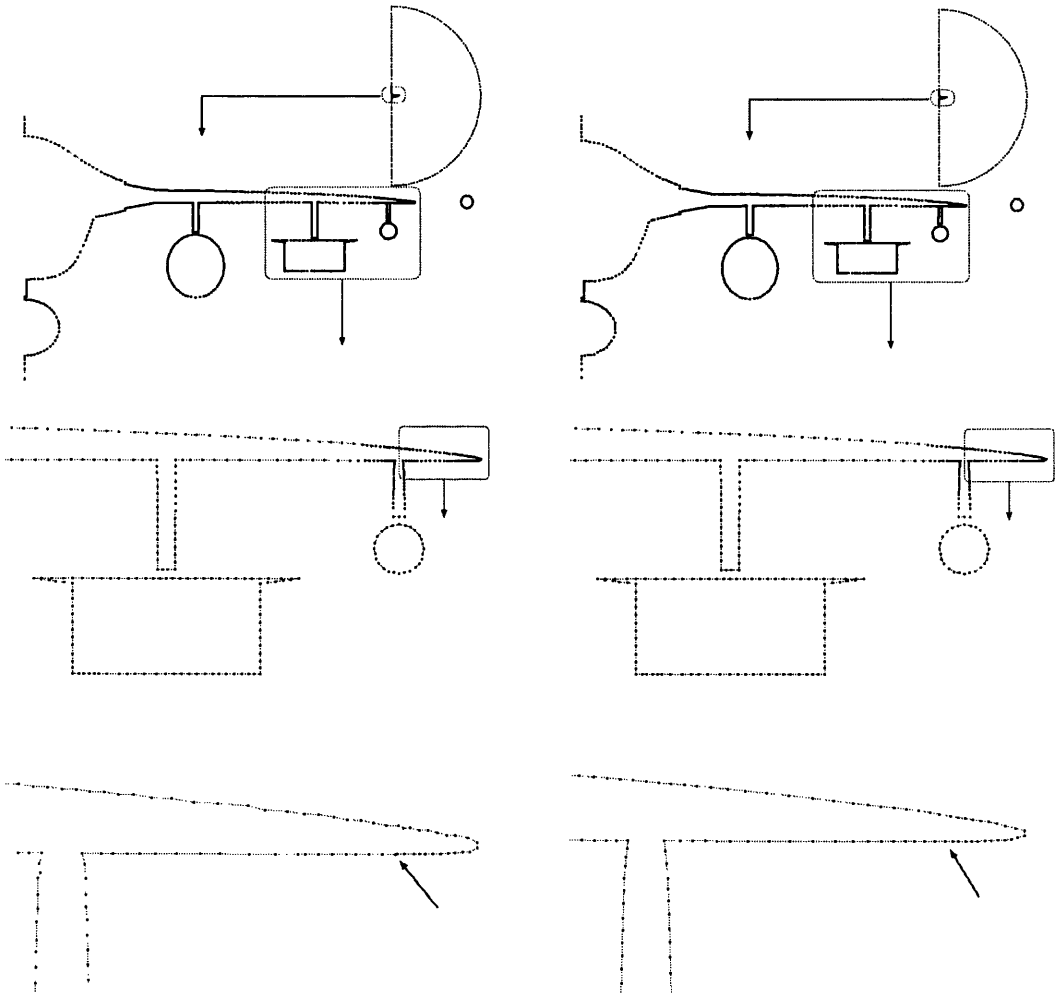


Figure 3.23: Two different cases for the projection of internal boundary edges (i.e. rim): using a direct orthogonal projection (left), and after integrating the 'correction procedure' (right). Notice the very limited impact of such a procedure on the general shape whilst a problem (as indicated by the arrows) is resolved.

The construction of a 'basic' grid on the planar cross section

It has been observed that completing the construction of the internal boundary grid on the planar surface in the two-dimensional space and then mapping it back into the three-dimensional space often produces a number of badly distorted elements, particularly in the region close to the rim. Furthermore, intersections between some elements on the internal boundary grid and the original boundary also could be found. Alternatively, it has been found that generating a relatively coarse grid in the two-dimensional space and then using it as a 'basic' grid for generating the final fine grid in the three-dimensional space can be more reliable. The same point insertion and edge swapping technique used in the indirect decomposition method, see Section 3.2.3, can be applied to the basic grid after it is been mapped back into the three-dimensional space.

In fact, the 'basic' grid could be built as just a Delaunay triangulation of the rim points, i.e. without inserting any new point at all as shown in Figure 3.24. However, introducing a limited number of points in the two-dimensional space, see Figure 3.25 has proved to be very effective in maintaining the plane-like shape of the internal boundary and to improve the quality of the final grid. This would subsequently increase the robustness and the reliability of this method substantially.

A number of enhancement techniques have been introduced in order to eliminate the possibility of having intersections between the internal and original boundaries; particularly at the high risk areas such as displayed in the closeup in the Figures 3.24 and 3.25. Clearly, an edge connecting two neighbouring points on the rim in the planar cross section surface grid might be simply a duplication of an already existing edge on the original boundary surface grid. Hence, an *extra* point on such an edge is interpolated and two new triangles replace the original one with the problematic edge (see the circulated elements on the right hand side in Figure 3.25, and compare them with same on the left hand side). Furthermore, an edge swapping technique is also implemented with a swapping criterion that tries to reduce the number of edges near the rim (in other words, maximizing the edge length by swapping); see Figure 3.26.

Generating the internal boundary surface grids using such a multi-step procedure has proved to be very effective. High quality and smooth triangular grids that have a good consistency with the original boundary grid point spacing are always guaranteed. A typical example of a final grid is presented in Figure 3.28, where the 'basic' grid (i.e. after been mapped back into the 3D) is also presented for comparison. Most importantly, the procedure is still very efficient and it is indeed a scalable algorithm, see Figure 3.27.

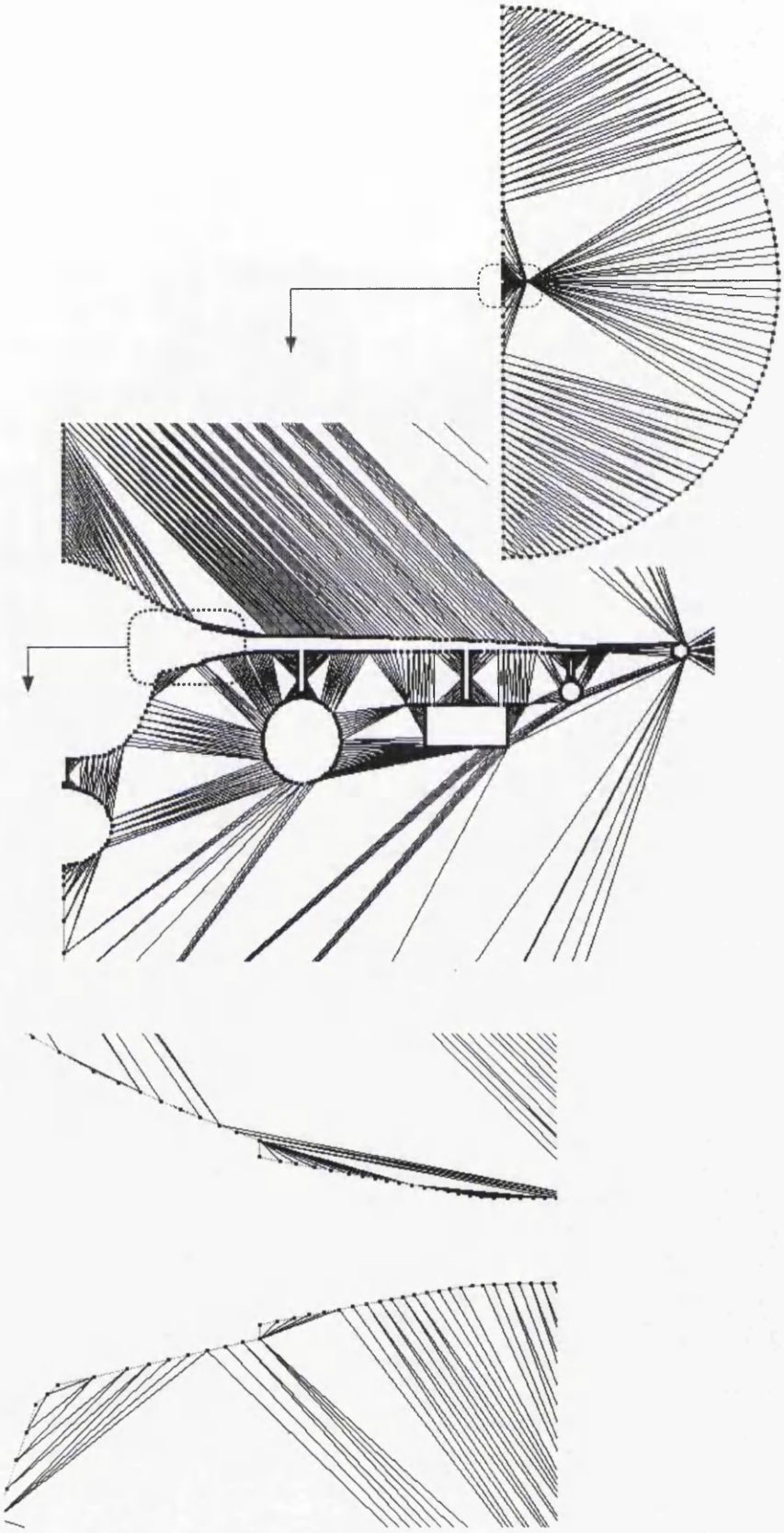


Figure 3.24: Triangulating the boundary (rim) points without introducing any new point on the planar surface.

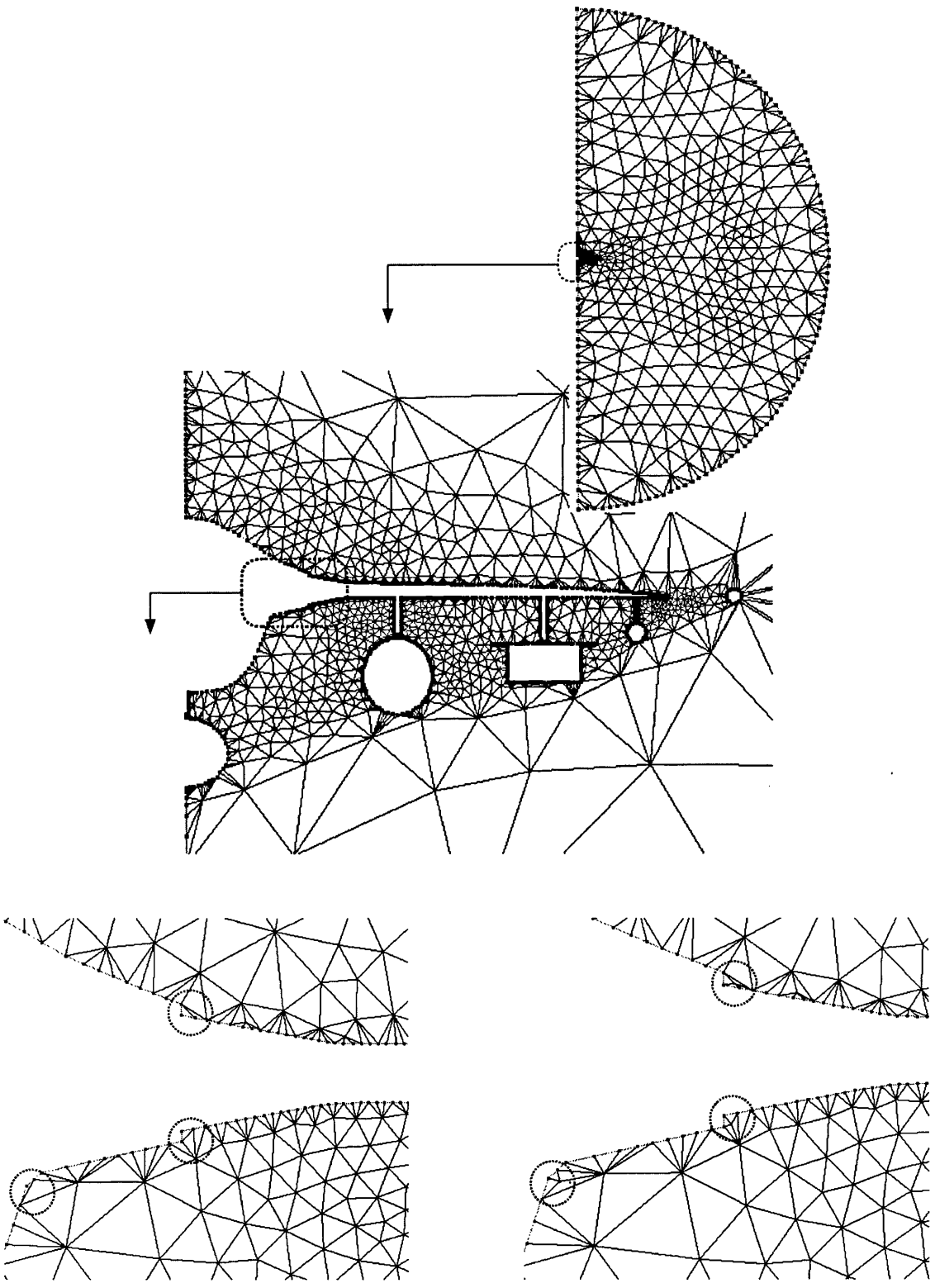


Figure 3.25: Triangulating the rim points in the planar surface with the option of introducing *some* points. Notice that edges connect two points on the rim (see circles in the LHS close-up) may also exist on the original boundary surface grid. Such a problematic edge can be eliminated employing a procedure that uses point interpolation (see circles in the RHS close-up) and edge swapping techniques (see Figure 3.26).

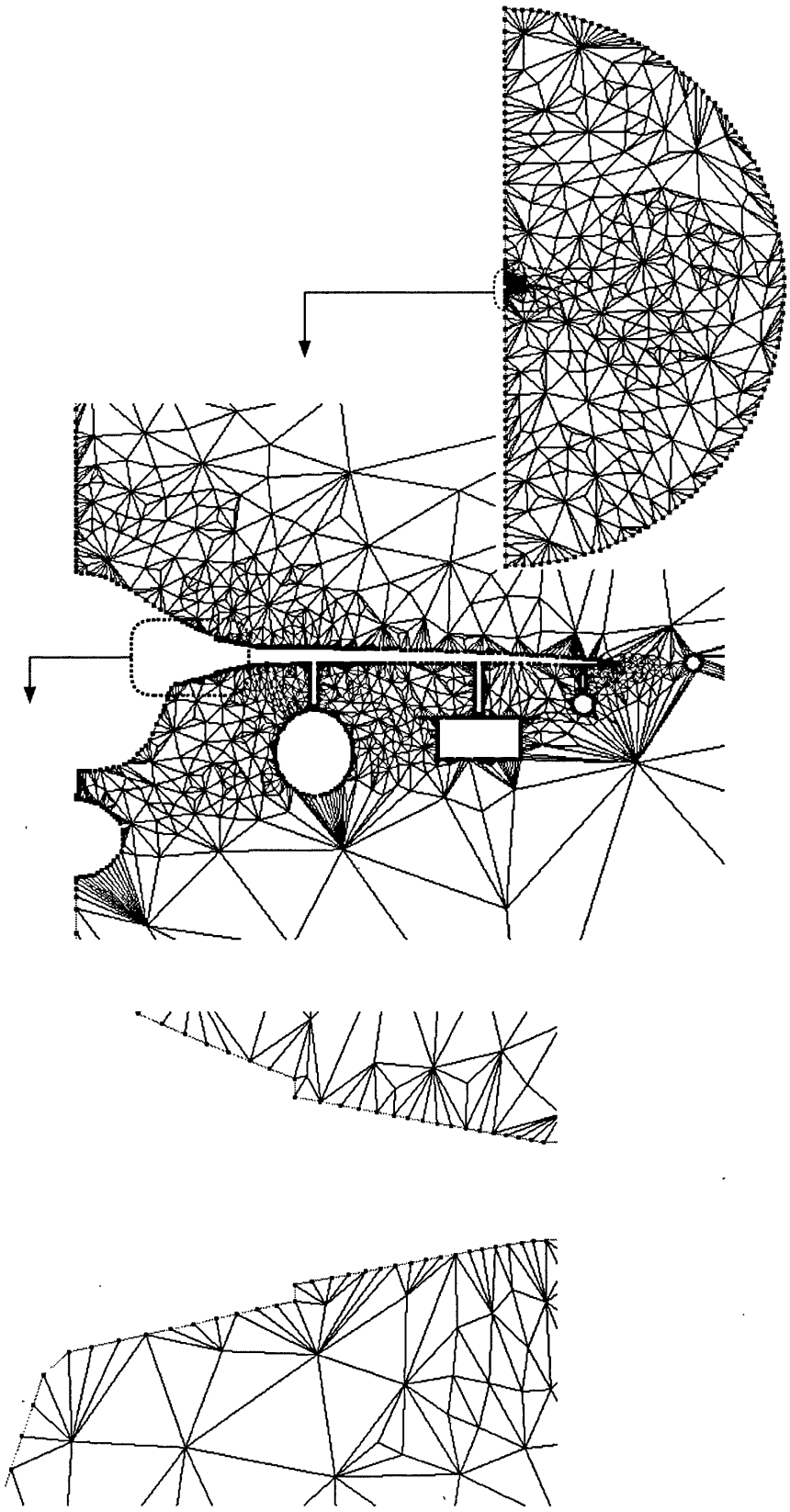


Figure 3.26: Apply the edge swapping technique in order to reduce the possibility of intersection with the original boundary grid.

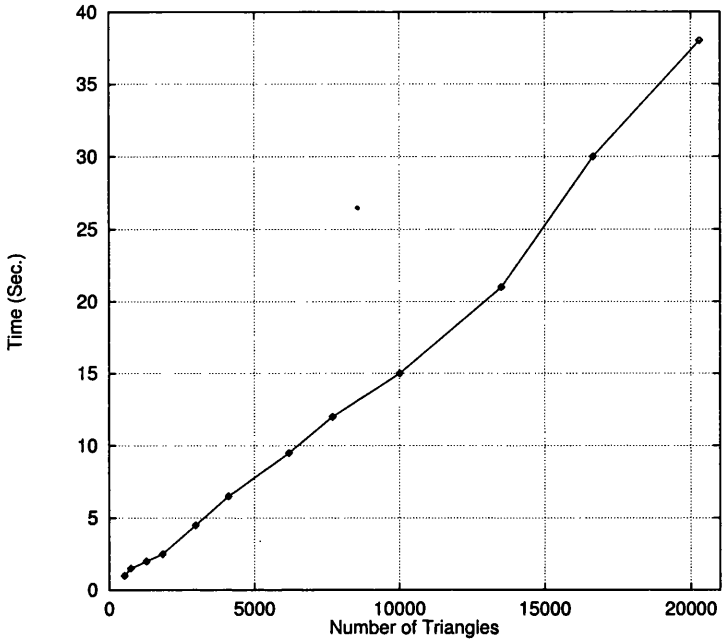


Figure 3.27: Total time required to generate a surface grid on an internal boundary. Notice the linear relationship with the total number of elements.

A flowchart of the general algorithm with an illustrating example

As in the indirect decomposition method case, we close the discussion by presenting a flowchart summarising all the steps involved in the general algorithm, see Figure 3.29. The boundary of individual sub-domains are constructed and then gridded exactly as in the indirect decomposition method, thus there is no need to repeat the same discussion here. However, the direct decomposition method is illustrated visually by presenting one complete example. A relatively small size tetrahedral grid of a configuration of a civilian aircraft (i.e. B60 model) inside a box, generated within 4 sub-domains using the algorithm discussed above, is presented over three Figures. First of all, an overview of the geometry is presented in (I) in Figure 3.30, whilst some snapshots of the original boundary surface grid before and after the domain partitioning are presented in (II) and (III) respectively. The construction of the surface grid on the three internal boundaries is then illustrated in Figure 3.31, where a general view of one boundary is displayed (at the RHS) with a snapshot of a close up from each internal boundary. Snapshots of the rim on the internal boundaries, after they have been mapped into the two dimensional space, are presented in (I) in Figure 3.31; notice that the first three images from the LHS are closeups of the cutting planes 1,2 and 3 respectively and the fourth image is an overview of the first plane. Following the same order as adopted in (I) in Figure 3.31, snapshots of different steps involved in the gridding procedure

are presented in (II) and (III). Details from each sub-domain surface grid are presented in column (I) in Figure 3.32, starting from sub-domain number 1 in the first row and so on. A planar cut in the volume grid inside each sub-domain is presented in column (II) of the same figure, whilst a close up of the same cut is presented in the last column; see Figure 3.32 page 79.

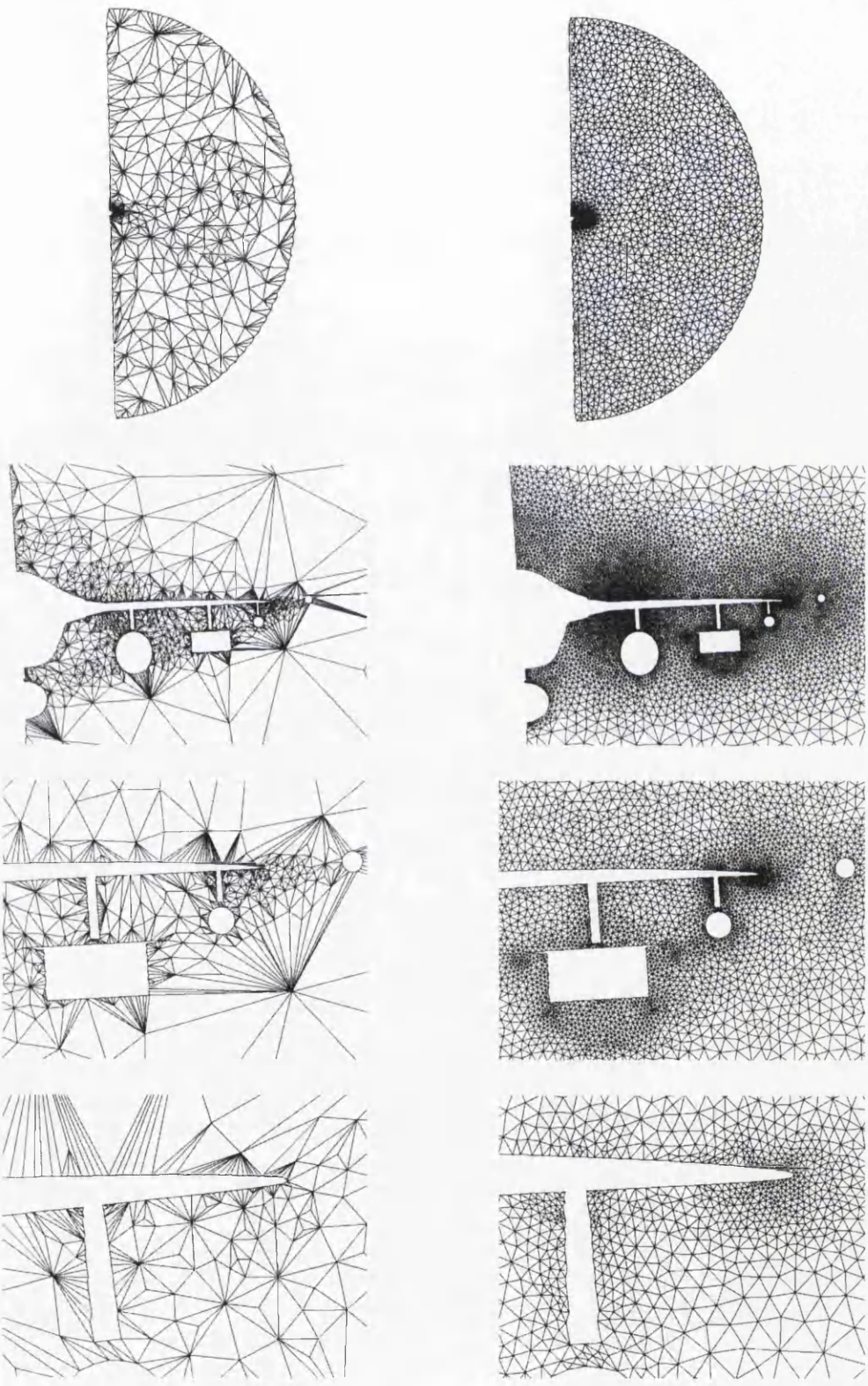


Figure 3.28: Different snapshots of an internal boundary grid: the grid after the mapping into the 3D space (left), and the final fine smooth grid (right).

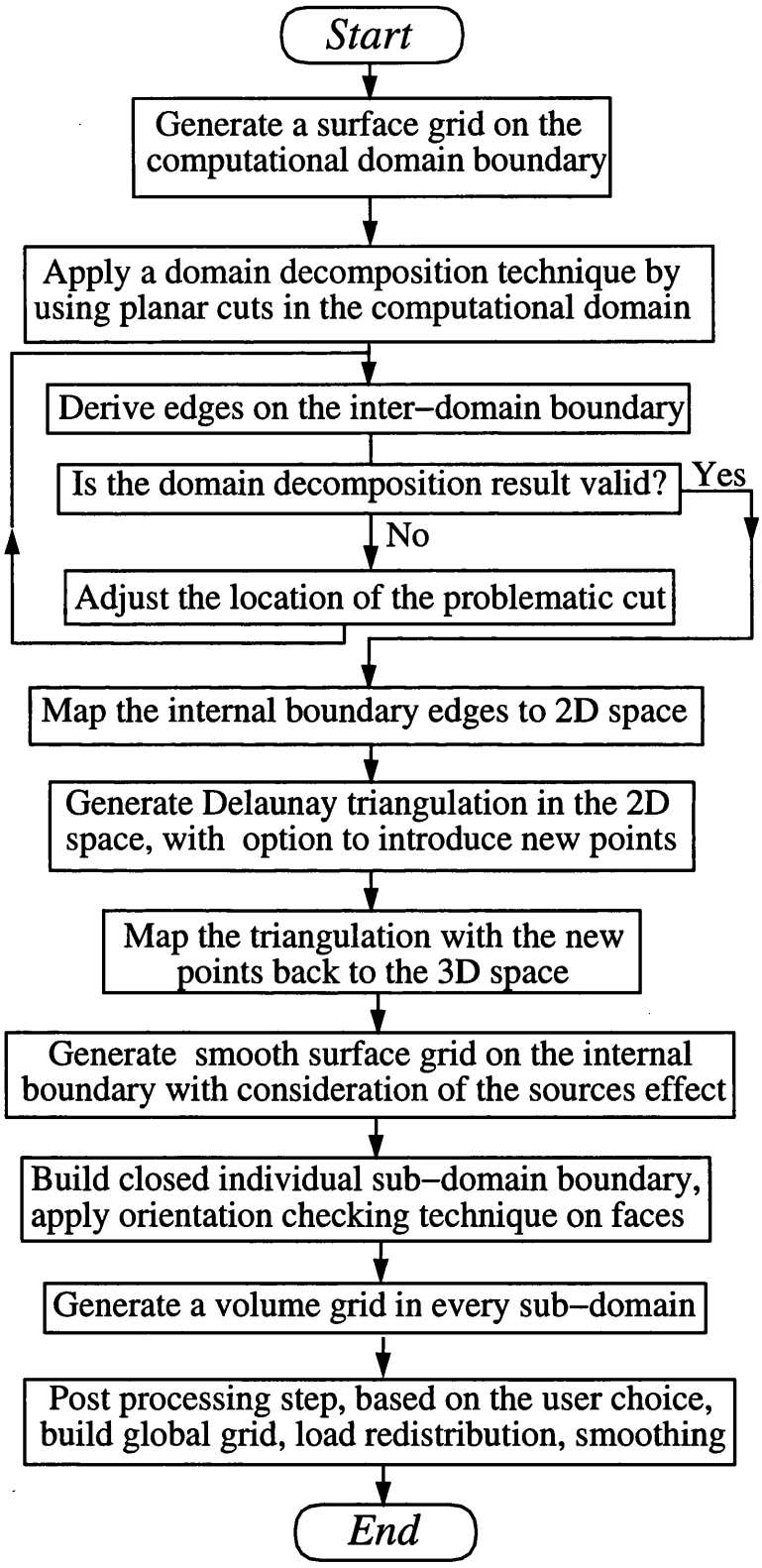
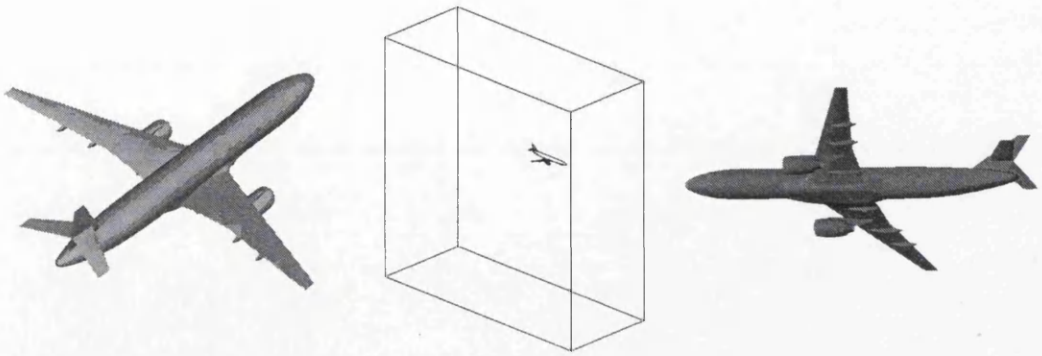
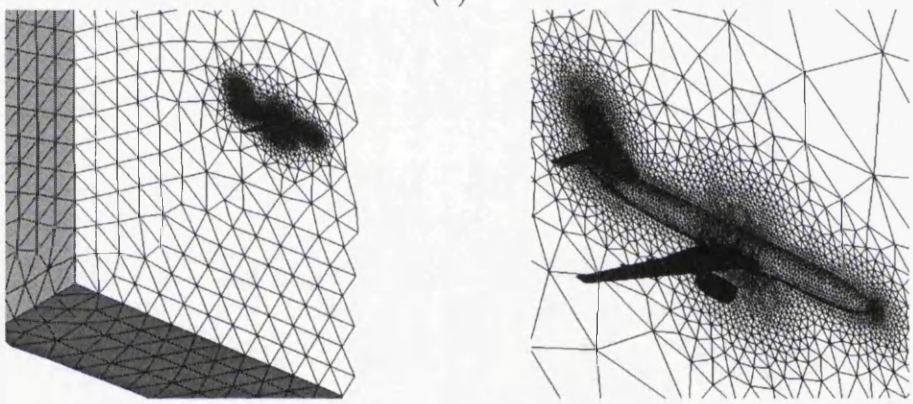


Figure 3.29: Overview of the general algorithm using the Direct Decomposition Method.



(I)



(II)

(III)

Figure 3.30: An example of generating a volume grid of a civilian aircraft configuration inside a box, using the direct decomposition Method. (I) Geometry definition, (II) Triangular grid on original boundary. (III) Four sub-domains obtained using uni-directional planar cuts.

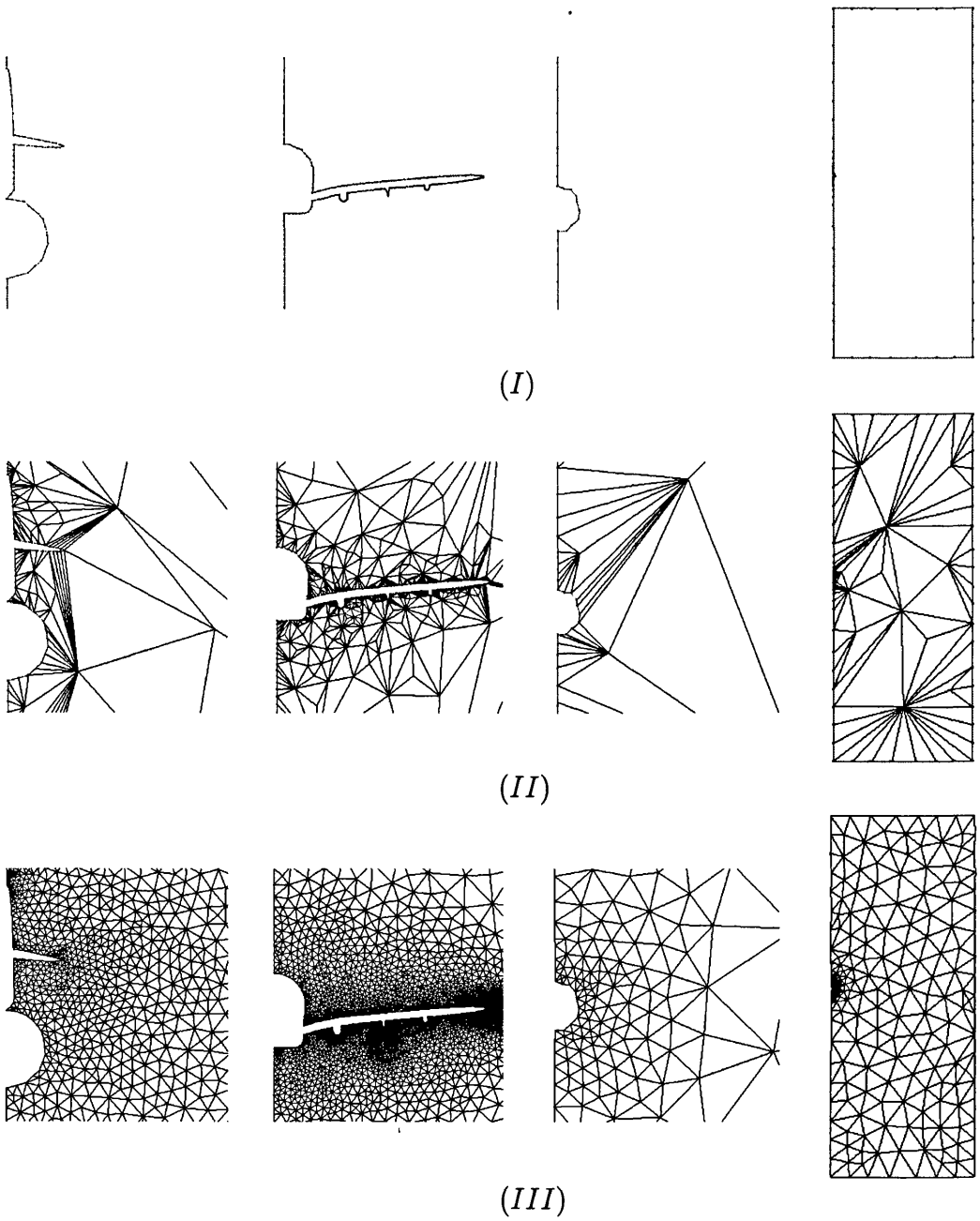
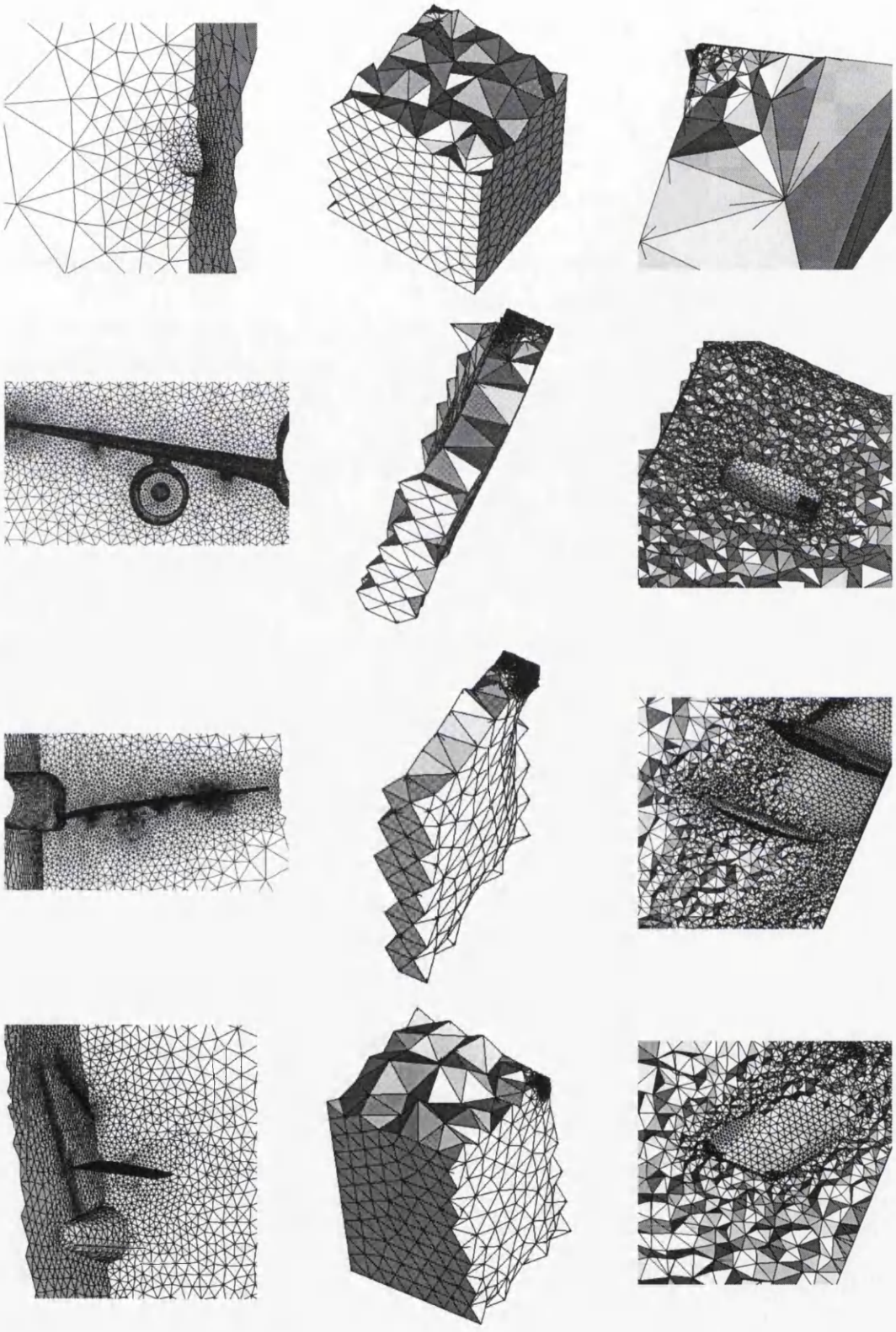


Figure 3.31: Generating high quality triangular grids on the internal boundary in the direct decomposition method. (I) Edges on the 'rim' in 2D space. (II) Delaunay triangulation of internal boundary in the 2D space, with the option of introducing points. (III) A complete smooth 3D triangular grid generated in the 3D space.



(I) (II) (III)

Figure 3.32: Volume grid in the four sub-domains, generated using a sequential Delaunay algorithm. (I) Details of the closed sub-domain triangular grids. (II) Planar cuts inside the volume grid. (III) Zoom in from (II).

3.4 Concluding Remarks

Two different methods (namely the indirect decomposition method and the direct decomposition method) based on the geometrical partitioning approach and belonging to the 'pre-gridding of the internal boundary' category[7] have been discussed in depth in this Chapter. A layout of the general procedure in each method has been presented using initially a simple 2D grid, followed by a detailed description of every technique associated with individual steps in the general algorithm. A flow-chart summarising each method and an illustrated example of a tetrahedral grid have been presented at the end of each section.

Issues found to be associated with the first method (i.e. indirect decomposition) have been identified and discussed before introducing the second method (i.e. direct decomposition) as an 'alternative'; see the list presented in page 57. The chapter has concluded: "the indirect decomposition method has failed to demonstrate a steady and robust performance in gridding complex geometries, neither has it proved to be a scalable and efficient method for generating tetrahedral grids in parallel". The major points considered while developing the direct decomposition method have been identified: improving the quality of the internal boundary grid, maintaining the overall scalability by avoiding any unscalable procedure and enhancing the efficiency in general by improving the workload distribution amongst the sub-domains.

Whilst, the quality of tetrahedral grids generated in parallel will be discussed in detail in Section 5.3, the issue has been addressed briefly in this chapter. The role of the sub-domain's general shape and the surface grids, particularly on the internal boundary, has been highlighted. The chapter has demonstrated that the direct decomposition method, unlike the indirect decomposition method, will always produce sub-domains of 'regular' shapes. Internal boundaries are always very similar, and generating a high quality smooth triangular grid on every internal boundary is guaranteed. The chapter presented the direct decomposition method as a reliable, scalable and efficient approach for generating tetrahedral grids in parallel.

[7] For more information about this category and its advantages see page 33.

Chapter 4

Parallel Implementation

4.1 Introduction

Whilst the previous chapter has focused on demonstrating different techniques involved in the partitioning and gridding operations, this chapter is devoted to discuss the integration of these techniques under one computational framework. The framework adopts the Message Passing Library (MPL) as a parallel programming model[1], and uses a unique 'template' for processing a 'task' in parallel. The chapter begins by reviewing in great detail the construction of the parallel processing template leading into a summary of its main features. A special technique termed as Dynamic Parallel Processing (DPP), which has been integrated into the template in order to enhance its flexibility, is discussed. The enhanced template, which will be referred to as a *DPP loop* throughout the rest of this thesis, represents the kernel of the overall parallelisation work in the developed framework.

This chapter demonstrates how the computational framework has been constructed by utilising one global algorithm, which represents both of the indirect and direct methods. Individual steps in this general algorithm are checked and a decision about 'to parallelise or not to parallelise' is made for each step. A generic form of the parallel framework is presented.

In order to illustrate the important role of the DPP loops in the computational framework, an example of a small size grid generated for a number of times using a different number of processors in each run will be discussed.

[1]For more information about programming models in general, see Section 2.3.1.

4.2 The Design of a Parallel Processing Template

Recall the discussion of 'why' the Message Passing Library (MPL) model have been adopted presented earlier in Sections 2.3.1 and 2.4. Issues associated with the construction of the template design such as the choice of the most appropriate parallel structure and the portability between machines of different architectures can be addressed. No previous knowledge of a message passing library is required to follow most of the discussion, however, familiarity with an MPI library[2] may become more important in later sections. Nevertheless, reviewing some of the material presented in Appendix B can be very beneficial for the interested reader.

Parallel structure

Recalling the algorithms presented in previous chapter shows that there is a number of *similar* jobs to be carried out in most of the main steps in the general algorithm (e.g. deriving and gridding M internal boundaries, constructing a closed boundary for N sub-domains, ...etc.). Clearly, each one of these steps can be accomplished by performing the same sub-algorithm on a set of distinguished data. Hence, it should not be hard to find out why the Single Program Multiple Data (SPMD) structure is the most suitable structure for the template.

The template implements the SPMD structure with a Manager/Workers mechanism. In such a structure, a processor acts as a *manager* and all other processors are *workers*. The *real* work is usually carried out in parallel by the Workers whilst the Manager carries out the synchronisation of job processing only. Specifying which processor is to be the Manager can be done in advance by the user, or at the beginning of the run by the program. A unique, and already known, 'ID' number is assigned to the Manager, also each Worker will have its own distinguished ID number. There must be a Manager and at least one Worker before the mechanism is valid. One physical processing unit can accommodate a number of Workers simultaneously, such a number depends on the message passing library and the computing platform (i.e. including the hardware and operating system). However, the terms Manager and Workers will be always used independently from the underlying computing platform.

Portability

The MPL model is known to be a highly portable parallel programming model, unfortunately that does not mean every parallel program that uses this model

[2]MPI in its own is just a library *specification* and the *real* library used in this research is an implementation called (MPICH).

will automatically be transferable among different types of parallel machines. It is still the programmers responsibility to take into consideration a number of factors such as the portability of the message passing library itself and, more importantly, the architecture of the targeted platforms[3]. For example, the construction of the template as presented so far (i.e. based on the MPL model and the SPMD structure) can still be developed in a way that the template operates on shared memory machines only. However, in short, our main interest is to ensure that the developed design can produce a machine independent parallel processing template/programs.

The template adopts a special procedure for processing a task (which consists of a set of similar jobs that are referred to as sub-tasks) in parallel. In this procedure, the Manager sorts the sub-tasks in descending order according to their estimated workload. And then, just before assigning each sub-task to a Worker, it extracts all the data required and initiates a new *local* numbering system. An appropriate link between such a system and the *global* data numbering system is established in case the data stored in the local system was to be integrated back into the global system. In fact, the two numbering systems above (i.e. local and global) and their interactions are discussed in depth in Section 5.2. However, it is enough here to report that the template utilises a 'data localisation' scheme, that ensures that every sub-task is defined to be a *self contained* and totally *independent* while it is been processed.

Clearly, the procedure above introduces some 'extra work' into the original algorithm, unfortunately, it seems that this is a price which has to be paid for having such a highly portable template. On the other hand, a parallel framework/ program that uses such a template would be able to operate on a single processor workstation, clusters of networked workstations, shared memory multiprocessor machines and massively parallel supercomputers. Furthermore, on the portability issues, a widely available and highly portable message passing library has been adopted. The MPICH library, which is an implementation of the MPI specifications, developed jointly by Argonne National Laboratory and Mississippi State University [53] has been adopted. However, for more information about the MPICH, and other topics related to message passing model, the reader is advised to consult appendix B.

Main features in the template design

As a summary of the various points discussed above, and before moving to discuss the DPP technique, it is appropriate to present a list of main features in the template design and an illustration of its generic form as in Figure 4.1.

[3]MIMD machines with shared memory or distributed memory architectures are the only types considered in here. See Section 2.3 for more information about types of parallel machines.

- A SPMD parallel structure is adopted with Manager/Workers mechanism. The Manager derives a set of *similar jobs* and administrates their processing in parallel by the Workers.
- A message passing library is needed for transferring data between the Manager and Workers. An MPI library has been used and it is strongly recommended, though switching to any other library is straightforward, see appendix B.
- There is no communication among the Workers themselves.
- Communication between the Manager and a Worker, whenever a task is assigned, takes place at the beginning and the end of the processing only.
- The administrative role of the Manager includes: sorting the *similar jobs* in descending order according to their estimated workload, performing the data localisation scheme on each job, assuring the independence of each job and finally synchronising the processing on the Workers.
- Parallel programs implement such a template would be able to operate on parallel machines with distributed and shared memory architecture and without any modifications whatsoever.

4.2.1 Dynamic Parallel Processing

In general, achieving a feasible level of speed up and efficiency in parallel programs might be very difficult. This is due to the following reasons:

1. The workload of the sub-tasks processed in parallel may vary dramatically from one to another.
2. It is possible to have very different levels of performance among the processors involved in one parallel run (for example, a network of heterogeneous workstations).

It is likely to have a set of processors waiting for the one with the largest workload, or for the slowest. To overcome this issue most of the parallel algorithms insert a *load balancing* technique, which, in general, diagnoses where the bottleneck is and subsequently redistributes the workload in order to achieve the same performance on all processors. Integrating such a technique within a parallel program to be used alongside the simulation makes it a *dynamic* load balancing (DLB) technique.

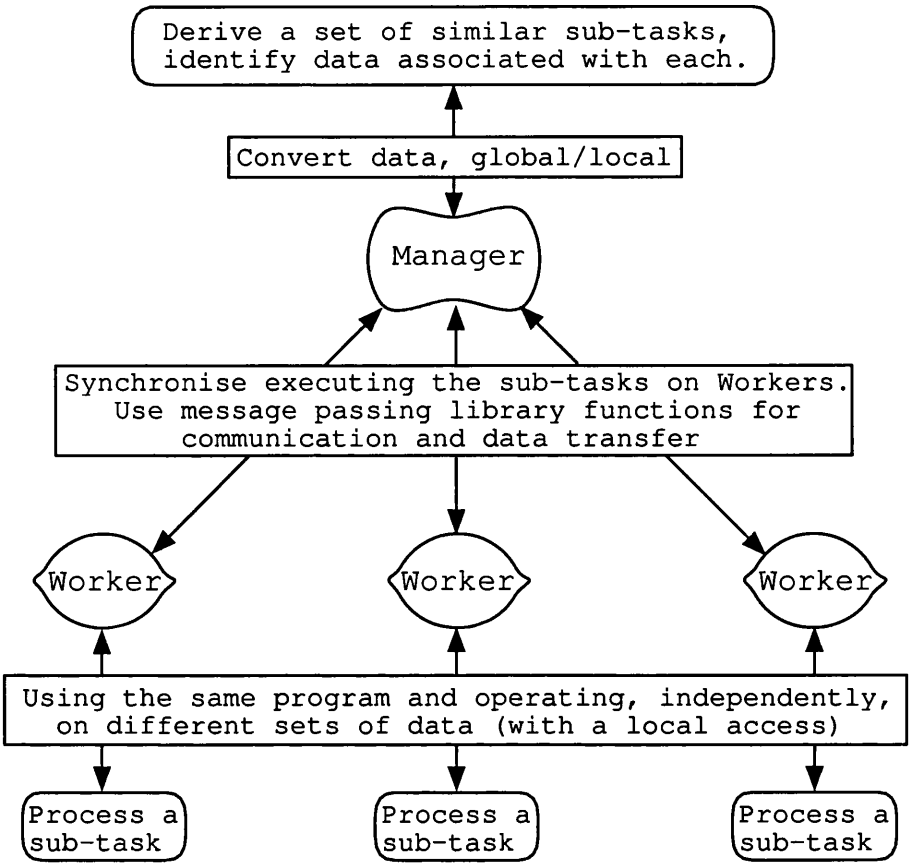


Figure 4.1: Illustration of the main concepts in the parallel template adopted in this research. (MPL) programming model, with (SPMD) structure and Manager/Workers mechanism.

DPP is not a DLB technique

Previous reports of this research has used the Dynamic Load Balancing (DLB) term to express the technique described above as Dynamic Parallel Processing (DPP) [128, 129, 145, 107]. In fact, after some more recent developments, particularly the 'post-processing' load balancing algorithm (see Section 5.4, maintaining the *old* DLB term was found to be slightly confusing. However, examining the DPP technique in more depth should resolve the confusion between the two terms, and demonstrates the different nature of the load balancing in the DPP technique itself.

The DPP is a requirement as well as the 'strategic choice'

The number of processors to be used by an MPI parallel program must be

known before the program can be executed[4]. On the other hand, it is impossible sometimes to predict the exact number of tasks that are due to be processed in parallel, (e.g. total number of the internal boundary within the indirect decomposition method). Obviously, reserving a set of processors in every parallel run *just in case* they become needed is not a realistic option. Therefore, the template design needs to be enhanced such that it can accept an unknown number of tasks whatever the number of processors, and hence the DPP is a 'requirement'!.

In fact, the (DPP) technique exploits some of the main features mentioned earlier, such as the SPMD structure, the Manager/Workers communication policy and the data localisation scheme. The DPP enables the Manager to administrate the processing of N_{task} tasks on N_{Proc} processors. Where both N_{task} and N_{Proc} can be any arbitrary number, including the cases: $N_{task} > N_{Proc}$, $N_{task} < N_{Proc}$ and $N_{task} = N_{Proc}$. The latest case is the most common one in traditional parallel programs, and it is considered as Static Parallel Processing (SPP) in this research.

Within the DPP technique, every Worker initiates an infinite loop, which can be exit by an 'order' from the Manager only. Whenever a Worker finishes processing a task, it notifies the Manager and goes back to 'stand by' again. On the other hand, the Manager starts the DPP technique by sending one task to each Worker, and then whenever a request is made by a Worker a new task is sent back to it. As soon as all tasks are sent out, the Manager starts sending exit messages instead of the new task. A flowchart that illustrates the Worker and the Manager actions in the DPP technique is presented in Figures 4.2 and 4.3.

Recalling the two conditions presented at the beginning of this Section, and the fact that the workload may vary dramatically among the sub-tasks indicates that using the traditional (SPP) will result in having a number of 'idle processors'. Whilst employing less processors and exploiting the (DPP) technique, a more efficient use of the resources is expected. The *efficiency* issue will be discussed in depth in Section 6.3.2, however, it is clear that the DPP can be adopted as a *strategic choice.* In fact, the DPP technique forms the major key behind having the developed programs capable of generating large size grids on regular computing resources.

What is the DPP loop?

The implementation of the DPP technique by both of the Manager and the Workers together is considered as one loop, which is referred to as DPP loop. The DPP loop is summarised below using MPI functions with the C language binding. Clearly, actions of the Manager are completed within four different

[4]Implementations of the recent MPI-2 may have a different policy in this regard

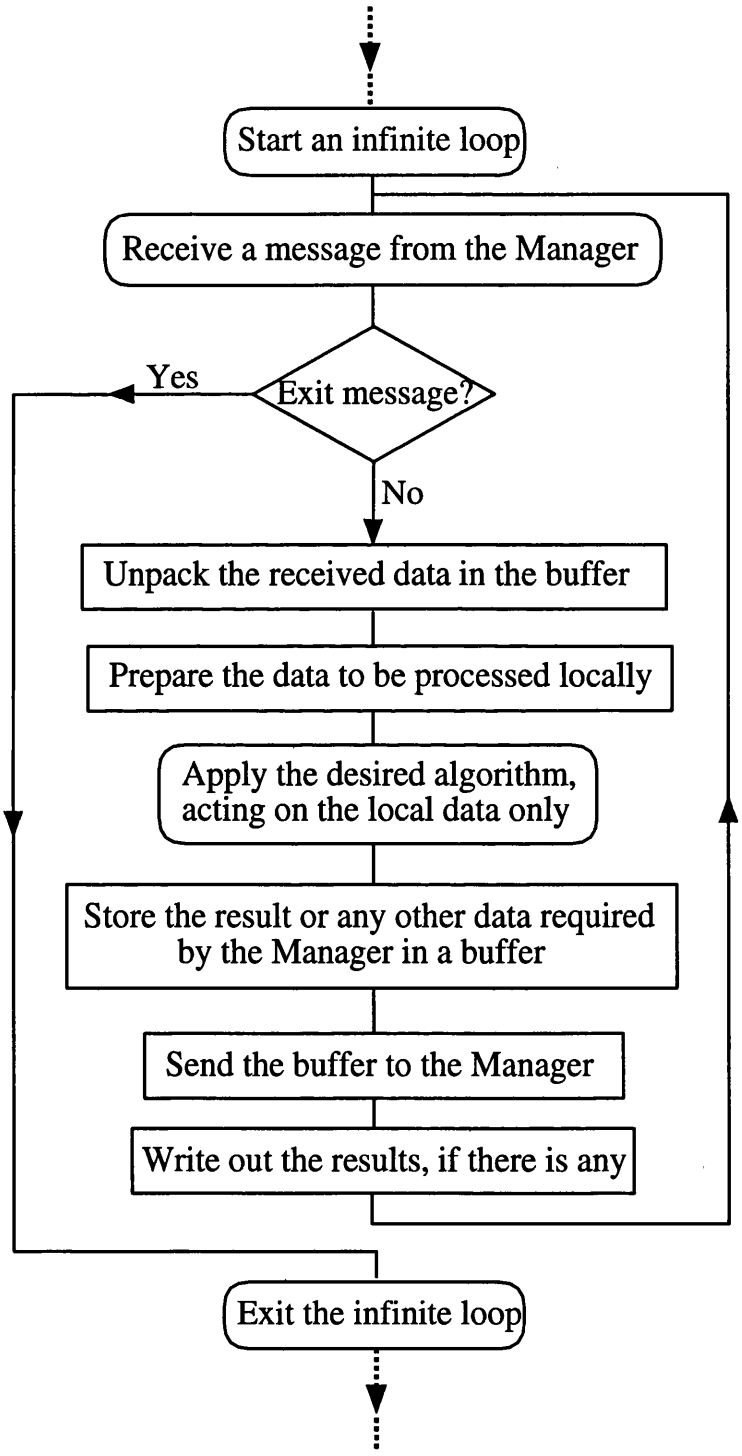


Figure 4.2: Dynamic Parallel Processing loop from the Worker point of view.

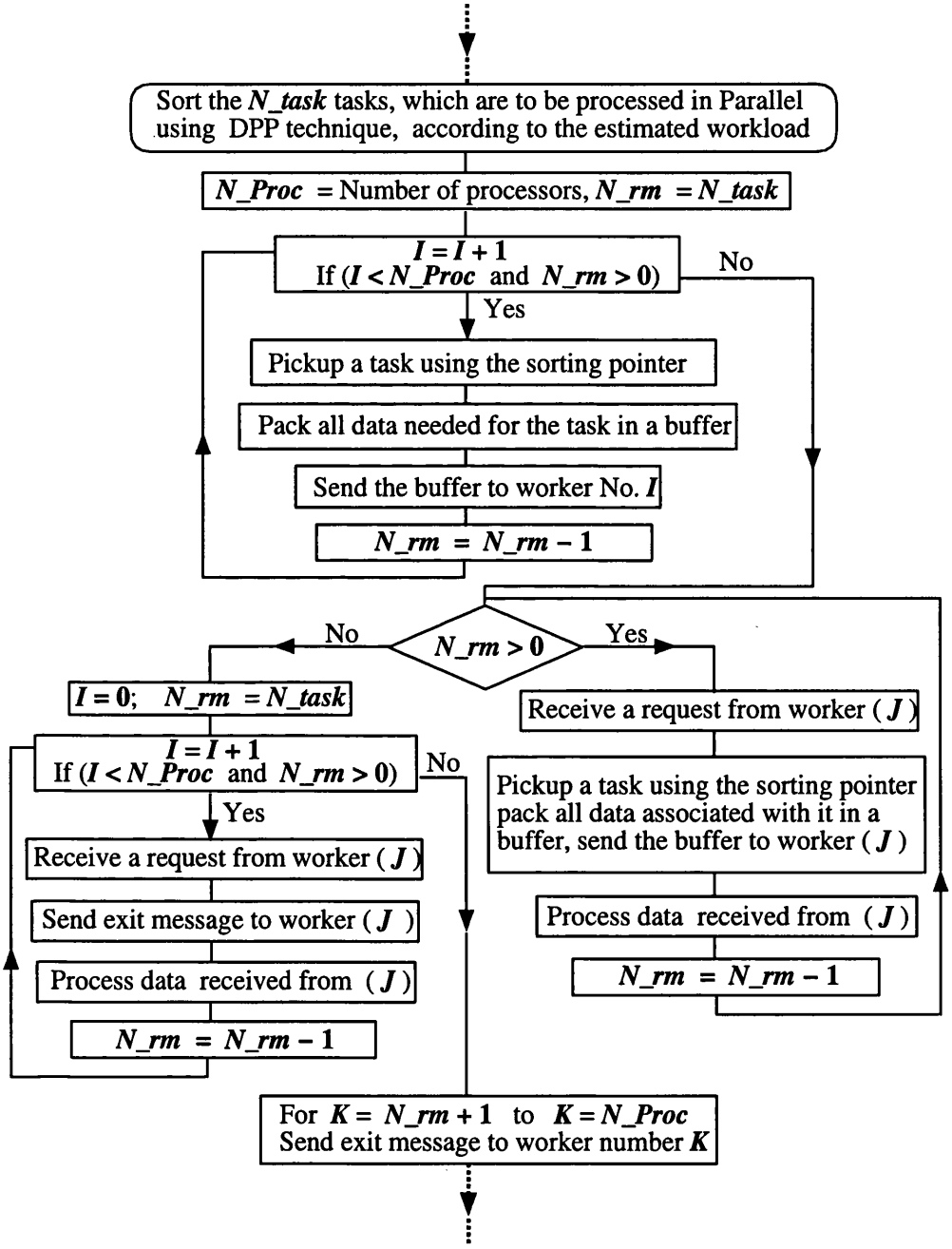


Figure 4.3: Dynamic Parallel Processing loop from the Manager point of view.

loops: 1^{st} loop is to send the first task to each Worker, 2^{nd} loop is to receive results/requests and then send a new task, 3^{rd} loop is similar but an exit message is sent back to the Worker and the 4^{th} loop is for sending exit messages only. Notice that every loop has its own control condition, therefore some of them might be inactive in some cases. For example, the second loop (i.e.`while N_rm > 0`) is not active when $N_{task} < N_{Proc}$.

Notice the use of the `status` data structure in the argument list of the MPI_Recv function in order to find out the information about the message tag and the rank of the sender. Also, notice that the Manager always has the processor with ID number as 0 (`Proc_ID = 0`), whilst any other processor is considered to be a Worker.

A pseudo code of the general DPP loop.

```
        .....
if(!Proc_ID) {           /*  ( THE MANAGER ROLE IN DPP LOOP )  */
        .....
  N_rm  = N_task;
  for(i=1; (i<N_Proc && N_rm>0); ++i){

    /* Use the sorting pointer to pick the next task from the
       remaining.  Pack all required data associated with the     ( A )
       chosen task, and apply a data localisation scheme.   */

     /* Send the buffer to Worker No. (i)  */

     --N_rm;
  }

  while(N_rm>0)

    MPI_Recv( ..... );    /* Receive a message with MPI_ANY_SOURCE */
    ID_sourc = status.MPI_SOURCE;
       .....
     /* Unpack data and carry out some jobs... */
       .....
     /* Repeat what is in ( A ) above */

    /* Send the buffer to Worker No. (ID_sourc) */

    --N_rm;
  }
```

```
  N_rm = N_task;
  for(i=1; (i<N_Proc && N_rm>0); ++i){

    MPI_Recv( ..... );   /* Receive a message with MPI_ANY_SOURCE */
    ID_sourc = status.MPI_SOURCE;
       .....
    /* Unpack data and carry out some jobs... */
       .....
    /*  Send an exit message to Worker No. (ID_sourc) */

    --N_rm;
  }

  for (i = N_Proc -1; i >N_task; --i){

    /* This loop will be active when the number of Workers is bigger
       than the total number of tasks. The only job here is to send
       an exit message to break the infinite loop on Worker No. (i) */
  }

} else {                    /*  ( THE WORKERS ROLE IN DPP LOOP )  */

       .....
  for( ; ; ){

    MPI_Recv( ..... );   /* Receive from the Manager with MPI_ANY_TAG */
    tag = MPI_Status.MPI_TAG

    /* Check, using the value of tag, if this is an exit message
       then free all the memory and break the loop */

    MPI_Unpack( ..... );
        .....
    /* Carry out a job calling a set of different subroutines */
        .....

    MPI_Pack( ..... );  /* pack data to be sent back to the Manager */

    MPI_Send( ..... );  /* Send back the results and a new request */

  }
}
         .....
```

4.3 Parallel Implementation of the General Algorithm

During the course of this research, the indirect decomposition method was the first proposed approach. A parallel program was written, implemented and analysed before having the direct decomposition method investigated as an alternative. Thus, another *new* parallel program for the direct decomposition method was due to be written. Fortunately, the two methods are very similar particularly from the parallel processing view point, and by exploiting the developed DPP loop extensively, things became much easier. In fact, the *new* parallel program was built by inserting minor changes to the *old* one. Therefore, we discuss the parallelisation of the two methods using a generic form of the general algorithm which represents the two algorithms presented in Figures 3.12(page54) and 3.29 (page76). The parallelisation of individual steps in this general algorithm is then discussed.

Layout of major steps in the 'general algorithm':

1. Read the input data: surface grid on the original boundary and the background grid with the point spacing sources.
2. Apply a domain decomposition algorithm to partition the volume inside the computational domain into N sub-domains.
3. Derive M new boundary surfaces, which are created between the adjacent sub-domains (i.e. internal boundary).
4. Generate a smooth grid on the M internal boundary surfaces.
5. Construct the boundary (original and internal) surface grid for N closed sub-domains.
6. Adjust the boundary faces orientation and generate a volume grid inside each of the N sub-domains.
7. Apply a post-processing procedure.

A set of *major* subroutines has been created by encapsulating all techniques that are related to individual steps in the general algorithm above. Message passing operations associated with each 'task' have also been grouped in individual subroutines. On the top of all these subroutines a *main* program is constructed, which starts the overall procedure by initialising the parallel processing environment. The initialisation procedure includes: finding the total number of involved processors, assigning a processor to be the Manager and

the rest as Workers, establishing connections and synchronising all processors. Having done that, every *major* subroutine is called and executed by the Manager (when it consists of sequential procedures only) or by the Manager and the Workers (whenever it involves parallel processing of a task).

Step 1
No attempt has been made to parallelise this step, though there would have been a possibility if a more recent version of MPI was adopted. To the best of our knowledge MPI-2, which became available around the mid of 1999, was the first version of MPI provided parallel I/O facility. Nevertheless, the time needed to read the input data (surface grid on the original boundary and background grid with sources) has always been very small in comparison to the time needed in other parts of the program overall, see Section 6.3.4.

Step 2
Workload involved in this step varies dramatically between the two methods. The need for generating an 'initial' volume grid sequentially has made the indirect decomposition method a rather inefficient procedure. However, despite having this step as a bottleneck, in the indirect decomposition method case, no attempt at all has been made to parallelise it!. In fact, no suitable parallel processing solution was possible, and this indeed has been one of the motivations behind developing the direct decomposition method.

In short, it is extremely difficult to parallelise Step 2 in the indirect decomposition method, and there is no need in the direct decomposition method case. In fact, partition of the computational domain of a grid in the order of 100 million tetrahedra can be accomplished within less than a few minutes, see Section 6.3. However, with the possibility of employing a more advanced partitioning procedure, such as an octree based technique with a sophisticated workload balancing, in the future parallel processing of this step may become essential.

Step 3
If predicting which sub-domains are adjacent to each other is straightforward in the direct decomposition method, unfortunately it is almost impossible in the indirect decomposition case[5]. Deriving an internal boundary in the direct decomposition method can be accomplished by accessing the triangles on two sub-domains boundaries only, whilst in the indirect method access to *all* tetrahedra in the initial volume grid is a must. In other words, it is clear in both methods that, if the internal boundaries were to be derived in parallel, a large amount of data has to be made available to all Workers first. Bearing in mind the high cost of data communication in contrast to data processing, it is very unlikely to benefit from processing Step 3 in parallel. Thus, deriving the internal boundary has been carried out sequentially in both programs.

[5]See both algorithms presented in Sections 3.2.3 and 3.3.3.

Step 4

Probably, the most challenging task introduced by a geometrical partitioning based parallel grid generator is the gridding of the internal boundary, see Table 2.2. It is very likely that this issue will be a subject of further research in the next few years [22], and new approaches may emerge. Nevertheless, we strongly believe that, however the gridding technique develops, the parallel processing of it will remain a vital point for the overall efficiency of the program. Thus, to parallelise Step 4 or not is out of question, and it is left to verify how the developed template has been used.

A typical implementation of the adapted template requires the Manager to filter both of the internal boundary data and the background grid, with the grid point spacing control parameters, by a data localisation scheme. In fact, this has been implemented on the internal boundary data only; apparently, extracting the grid point spacing data for each internal boundary is more expensive computationally than sending a *full* copy to each Worker. The Manager *packs* the internal boundary data (after been numbered locally) with the background grid and sources data and *sends* them to a Worker. The Worker in turn, after *receiving* and *unpacking* the data, generates a smooth surface grid, then *packs* the result and *sends* it back to the Manager. The operations (pack, send, receive, unpack) are carried out using standard message passing functions.

The algorithms that carry out the gridding of the internal boundary vary between the two methods, for example, an extra 2D triangulation algorithm is needed within the direct decomposition method. Subsequently, some differences in the communicated data itself are expected (e.g. a set of triangles in indirect decomposition method against a set of edges in direct method). In fact, this is one of the places where some modifications had to be introduced into the old parallel framework while the new one was under development.

Each of the surface grids on the internal boundary is sent back to the Manager in its own local numbering, whilst some nodes and edges may co-exist on other boundary grids. Thus, a linking and renumbering operation must take place by the Manager in order to construct a one global 'valid' grid. In fact, this procedure is examined thoroughly in Section 5.2.1, however, it is appropriate to mention that *by the end of Step 4 one global surface grid covers both original and all internal boundaries and that this grid is available on the Manager.*

Step 5

In Step 5 a number of sub-domain closed boundaries have to be derived from the global surface grid, and once again the question is 'to parallelise or not to parallelise?!'. In fact, the discussion presented in Step 3 (about the communication cost if the adopted template was to be employed) is applicable on this Step as well. Thus, Step 5 is carried out sequentially by the Manager, and a data localisation scheme is applied on each sub-domain boundary grid before it is packed and sent to a Worker.

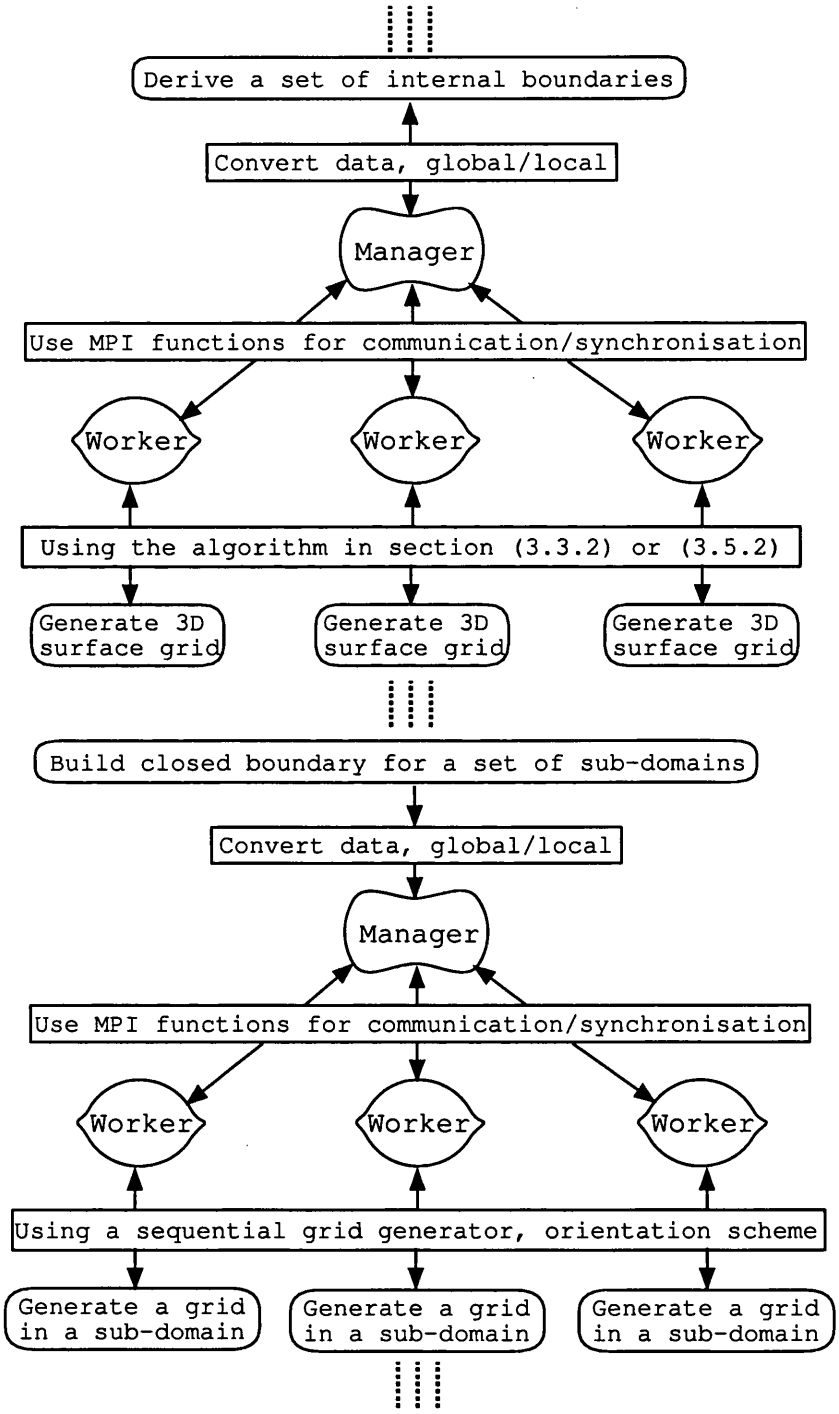


Figure 4.4: Enforcement of the adopted parallel template on the general algorithm. Two sub-tasks only are carried out in parallel, i.e. generating surface grids on the internal boundary and generating volume grids in the sub-domains.

Step 6
The background grid and sources data, which are to be utilised by the sequential grid generator, have already been made available to the Workers at Step 4. Accordingly, data to be communicated in this step consists of the sub-domain boundary surface grid. Thus, having the boundary grid of a sub-domain received and unpacked on a Worker, the two major subroutines (i.e. faces orientation and volume grid generation) can be executed immediately. Obviously, the two subroutines are carried out on the same Worker and there is no need for any 'message passing' within the procedure for one sub-domain.

Step 7
Actions to be carried out by a Worker at the end of Step 6 may vary dramatically, based on the user choice (e.g. a copy of each grid is sent back to the Manager, or written out to disc, or written out with *some* of its data only sent back to the Manager, ...etc.). However, from the grid generation point of view the algorithm terminates here, and any other procedure evolves after this point is ranked as a *post processing* activity, see chapter 5.

4.3.1 Parallel Framework with DPP

It is recognisable that the parallelisation of the general algorithm is completed by enforcing the adopted parallel template twice: first to generate the internal boundary grids, and secondly to generate the sub-domain grids. The enforcement has shown that the adopted template is very flexible, and a quick comparison between Figures 4.1 and 4.4 clearly shows this.

Recall the discussion on the parallelisation of individual steps in the general algorithm presented above, and the flowchart in Figure 4.4. It is straightforward to conclude that constructing a general parallel framework using the DPP technique can be complete by employing two DPP loops only. The first one is associated with the generation of surface grids on the internal boundary and the second with the volume grids in the sub-domains. Now that all aspects of the parallel implementation are elucidated, a *parallel version* of the general algorithm presented in page 91 can be presented in a similar manner:

Parallel version of the 'general algorithm'

1. (*Manager*), read the surface grid on the original boundary
2. (*Manager*), partition the domain into N sub-domains.
3. (*Manager*), derive and sort M internal boundary gridding sub-tasks.

↓ **Start DPP Loop**

4-1. (*Manager*), administrate the processing of M sub-tasks in parallel.

4-2. (*Worker*), generate a smooth surface grid on an internal boundary.

↑ **End DPP Loop**

5. (*Manager*), derive and sort N sub-domain gridding sub-tasks.

↓ **Start DPP Loop**

6-1 (*Manager*), administrate the processing of N sub-tasks in parallel.

6-2 (*Worker*), generate a volume grid inside a sub-domain.

↑ **End DPP Loop**

7. Apply a post-processing procedure (`which may involve several DPP loops in some cases`).

4.3.2 Impact of DPP on the Parallel Framework

In order to demonstrate some features of the developed framework, particularly the role of the DPP technique, we present a rather simple example of a small size grid generated a few times using different number of processors. The grid is of a configuration of a civilian aircraft inside a hemisphere. The computational domain has been partitioned into four sub-domains using the direct decomposition method, details about the generated grid and the time needed in each run are presented in Table 4.1. References will be made to Figure 4.5, which contains screen dumps of the UPSHOT tools[6] [72].

It is recognisable in Table 4.1(I) that the workload is badly distributed among the sub-domains. Utilising the SPP technique, i.e. using four Workers, it is clear in Figure 4.5 (IV) that some Workers do not contribute for most of the run time and just wait for Worker No. 1 to complete. Whilst Figure 4.5(II), where the DPP technique is active and two Workers only are employed, shows more efficient use of the available resources. Of course, as a total time SPP

[6]UPSHOT is a very useful tool for understanding parallel programs behaviour. It offers a graphical display of parallel time-lines. Each line is associated with a processor, and coloured bars reflect the state of the processor at any time can be utilised, see appendix B, chapter 6.

Domain	No. Tetra.	No. Points	No. Trian.
2	306618	56729	35620
1	111642	21643	16422
3	120178	22924	16318
4	73422	13697	8846
Total:	611860	114993	77206

(I)

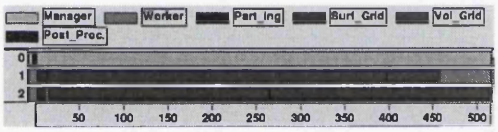
No. Proc.	Time (Sec.)
1	963
2	517
3	394
4	385

(II)

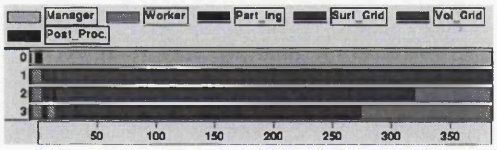
Table 4.1: Details of sub-domain grids (I) and timing using DPP on 1-4 processors (II). The surface grid on the original boundary consists of 63362 Triangles and 31683 Points. Time needed for generating the same grid sequentially = 965 Sec.

will always be better, see (II) in Table 4.1, but on the other hand it is always more expensive and less efficient.

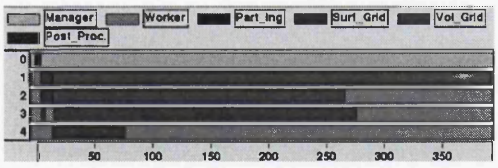
(I)



(II)



(III)



(IV)

Figure 4.5: Using Dynamic Parallel Processing technique to process seven sub-tasks, generating volume grid in four sub-domains and surface grid on three internal boundaries, in parallel. (I) using one processor, (II) using two processors, and so on.

Observing the four cases presented in Figure 4.5 reveals that, in (I) and (II) ($N_{task} > N_{Proc}$) for both DPP loops, while ($N_{task} = N_{Proc}$) in the first DPP loop in (III) and ($N_{task} < N_{Proc}$) in the same loop in (IV). Various number of processors are used, including one single processor see (I) in the figure. So, it is visible how the enhancement of the adopted template by the DPP technique has produced an adequate parallel processing framework. Such that *unknown* number of tasks now is always acceptable, and reserving a set of processors for 'just in case' is not required any more.

4.4 Concluding Remarks

This chapter has demonstrated in general the parallel implementation of the two methods developed for generating large size unstructured grids very efficiently in parallel. A parallel processing template which implements the MPL model with SPMD structure and Manager/Workers mechanism, in addition to a special technique called the Dynamic Parallel Processing (DPP), has been discussed thoroughly. Constructing a computational framework based on this template has been examined, and a generic form of the general 'parallel algorithm' has been presented. Benefiting from the DPP technique in general has been highlighted, and further discussion of its impact on the framework and the overall performance will appear in the next two chapters.

Chapter 5

Issues Associated with the Geometrical Partitioning Approach

5.1 Introduction

This Chapter presents three different issues that are naturally associated with the geometrical partitioning approach. The first one is the demand for establishing a link (namely inter-domain communication tables) between the independent local numbering systems used in the sub-domains grid. The second issue to be addressed is the quality of generated grids in comparison with the traditional sequential ones. And the last issue is the need to shuffle elements among the sub-domains in order to ensure highly balanced distribution of the total workload.

Not surprisingly, the Chapter is divided into three main sections, where each one is devoted to discuss all the aspects that are related to one of the issues. Broadly speaking, every section starts by addressing the main cause and introducing all relevent topics. Following that, an algorithm developed in order to solve the associated problems is presented. An example is then given to demonstrate the approach.

All algorithms presented operate independently from the method used in generating the grids. In general, grids generated using the direct decomposition method have shown some advances to the ones generated using the indirect decomposition.

5.2 Inter-domain Communication

It has been mentioned in Section 4.2 that a 'data localisation' scheme is introduced into the general parallel template in order to minimise the memory usage on Workers. It is shown that the Manager performs this scheme on each task before it is assigned to a Worker. This results in having entities of the sub-domain grids (i.e. points, edges, boundary faces and volume elements) numbered locally per sub-domain and independently from the original boundary grid as well. On the other hand, parallel algorithms require some data to be transfered among the sub-domains themselves and from the original boundary to the sub-domains throughout the simulation. To ensure the continuity in the solution such data must be transfered via 'communication tables', which always secure the matching between the sending and the receiving grid entities. Demonstrating how such communication tables are established is the main objective of this Section, so first of all we review the two numbering systems which have been alluded to several times so far. Following this we discuss different types of communication tables, and then demonstrate the algorithm developed to construct *point to point* type tables.

5.2.1 Global and Local Numbering Systems

It is well known by now *why* there are two different numbering systems, but probably more details about their features and interlinks are necessary before exploring some of their use throughout the program. The global system is established as early as the first step in the general algorithm and remains until the very last step of it. Broadly speaking, the global system is used for the 'public' objects, i.e. objects required by more than one sub-algorithm, like the surface grid on the original and internal boundaries or the background grid and sources. Whilst a local numbering system is a 'temporary' one, and usually associated with individual tasks processed on the Workers. Such that, a local numbering system is created whenever a task is to be sent to a Worker and it is destroyed as soon as the task is done. Apparently there might be more than one local system, but only one global system, at any time. The global system may interact with the local systems, whilst local systems are always independent from each others.

Whenever a local system is created and, if and only if, there is a need for further communication with the global system, a list of pointers is established, which is referred to as the local-global pointers list. Such a list gives every item copied across from the global to the new local system a unique entry number, which points to the origin of the item. Objects in the global system maintain the same numbers for their entities throughout the program, though the contents may go through some modifications. Also, any data manipulation

that takes place in a local system is recognised within that system only, unless the contents are transfered to other systems explicitly.

Linking a local system back to the global system

Some of the local systems may produce 'new objects' which are needed to be used later in the program [1], or may introduce some modifications to an 'old object' (i.e. any item was copied across from the global system). In either of the two cases, entities required from the local system must be linked back to the global system through a 'filtration' procedure. A simple flowchart which illustrates the linking of a local system back to the global system is presented in Figure 5.1.

Creating the local systems and linking them back are, more or less, embedded within the communication operations which take place between the Manager and Workers. In addition, it is the Manager's responsibility to ensure when the local systems are linked back such that no overlapping or gaps occur in the global numbering system.

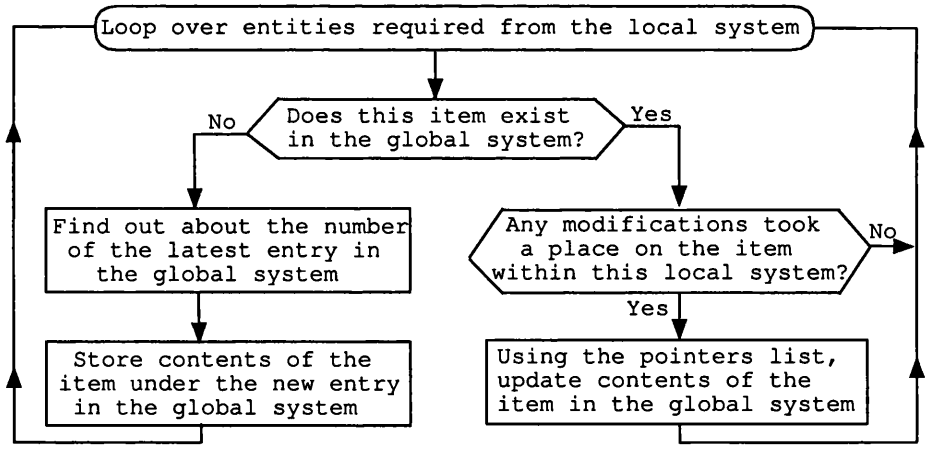


Figure 5.1: The procedure for linking a local numbering system back to the global numbering system.

5.2.2 Constructing Communication Tables for Local Systems

Different types of communication tables

Numerical algorithms across the computational engineering field may differ in the way of utilising the available grid. Some algorithms may carry out the

[1] See Section 4.3 for the discussion of enforcing the parallel processing template on Step 4 in particular. Also, another example of such local systems will appear next Section in 5.3.2.

simulation on element based calculations whilst others may depend on edge or point based ones. Also, some may define the boundary conditions on surfaces, while others may do that explicitly on the grid faces/points, and similarly in applying the external loads. Hence, parallel versions of these algorithms would require information to be transfered among sub-domains in many different ways. Apparently, when an algorithm uses element based calculations then, ideally, the information must be transfered through element to element communication tables, and edge based calculations through edge to edge tables, and so on.

Although, only one type of communication table is produced by the programs (i.e. point to point) some other useful information is provided in case building a different type of communication tables is sought:

- Volume elements within the sub-domain grids which have at least one boundary (internal or original) face are identified. This set of elements forms a 'shell' for every sub-domain which contains all the volume elements that may communicate with other sub-domains. Obviously, having these 'shells' defined in all sub-domain grids must help in constructing element to element communication tables.

- Each original boundary face on a sub-domain grid has its surface number from the global system reserved. This can be exploited to transfer boundary conditions and loading applied on original boundary *directly* from the global system to the local systems.

- Faces of every internal boundary will have the same order whenever they appear in the closed boundary faces list of the two adjacent sub-domains. Also, every internal boundary face, within each sub-domain, has the number of the other adjacent sub-domain available. These two features should make the constructing of face to face or edge to edge communication tables a straightforward procedure.

Constructing Point to Point Communication Tables

Constructing communication tables that can fulfil all the conflicting demands mentioned above might be very expensive, and therefore a choice has to be made. An algorithm which produces point to point communication tables has been developed. In fact, this type is the only one that can cover all the possible configurations which may exist for the contacts between any two sub-domains. For example, if an algorithm with face to face tables is adopted then no communication will be established between two sub-domains which share edge(s) or point(s) only. However, though the algorithm is developed and used to generate point to point tables only it can be easily extended to

produce other types as well. On the other hand, having the point to point tables available, in addition to some of the extra information listed above, it is also possible to construct any other type of communication tables in a 'post processing' step.

The developed algorithm relies heavily on the outcome of a previous step in the general algorithm, i.e. linking the internal boundary grids back to the global numbering system. So before any sub-domain closed boundary is built, the surface grid on all boundaries (internal and original) are stored according to the global numbering system. Any grid entity (point, face or edge) that exists on more than one sub-domain boundary will have more than one entry from the local-global pointer lists. Thus, by examining all entries in these lists, using the global numbering index, the inter-domain communication points are identified.

Having *all* communication points been identified within the global system, a data structure of information that correspond to these points in every local system is built. Obviously, such a data structure should take into consideration the efficiency in searching through massive amounts of data, bearing in mind the scalability with problem size and the total number of sub-domains. A naive approach would lead to an expensive procedure. For an intensive discussion of constructing advanced data structures and search algorithms the reader is advised to consult [9, 77, 38]. However, a flowchart which summarises major steps in the algorithm is presented in Figure 5.2, and an illustrative example is demonstrated between Figure 5.3 and Table 5.1. The efficiency of the algorithm is discussed shortly.

Two 'point to point' communication tables for every sub-domain

As presented in the flowchart, the developed algorithm creates two separate communication tables for each sub-domain. The first communication table consists of one binary list only, which interprets the local-global pointers directly. This list has in its first column the *local* numbers of all original boundary points exist on this sub-domain, whilst the second column has the *global* numbers of the same points. See Figure 5.3 and Table 5.1. The second communication table must have n binary lists, where n represents the total number of sub-domains that share one point or more with this sub-domain, every list corresponds to one sub-domain. The first column in each list has the local numbers of the boundary points (original and internal) shared with the relevant sub-domain, while the second column has the local numbers of the same points in the other local system.

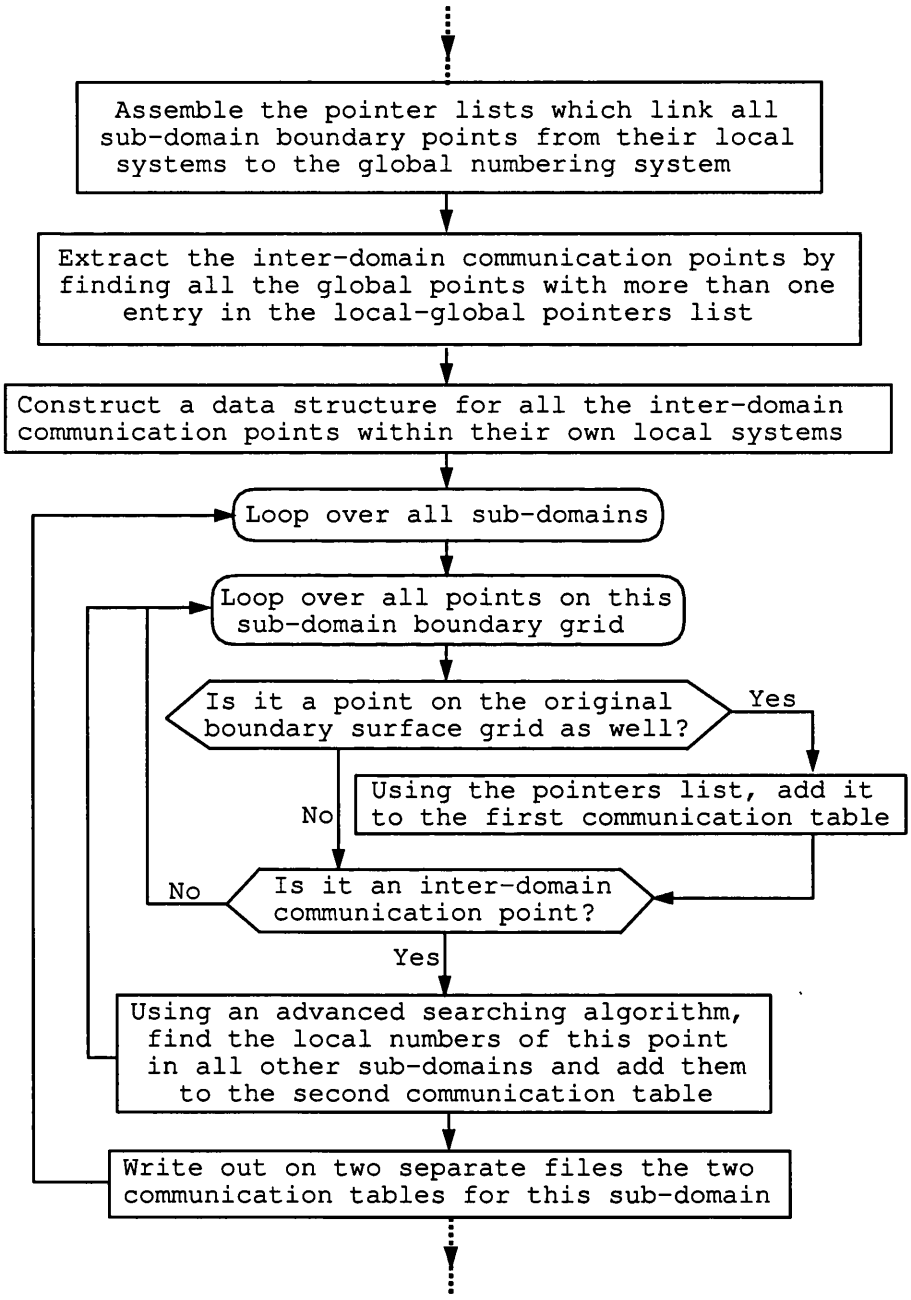


Figure 5.2: Algorithm for extracting the Inter-domain communication data, including the communication between the sub-domains boundary and the original boundary as well.

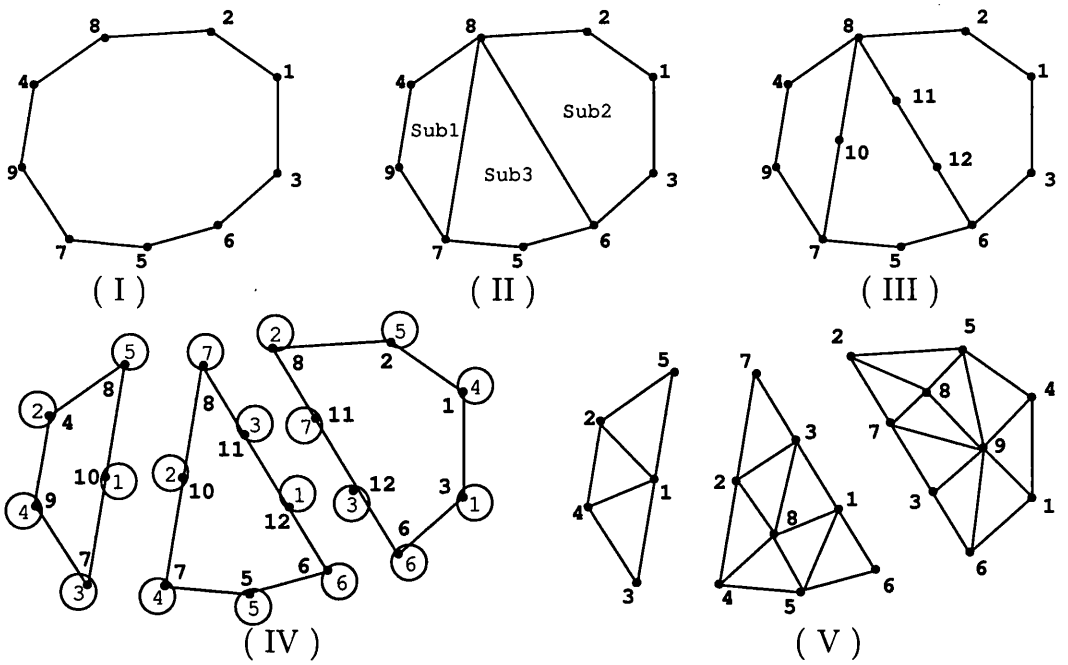


Figure 5.3: Illustration of the point to point communication algorithm, (I) Original boundary with global numbering system, (II) Three sub-domains and two internal boundary at the end of the domain decomposition procedure, (III) Grids on the internal boundary are linked back to the global system, (IV) pointer lists are established while the local systems of sub-domains being created, (V) local systems only exist in the final grids and inter-domain communication tables have to be constructed. See the two communication tables for Sub1 in table 5.1.

First Commun. Table	Second Commun. Table	
With original boundary	With Sub-domain No.2	With Sub-domain No.3
Total = 4	Total = 1	Total = 3
2 4	5 2	1 2
4 9		3 4
5 8		5 7
3 7		

Table 5.1: An example of point to point Inter-domain communication tables, see Sub-domain No. 1 in Figure 5.3. The first columns contain local numbers in the considered sub-domain (No. 1), and the second columns contain the same point local number in the other sub-domains(Second Table), or the global number in the surface grid on the original boundary(First Table). Notice that sub-domains 1 and 2 share one single point only.

It is appropriate to mention that the same algorithm discussed above can still be applicable to construct other types of communication table. The only

required modification would be to have the local-global pointers established based on the boundary grid faces (or edges) instead of the boundary grid points.

Cost of constructing the inter-domain communication tables

So, is the developed algorithm capable of providing the required information within reasonable time?. Figure 5.4 presents the total time needed to construct and write out all the related files on the disc for a grid of size in the order of 20 million tetrahedra. The grid has been generated several times within a different number of sub-domains, where the number of boundary faces changes as shown in Tables 5.2. Obviously, the time required for constructing the communication tables is effected only by the total number of boundary points and the number of sub-domains. In another words, it is *independent* of the complexity of the gridded configuration. However, it has been observed that this time is always very small in comparison to the total run time. This is discussed thoroughly in Section 6.3.4 in respect to the scalability of the program overall.

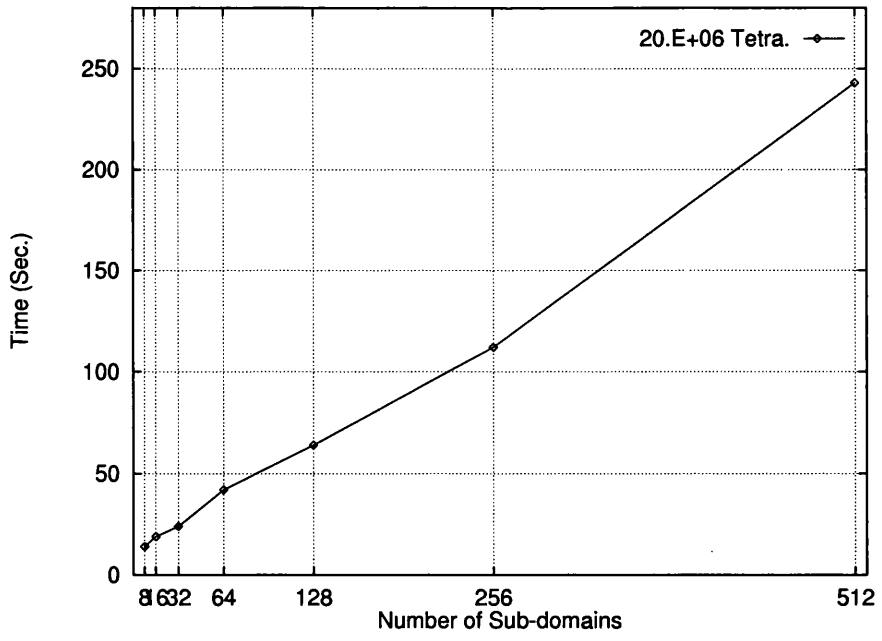


Figure 5.4: Time required to establish the communication binary lists, the volume grid consists of 20 million tetrahedra. Number of sub-domains in the range $[8 - 512]$ associated with number of boundary faces in the range $[0.9E+6 - 8.78E+6]$.

No. of Sub-domains	No. of Boundary Faces
8	953732
16	1076412
32	1323276
64	1815832
128	2808748
256	4796372
512	8764620

Table 5.2: Number of faces on the sub-domains boundary, including both internal and original one. Time needed is presented in Figure 5.4.

5.3 Grid Quality

In general, the quality of a solution obtained by a numerical algorithm depends on the quality of the grid used to discretize the computational domain. It is a well recognized issue: employing a poor quality grid has a direct impact on the quality and efficiency of the solution overall. For example, in the time-dependent solvers the simulation advances based on a time step, which may have to become very small if there were some badly distorted elements in the grid. Consequently, an excessive number of time steps would be required to complete the simulation[2]. ".... these algorithms are rather sensitive to the quality of the grid being employed, and so great care has to be taken in the generation process. The improvement of grid quality is problem of major importance.... " [58].

Examining and improving the quality of unstructured grids

If displaying a small size surface grid on a computer screen can be a feasible way to check the quality of its elements, then for a grid size in the order of several million elements the approach is not practical. Therefore, some computable measurements have been introduced and adopted as a common way to inspect grid quality. Some of the well known measurements for tetrahedral grids are: minimum dihedral angle per element, ratio of volumes of two adjacent elements, ratio of maximum to minimum edge length per element (point), number of elements surrounding a point and alpha = average edge length **3/volume [147].

Further consideration has been given to the grid quality issue, so quality enhancement techniques have been developed. Operations such as diagonal swap-

[2]Some algorithms may introduce a smaller time step locally at these bad elements only, however, in general this can effect the quality of the solution overall.

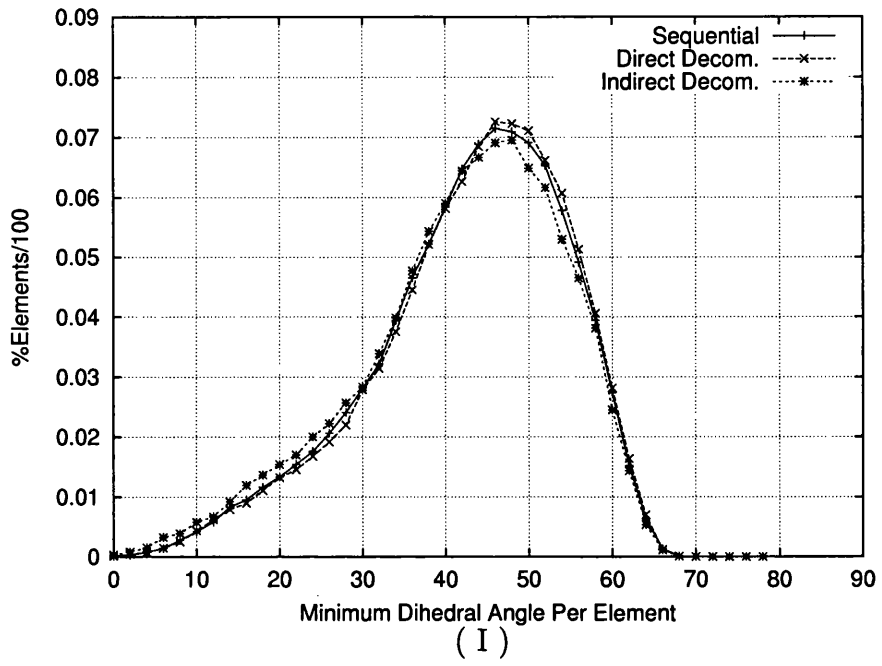
ping, element reconnection, element removal and grid point relaxation are integrated as a default post processing step within some grid generation procedures nowadays [58]. However, by applying such operations, with a predefined criterion considering optimal values of the measurements mentioned above, improvements in the quality of the grid overall must be noticeable. Also, the number of severely distorted element must be kept to the very minimum.

5.3.1 Impact of the Geometrical Partitioning Approach on the Grid Quality

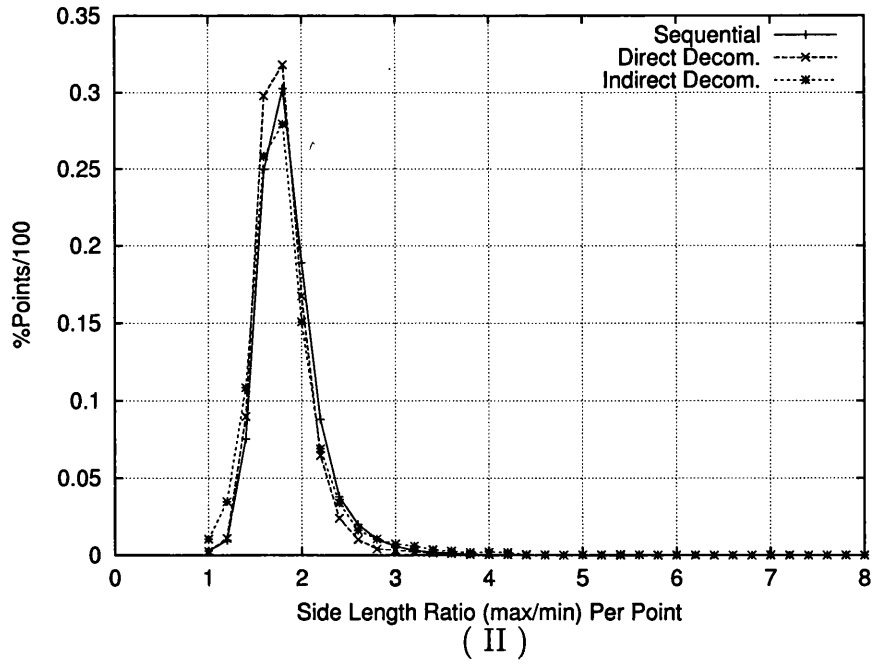
Having the same traditional sequential grid generators employed in generating the sub-domains grid, one may expect the same level of element quality in both sequential and parallel grids. In fact, although this has been achieved often some differences are always expected. In general, the level of element quality obtained in unstructured volume grids is strongly connected to the complexity of the gridded geometry as well as the quality of the boundary surface grid. Recall that the geometrical partitioning approach leads to the gridding of a set of *new* configurations (i.e. the sub-domains) which may have substantially more complex boundaries than the original configuration, see Sections 3.2.3 and 3.3.3. Encountering some differences in the element quality between grids generated sequentially and grids generated in parallel is totally understandable. However, obviously what is important at the end is to find out if the quality of grids generated in parallel is still satisfactory or not.

A volume grid of a configuration of a civilian aircraft inside an hemisphere has been generated three times: sequentially, in parallel using the indirect decomposition method and in parallel using the direct decomposition method. The same surface grid on the original boundary and the background grid with resources have been used in the three different runs. Minimum dihedral angle per element and ratio of the sides length (maximum to minimum) per point are inspected, and the results are presented in (I) and (II) Figure 5.5. It is noticeable, that although there are some differences the quality of grids generated in parallel is still very similar to the one generated sequentially. Furthermore, a more important and realiable validation of parallel grids is demonstrated in the next Chapter, see Section 6.4, where some grids generated by the developed programs are employed in realistic problems.

It has been noticed that sometimes, despite of the high quality surface grid on the boundary, some distorted elements may still exist locally in the volume grid. Two cases have been identified: the first case, which may appear very frequently in the indirect decomposition method, is when two boundary surfaces meet forming a very sharp corner. The second case is when a small gap, i.e. less than the minimum *local* point spacing, separates an internal boundary



(I)



(II)

Figure 5.5: Statistics of quality measurements in a grid generated sequentially and in parallel, minimum dihedral angle per element in (I) and sides length ratio (maximum to minimum) per point in (II).

from the original boundary. Generating a limited number of distorted tetrahedra might be inevitable in such cases. However, a special procedure has been introduced in the direct decomposition method case, by which the locations of the internal boundary are adjusted in attempt to avoid 'roblomatic regions' in domain, see Section 3.3.2. Whilst, the issue in the indirect decomposition methods case could not be recovered effectively.

5.3.2 A Post-Processing Relaxation Algorithm on the Internal Boundary Points

The quality of the sub-domain grids can always be improved by applying some of the enhancement operations mentioned earlier in this Section. It must be straightforward to integrate an element quality enhancement procedure within the parallel framework. Such a procedure can be introduced as a post processing operation applied on every sub-domain volume grid after been generated on a Worker. However, quality enhacement procedures normally operate under the condition by which the boundary surface grid can not be manipulated at all [58]. In the case of grids generated in parallel, all the internal boundary in the sub-domains are 'artificial' and there is very little restriction on manipulating them. In fact, this *extra* option of moving points on the boundary grid has allowed for another grid quality enhacement procedure to be introduced:

A relaxation algorithm operating on the internal boundary points as a post-processing procedure has been investigated. The major steps of the proposed algorithm are summarised in the flowchart in Figure 5.6. The main concept is to assemble all edges in the sub-domain volume grids connected to the internal boundary and simultaneously construct one *local system* on the Manager[3]. A Laplacian relaxation technique is then applied on the new local system, see equation 3.6 in page 50, where only the internal boundary points are 'free' to move.

An illustrative diagram is presented in Figure 5.7, which demonstrates the algorithm using a simple 2D example. The *local effect*, which involves only the elements that have an internal boundary point, of the algorithm is noticeable. A 3D example is presented in Figure 5.8, where the boundary surface grid of a sub-domain is shown before and after applying the relaxation algorithm. The impact on grid quality is demonstrated in Figure 5.9 by comparing statistics of two different quality measurements within a sub-domain volume grid.

The algorithm has been recognised to be a very expensive procedure computationally, and in the light of the 'little' improvements obtained, it has been omitted in more recent versions of the program. Also, no attempt at all was

[3]Choosing the Manager in here was to exploit the other communication operations which take place between the Manager and the Workers in transferring the relevant data.

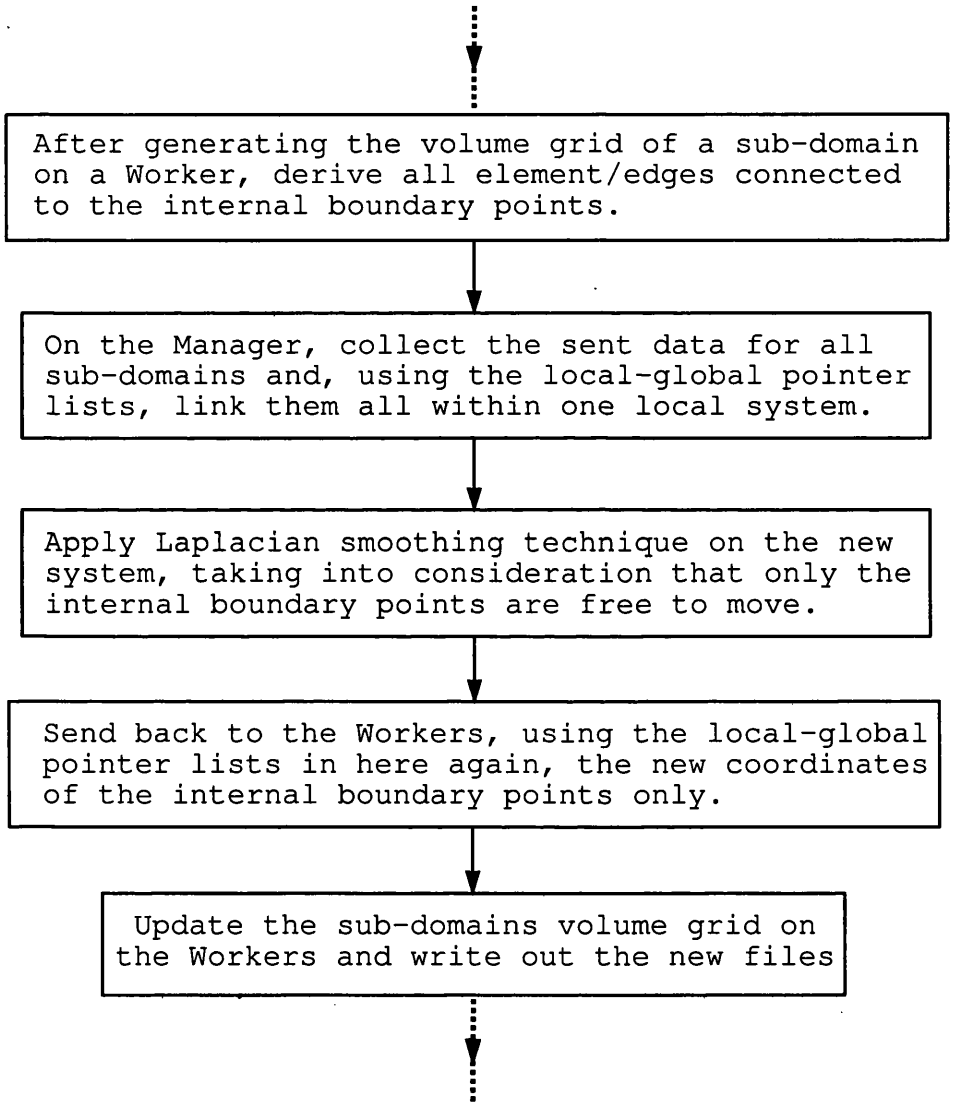
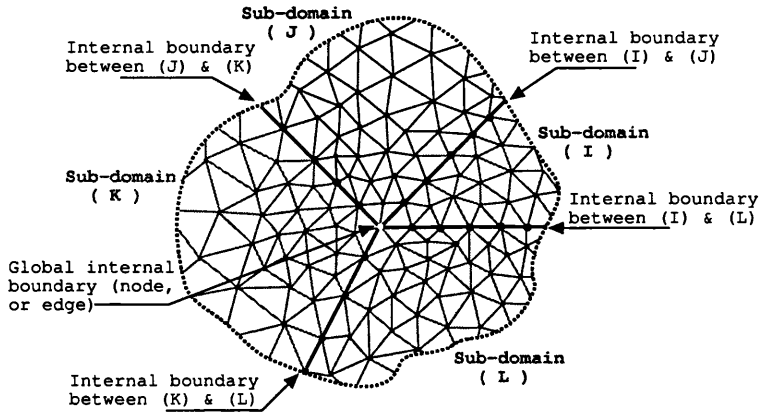
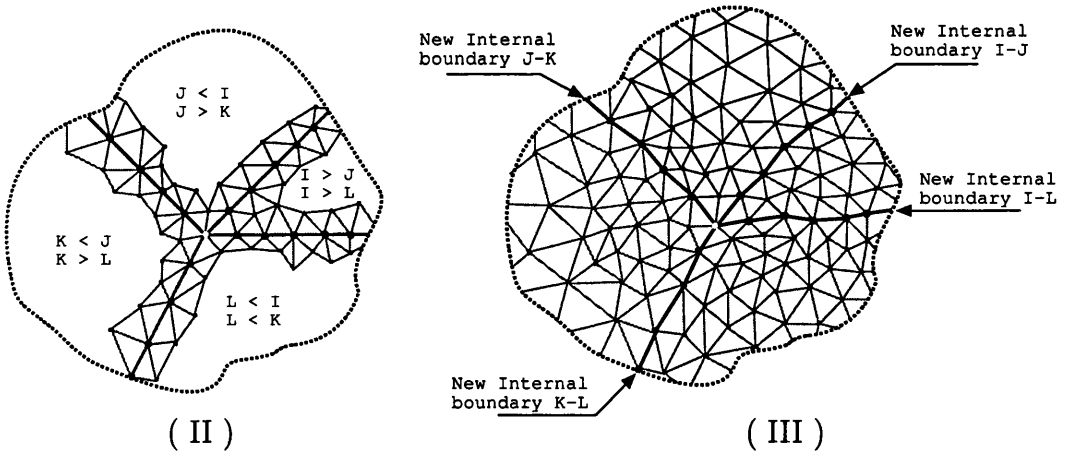


Figure 5.6: Major steps in the relaxation algorithm applied on the internal boundary points as a post-processing procedure.



(I)



(II) (III)

Figure 5.7: Illustration of the post-processing relaxation algorithm on the internal boundary points: (I) a typical configuration of few neighbouring sub-domains, (II) elements/edges connected to internal boundary in every sub-domain are derived in parallel and assembled within one system by the Manager (edges on the internal boundary itself are sent to the Manager from the sub-domain with the higher order only), (III) the new inter-domain boundary (i.e. after applying the relaxation algorithm on the internal boundary nodes).

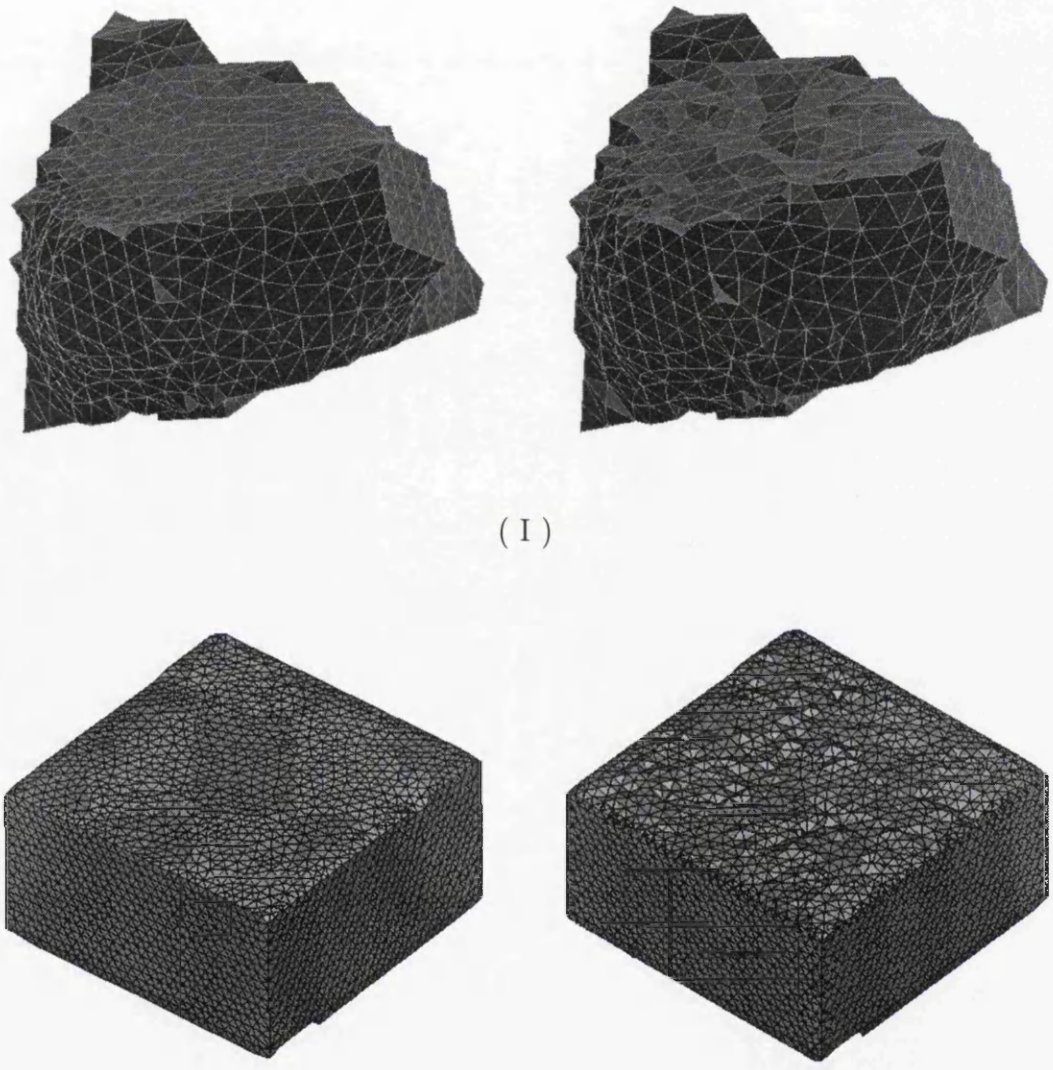


Figure 5.8: The impact of the internal boundary relaxation algorithm on two different sub-domains, (I) generated using the indirect decomposition method and (II) using the direct decomposition method. See Figure 5.7 for an illustration of the 'relaxation algorithm'

made to parallelise it, since a massive number of communication operations would have been unavoidable, particularly if DPP loop was to be used.

5.4 Load Balancing

Previous chapters have shown that achieving a well balanced distribution for the grid elements among the sub-domains could not be secured, neither at the domain decomposition step nor during the volume grid generation. On the other hand, achieving an acceptable performance by a parallel solver requires a well balanced distribution of the total workload among the sub-domains. Therefore, and in order to ensure the effectiveness of the developed framework overall, introducing a *post-processing* workload redistribution technique becomes essential.

Several distinguished algorithms that address the problems of partitioning unstructured grids and distributing the workload evenly have evolved in recent years [30, 112, 31, 6, 83, 74, 75, 133, 134]. These algorithms may vary dramatically in their performance, complexity of the general procedure and quality of the final results. It is beyond our intention to get involved in any sort of comparison or evaluation herein. An interested reader can consult [65, 125, 150], also see Section 3.2.2. Nevertheless, it still ought to be mentioned that most of these advanced algorithms are very expensive in respect to their requirements of computing resources. Therefore, and in order to maintain our global objective of generating large size grids on regular computing resources, seeking a purpose-built load balancing technique is justified. However, the option for using any other partitioning and load balancing algorithm is still available since one global grid of the entire domain can be built, see Section 6.2.2.

The proposed technique consists of two main stages. First, the number of elements that due to be moved out, or attached to, each sub-domain is identified using an elegant algorithm developed by Hu and Blake [66]. Secondly, the element migration is carried out in parallel employing the same developed template as presented in Section 4.2. More details about the two stages are discussed throughout the rest of this Chapter, but before going any further it might be essential to emphasise that the proposed technique was mainly integrated into the framework just as an attempt to have a 'self-contained' parallel grid generation tool. So, we are not in any way proposing the technique as an alternative complete load balancing algorithm, since apparently such work can be a subject for a full PhD research program [73, 132]. However, having the developed framework enhanced by this technique means that if a grid is generated on a certain platform then a *balanced distribution* of its elements will be achievable without the need of any extra resources. Furthermore, we strongly believe that this technique can be more than adequate particularly

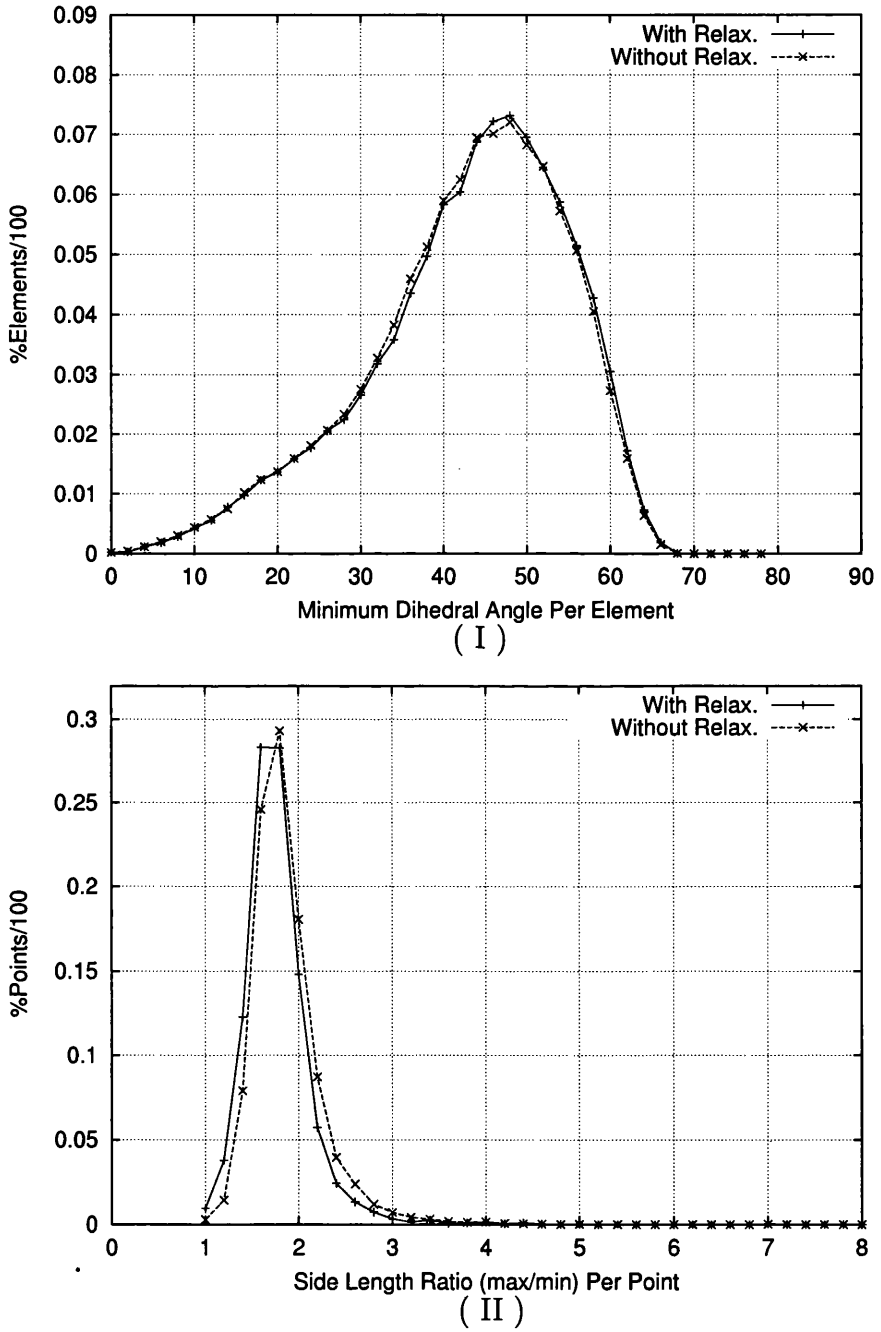


Figure 5.9: Statistics of quality measurements in the same grid, generated twice with and without the post-processing relaxation algorithm. In (I) the minimum dihedral angle per element, (II) the side length maximum to minimum ratio per point.

after improving the initial work load distribution in the direct decomposition method.

5.4.1 Hu & Blake Algorithm

This algorithm works out a schedule for the number of elements that must be moved between any two sub-domains, such that each sub-domain will have the same number of elements on completion. This schedule (termed as elements migration schedule) provides an *optimal* solution by: (a) keeping the element movement to a minimum and (b) restricting the migration among adjacent sub-domains only. Obviously, these two features should help in reducing the computational cost of the load balancing algorithm itself, also to maintain the inter-domain communication cost as low as possible [66, 67].

The algorithm defines a graph (V, E) to represent the sub-domains configuration, where $V = (1, 2, ...P)$ is the set of vertices that represents a sub-domain, and E is the set of edges. There is an edge between two vertices (sub-domains) if and only if they share an internal boundary. Associated with each vertex i there is a scalar l_i representing the load (number of elements) in the sub-domain. Also, with each edge (i, j) there is another scalar δ_{ij} representing the number of elements due to migrate between i and j in order to achieve the average load $\bar{l}$ in every sub-domain.

$$\bar{l} = \frac{\sum_{i=1}^{P} l_i}{P} \tag{5.1}$$

The algorithm converts the problem of finding an optimal element migration schedule into solving the linear equation (5.2).

$$L\lambda = b \tag{5.2}$$

where L is the Laplacian matrix of the graph defined as

$$(L)_{ij} = \begin{cases} -1, & \text{if } i \neq j, (i,j) \in E, \\ deg(i), & \text{if } i = j, \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

and where b is

$$b_i = l_i - \bar{l} \tag{5.4}$$

For a typical example, consider the 6 sub-domains configuration as shown in Figure 5.10, where the symmetric matrix L will be

$$\begin{vmatrix} 3 & 0 & -1 & 0 & -1 & -1 \\ 0 & 2 & -1 & -1 & 0 & 0 \\ -1 & -1 & 3 & 0 & 0 & -1 \\ 0 & -1 & 0 & 2 & -1 & 0 \\ -1 & 0 & 0 & -1 & 3 & -1 \\ -1 & 0 & -1 & 0 & -1 & 3 \end{vmatrix}$$

and having $\bar{l} = 58.83$ the right hand side vector b will be:

$$\begin{vmatrix} 4.17 \\ -29.83 \\ -16.83 \\ -0.83 \\ 13.17 \\ 30.17 \end{vmatrix}$$

It is proved in [66], that using the conjugate gradient algorithm in solving the main equation should give a very fast convergence. An algorithm presented in [37] was coded and integrated. The solution of the presented example is achieved in 4 iterations only:

$$\begin{vmatrix} 7.58 \\ -20.40 \\ -5.19 \\ -5.77 \\ 9.69 \\ 14.08 \end{vmatrix}$$

Having the main equation (5.2) solved for λ, then the number of elements due to move from sub-domain i to sub-domain j is simply:

$$\delta_{ij} = \lambda_i - \lambda_j \tag{5.5}$$

Obviously $\delta_{ij} = -\delta_{ji}$, which represents the fact that if a sub-domain i is to send δ_{ij} elements to j, then sub-domain j is to receive the same amount (to send $-\delta_{ji}$). Since the result from solving the main equation are real numbers, it is necessary to round every δ_{ij} into the nearest integer. By doing so the final load in any sub-domain i will be no more than $deg(i)/2$ away from the average load $\bar{l}$, see Graphs (III) and (IV) in Figure 5.10.

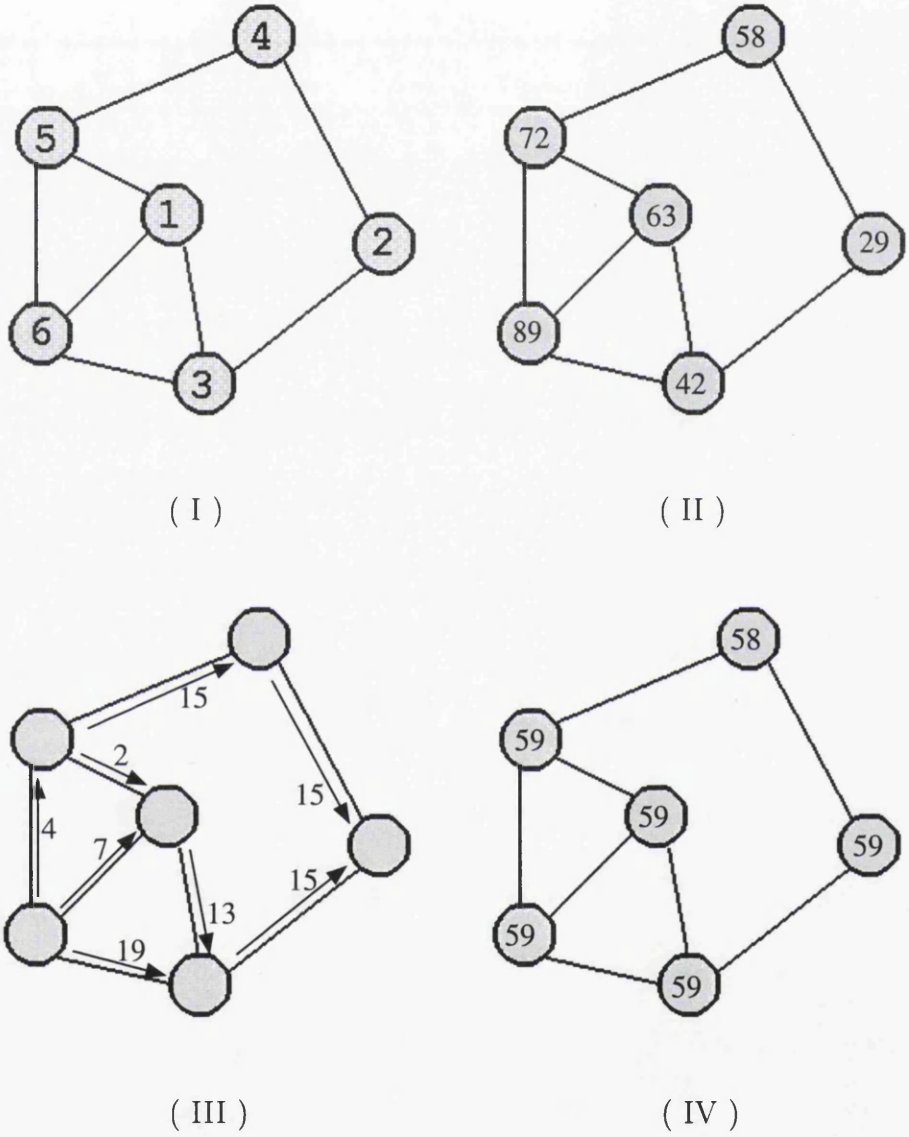


Figure 5.10: An example to demonstrate Hu & Blake algorithm, (I) The sub-domains graph, (II) The initial unbalanced load in the sub-domains, (III) Element to be moved among sub-domains based on the Hu & Blake result, (IV) The final balanced load in the sub-domains.

5.4.2 Implementation of Elements Migration Schedule

We enforce the obtained schedule in two distinct steps: First, a number of independent volume grids are constructed, where each one of them represents a 'sub-sub-domain' extracted from sub-domain i and to be attached to sub-domains j. Obviously, each sub-sub-domain should consist of the δ_{ij} elements recommended by the schedule. The second step is just simply to attach every sub-sub-domain to its destination (i.e. sub-domain j). The two steps together are described as an 'element migration cycle'. Procedures involved in completing one cycle are presented shortly, but first let us emphasise the overall strategy for enforcing the element migration schedule.

Representing the total number of elements in all sub-sub-domains that are to be extracted from a sub-domain i as:

$$\beta_i = \sum_{k=1}^{deg(i)} \psi_{ik} \tag{5.6}$$

Where as stated above ψ_{ik} is defined as:

$$\psi_{ik} = \begin{cases} 0, & \text{if } \delta_{ij} < 0 \\ \delta_{ij} & \text{if } \delta_{ij} > 0 \end{cases} \tag{5.7}$$

then a case where $(\beta_i \geq l_i)$ can be identified.

To be able to proceed in implementing the recommended schedule, whenever such a case occurs, another extra cycle has to be initiated. So, if the total number of elements in a sub-domain becomes less than what is needed to create another sub-sub-domain it is logged for the next cycle, while the current cycle carries on progressing. A flowchart that summaries the layout of the load balancing technique is presented in Figure 5.11, where the implementation strategy of the schedule is clear. Also, a case where two cycles were needed to achieve the balanced distribution is illustrated in Figure 5.12.

Although there is no clear restriction on the total number of cycles, apparently, *all* grids generated using the indirect decomposition method needed a single cycle only. Whilst cases with two cycles were observed among grids generated using the direct decomposition method.

Procedures within an elements migration cycle

Whenever a sub-sub-domain is to be extracted from sub-domain `I`, in order to be attached to `J`, a Greedy type algorithm is exploited twice. First, on the surface grid on the internal boundary shared between `I` and `J`, and secondly within the volume grid in `I`. A pseudo code which summarises the layout of the procedure used in extracting (`Nr_sub`) sub-sub-domains from sub-domain

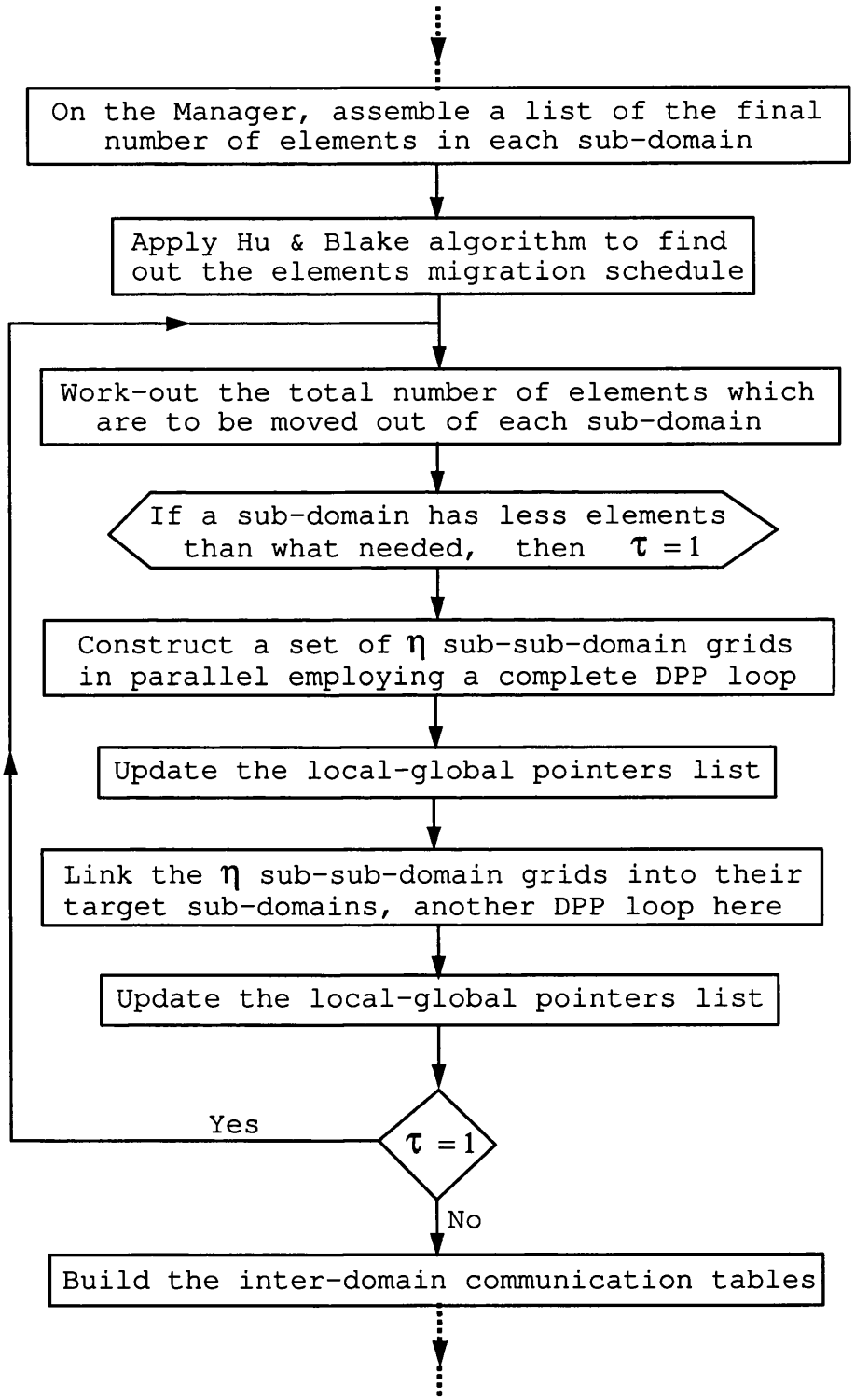


Figure 5.11: A flowchart for the strategy adopted to implement the elements migration schedule, notice the need for two DPP loops in each cycle.

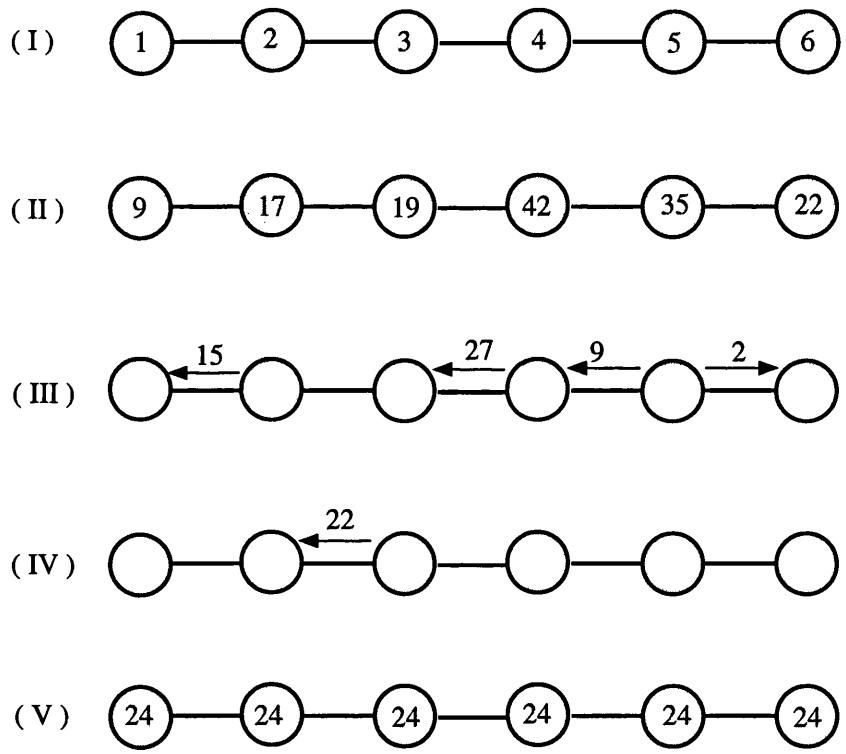


Figure 5.12: An example where more than one element migration cycle is needed, (I) sub-domains graph, (II) unbalanced element distribution, (III) first element migration cycle, (IV) second cycle and (V) the final 'balanced' element distribution.

I is presented here, whilst details about the Greedy type algorithm itself can be found in Section 3.2 (see page 40).

```
Ne(I) = Total number of tetrahedra in Sub-domain (I);
For(ir = 0; ir < Nr_sub; ++ir){

  (1) Find out about the required number of tetrahedra for the (ir)
      sub-sub-domain Ne(ir), and the destination sub-domain (J);

  (2) If Ne(ir) > Ne(I), then (I) is logged into the next cycle;
      (2-1) Break the loop;

  Ne_acc  = 0;
  (3) Starting from a face on the (I-J) internal boundary:

    (3-1) Assign (ir) to the tetrahedron associated with this
          boundary face;     ++Ne_acc;
    (3-2) Repeat (3-1) on the immediate adjacent tetrahedra only;
    (3-3) If Ne_acc >= Ne(ir) then do (5)
    (3-4) Using Greedy approach, progress over the surface grid on
          the (I-J) internal boundary; back to (3-1) for every face;

  (4) Considering the tetrahedra collected recently as a 'Front',
      then progress in the volume grid of (I) using Greedy approach;
      assign (ir) to tetrahedra that are still in (I);  ++Ne_acc;

    (4-1) If Ne_acc >= Ne(ir) then do (5); otherwise update the
          'Front' and carry on as in (4);

  (5) Update the current total number of elements Ne(I);
}

If(!ir){
  (6) Convert every set of tetrahedra associated with an (ir) into a
      stand alone volume grid (a grid with its own local numbering);
      Update the remaining tetrahedra in sub-domain (I);

  (7) Derive the new boundary for (I) and all its sub-sub-domains;

  (8) Update the local-global pointers list;
}
```

To be able to proceed in the migration cycle, the above procedure has to be

performed on all relavent sub-domains before the second step (attachment) is initiated. All new sub-sub-domains, and the original sub-domains, are considered as independent (i.e. local numbering systems) grids now. Having the local-global pointers list updated, the task of attaching `Nt_sub` sub-sub-domains to a sub-domain `J` can be defined as "building one new local system by merging `Nt_sub + 1` different local systems". However, the layout of such straightforward procedure can be summerised as:

```
For(it = 0; it < Nt_sub; ++it){

  (1) Establish an access to the global numbering for all,
      and only all, the boundary points on (J) and (it);

  (2) Find out both Ne(it) and the source sub-domain (I);

  (3) Loop over the Ne(it) tetrahedra;
      Attach every tetrahedron to (J) by:
      (3-1) ++Ne(J);
      (3-2) adjust connectivity of the new element exploiting the
            accesses to the global numbering system in (1);

  (4) Update the boundary grid for (J);

  (5) Update the local-global pointers list;
}
```

Parallel processing of the elements migration cycles

Two complete DPP loops are implemented within every element migration cycle, first loop is associated with the derivation of sub-sub-domains and the second loop with attaching them. DPP loop has been discussed in detail and presented in Section 4.2.1, however, information related to the implementation of DPP loop in the load balancing algorithm is presented herein.

Index in each DPP loop is controlled by the total number of sub-domains involved in the derivation (or attaching) procedure. The order of tasks processing in each DPP loop is selected based on the total number of elements due to be extracted from (or attached to) every sub-domain. The number of sub-sub-domains, as the total in the grid overall or per sub-domain, is not recognised at all in any of the loops. Because, if the index was to be controlled by the total number of sub-sub-domains in the grid instead, a Worker-Worker type of communication would have been required [4]. However, a typical parallel processing of a migration cycle (i.e. two DPP loops) is illustrated in both

[4]This is because a sub-domain may exist on two different Workers at the same time, and two different sub-sub-domains may have to be extracted/attached simultaneously.

Figure 5.13 and Table 5.3. Four Workers are employed to process 7 tasks in each DPP loop, and 19 sub-sub-domains are extracted and attached. More information about this grid can be seen in in Table 5.4, option (3).

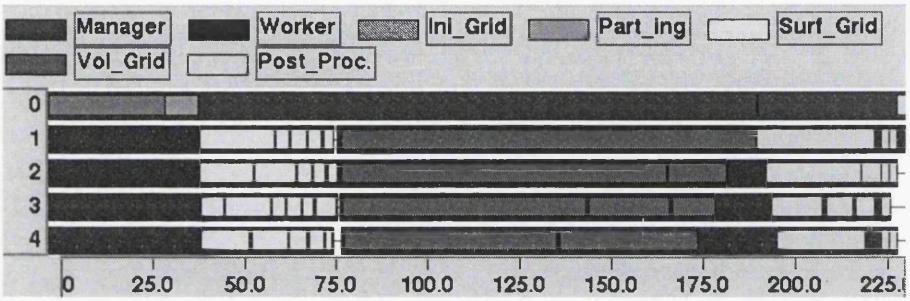


Figure 5.13: Dynamic Parallel Processing for the load balancing algorithm. Two DPP loops, with 7 tasks in each processed on 4 Workers, number of sub-sub-domains within each task is presented in Table 5.3.

Very little information is transferred via the Manager-Worker communication operations. The sub-sub-domains grid data are stored on the disc instead of the Manger's (or any other Worker's) memory. Of course, exchanging data through reading and writing to the disc is a very slow procedure in comparison to the direct use of MPI functions. But, on the other hand, overloading the Manager's memory would have introduced some limitations into the framework overall. May be, quoting from [40] is an appropriate conclusion: "In parallel programming, as in other engineering disciplines, the goal of the design process is not to optimise a single metric such as speed. Rather, a good design must optimise a problem-specific function of execution time, memory requirements, implementation costs, maintenance costs, and so on. Such design optimisation involves tradeoffs between simplicity, performance, portability, and other factors."

No Sub-domain:	1	2	3	4	5	6	7	8	Total-DPP
First step:	5	2	4	2	3	2	1	0	7
	W1	W4	W2	W2	W3	W3	W3	–	
Second step:	0	3	1	2	2	3	3	5	7
	–	W4	W2	W4	W1	W2	W1	W3	

Table 5.3: Total number of sub-sub-domains due to be extracted (First step) or linked (Second step) in every sub-domain, Worker associated with each Sub-domain is presented as W(?). The last column (Total-DPP) contains the total number of tasks (i.e. sub-domains) have been identified in each DPP loop.

5.4.3 Load Balancing and Inter-domain Communication

Usually, high quality partitioning of a grid is identified by: (a) well balance distribution of total workload among the sub-domains and (b) minimum inter-domain communication cost. Although the developed algorithm has demonstrated that condition (a) can be satisfied always, unfortunately, the situation for condition (b) is often far from optimal. Obviously, the poor performance in minimising the inter-domain communication cost is not due to the load balancing algorithm itself, the *initial interface* created at the domain decomposition step has a significant impact. Also, the use of uni-directional planar cuts is not the *ideal* approach in this respect, and it is highly likely that implementing a more advanced domain decomposition algorithm will change the *final* communication cost dramatically. In addition, a better balanced workload distribution at the domain decomposition stage means less elements migration will be needed, subsequently a better reservation of the *initial interface* is expected.

In order to improve the performance in respect to the inter-domain communication cost, a *smoothing* algorithm is applied on the 'new' sub-domain internal boundaries. The core idea of this algorithm has been illustrated earlier in a simple form of 2D triangular grid, see Figure 3.18 in page 63. Obviously, it is more complex in the 3D tetrahedral grids case, but we believe the concept is clear and there is no need for further discussion. However, the extended form of element connectivity (i.e. when the neighbouring elements are known) is the only data required. The procedure is carried out in parallel while the sub-sub-domains are created on the Workers. In order to demonstrate the impact of such smoothing technique on the inter-domain communication cost, a grid of an airfoil (M6, Dassault) is presented. The grid has been generated in 8 sub-domains using both indirect decomposition method (Figure 5.14 and 5.15) and direct decomposition method (Figure 5.16 and 5.17). Three different runs have been carried out where every run is associated with a different option, see Tables 5.4 and 5.5 for more details.

We recall the procedure adopted in extracting the sub-sub-domains to emphasise some points related to inter-domain communication cost. First of all, applying Greedy on the internal boundary faces to collect all adjacent tetrahedra before marching towards the interior ones is a vital point in this respect. Because otherwise there will be a risk of maintaining the *old* internal boundary and exposing new tetrahedra faces as an *additional* interface. Secondly, a Greedy type algorithm has been embraced mainly because of the interest in keeping the memory usage to a minimum. So, if this is not a concern at all, a more sophisticated grid partitioning technique can be integrated instead [6, 133, 75]. In fact, Greedy algorithm is well recognised to produce partitioned grids with expensive inter-domain communication cost [150]. Disjoint

	Option No.1		Option No.2		Option No.3	
No_Sub.	No_Tetr.	No_Com.	No_Tetr.	No_Com.	No_Tetr.	No_Com.
1	159523	5111	59755	4341	59697	3467
2	63092	4838	59757	8051	59847	7195
3	76208	5603	59757	8550	59697	7647
4	38898	3370	59757	5379	59766	4649
5	70874	6601	59755	7910	59854	6756
6	25378	1659	59757	5176	59769	4536
7	21117	1188	59756	5573	59754	4933
8	22959	1444	59755	4516	59665	4141
Total	478049	29814	478049	49496	478049	43324

Table 5.4: Number of tetrahedra (No_Tetr.) and inter-domain communication points (No_Com.) per sub-domain (No_Sub.) for a grid of the M6 (Dassault) airfoil geometry, using the indirect decomposition method: -Option No.1, without implementing the workload distribution developed algorithm. -Option No.2, implementing the developed algorithm without the smoothing technique applied on the new internal boundary. -Option No.3, the default option for using the workload distribution algorithm with the smoothing technique.

sub-domains are very likely to exist [31], though a comparison between information in Table 5.4 and Figure 5.15 shows that a disjoint sub-domain may still have a total number of communication points less than other continues ones. However, it has been observed throughout this research, that while disjoint sub-domains are very likely to appear in the indirect decomposition grids only few cases were reported among the direct decomposition grids. Nevertheless, the presented algorithm after all can be considered as a very reasonable load balancing solution since no extra computing resources is needed.

The local-global pointers list has been updated throughout the load balancing algorithm, and therefore, inter-domain communication tables for the 'new sub-domains' can be constructed by implementing the same technique presented in Section 5.2.2. In addition, all information required for building different types of communication tables, as discussed in the same Section, are extracted and available for each sub-domain.

5.5 Concluding Remarks

Three different issues that may appear in most of the geometrical partitioning algorithms have been identified, a *post processing* solution for each issue has been demonstrated. The first algorithm, which is the only 'compulsory' one, deals with the task of constructing inter-domain communication tables.

	Option No.1		Option No.2		Option No.3	
No_Sub.	No_Tetr.	No_Com.	No_Tetr.	No_Com.	No_Tetr.	No_Com.
1	9895	761	57093	1527	57036	1235
2	24655	1999	57092	6472	56583	4848
3	34849	2869	57093	8087	57526	6938
4	135486	4764	57093	12141	57455	10155
5	183349	4712	57092	3673	56951	2581
6	35407	2760	57093	7444	57229	6288
7	23722	1875	57092	5861	56892	4989
8	9378	694	57093	1597	57069	1292
Total	456741	20434	456741	46802	456741	38326

Table 5.5: The same information as in Table 5.4, while the direct decomposition method is used. Notice that total number of tetrahedra is different, though the same surface grid on the original boundary is used.

These tables identify all boundary points that coexist on more than one sub-domain and provide a direct link between their independent local numbering systems. The developed algorithm is a straightforward implementation of a typical search procedure on an advanced data structure of the sub-domains boundary points.

The other two algorithms, unlike the first one, are heavily effected by the complexity of the gridded geometry, and they involve more expensive computational work. However, it has been demonstrated that the second algorithm, which tries to improve the grid quality at the sub-domains interface, can be ignored completely in the light of the little improvements obtained. Nevertheless, inspecting the quality of generated grids showed that a satisfactory level has always been achieved. The last algorithm presented has been developed in order to ensure that a grid generated by the parallel framework can have its total number of elements redistributed evenly on sub-domains within the *same* environments. The algorithm demonstrates an excellent performance in regards to the elements distribution balance but, unfortunately, it can not secure the associated inter-domain communication cost. Obviously, implementing a more advanced technique for the initial domain decomposition in the future will have a significant impact on this issue. Meanwhile, if a user is keen to implement an 'off the shelf' grid partitioning software, he/she can take advantage of the available option for constructing one global grid, which is discussed next the Chapter. However, generating a well balanced grid with optimal communication cost without the need of any 'post processing' must be the ultimate target in the future.

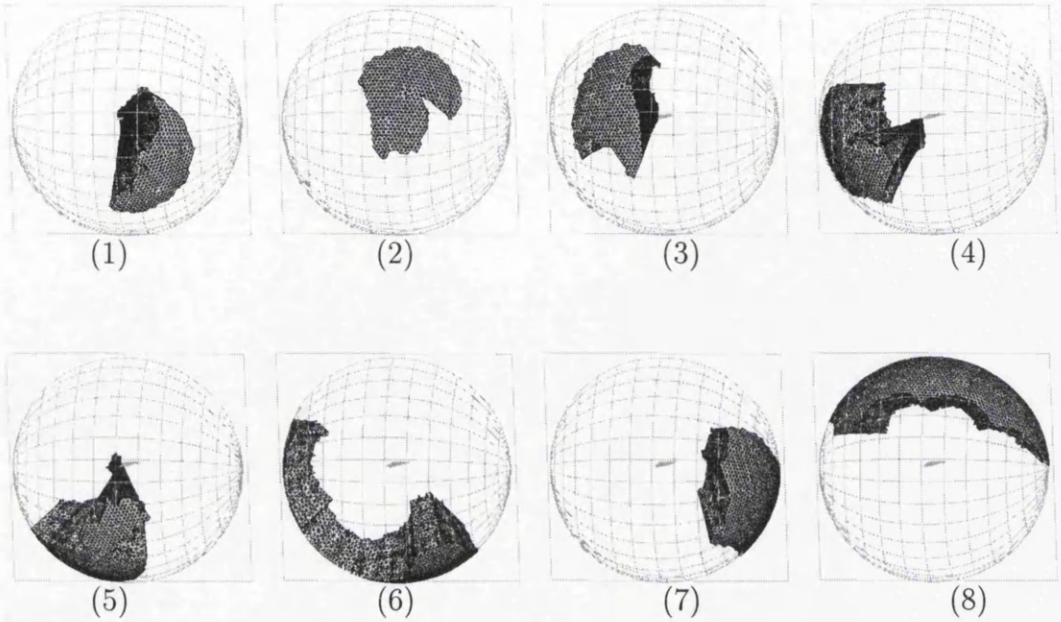


Figure 5.14: Sub-domains before applying the load redistribution algorithm.

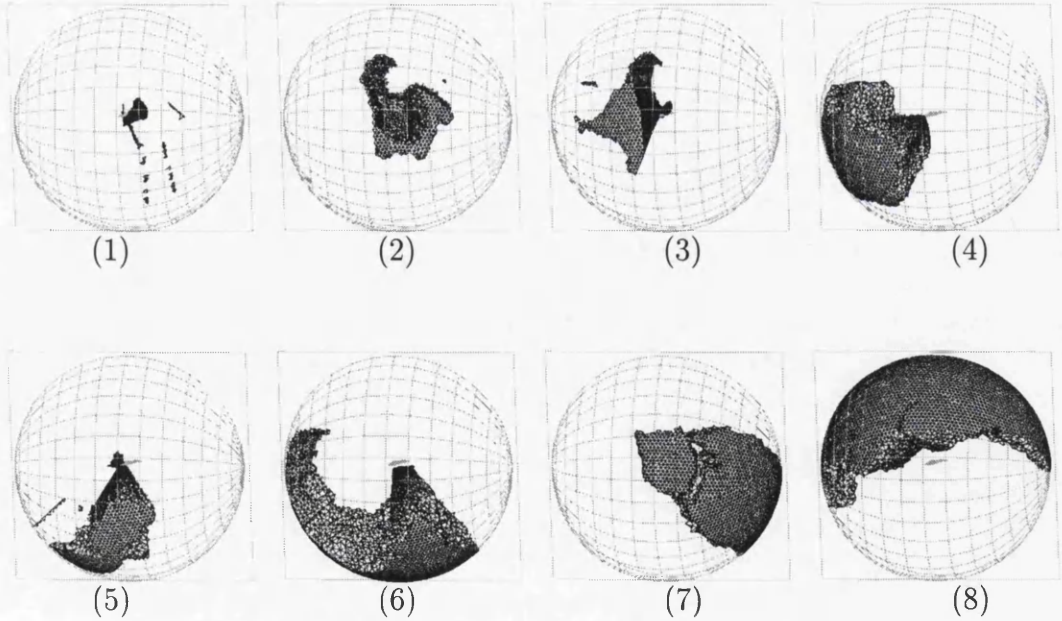


Figure 5.15: The sub-domains grid after applying the load redistribution algorithm.

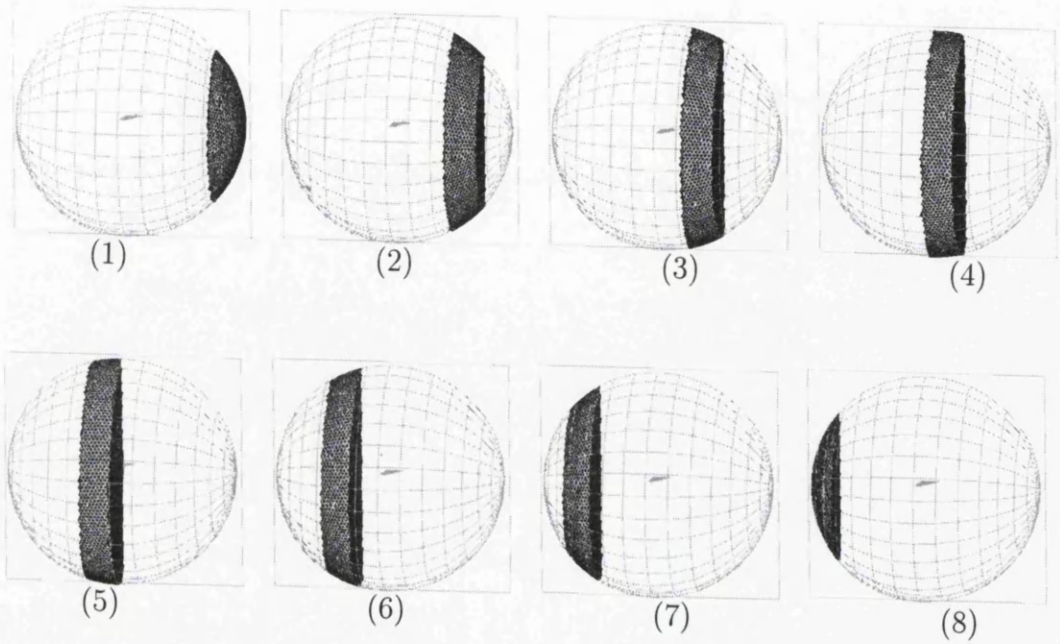


Figure 5.16: Sub-domains before applying the load redistribution algorithm.

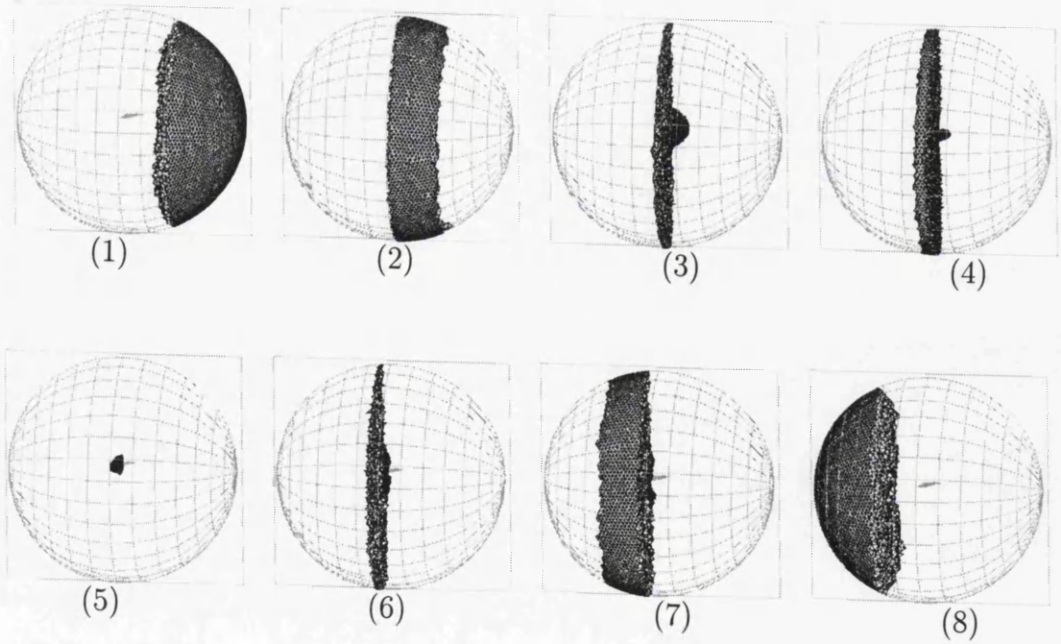


Figure 5.17: The sub-domains grid after applying the load redistribution algorithm.

Chapter 6

Results and Analysis

6.1 Introduction

This chapter focuses on the algorithms (programs) developed during this study as they are used in the 'real world'. The chapter starts by demonstrating the final shape of the computational framework with three different options available to serve various demands from the view point of the end user. Most of the algorithms have already been discussed with the exception of a rather simple, but very effective, procedure. It is the construction of one global volume grid by assembling all individual sub-domains grid. The utilisation of this procedure in carrying out local repartitioning of some sub- domains is then illustrated. Some examples of large size grids of realistic engineering problems, which have been generated on a medium size computer, are presented. In fact, the presented examples provide a complete coverage of all the various options available in the domain decomposition and the post-processing procedures.

A comprehensive study of the program performance and scalability is presented. In order to meet the dynamic nature of the framework, a specially designed policy has been developed. The number of processors, number of sub-domains and size of the grid are considered to be the parameters that can effect both performance and scalability. A unique performance visualisation tool is used[1], and all well known measurements (e.g. speedup factor and efficiency) are investigated. The scalability of the framework is inspected in respect to the problem size and the available computing resources. The impact of the sequential parts in the general algorithm and the inter-processor communications on the performance overall is inspected as well.

Integrating the framework into an in-house parallel computation engineering environment is highlighted. This environment contains parallel CFD and CEM

[1]For information about these tools and their implementation in the developed programs the reader is advised to consult Appendix B.

algorithms, together with some basic CAD functionalities and an advanced parallel visualisation tool. Employing some of the generated grids within this environment in order to carry out typical large scale CFD simulations is also demonstrated.

6.2 The Developed Framework in Practice

Having discussed all the algorithms thoroughly throughout the previous chapters, it is time to explore how they have been effectively integrated into one general program. The main interest behind this integration is to fulfill various requests that may emerge from using the program in different type of applications. As well as, to increase the program flexibility, such that the available computing resources are always exploited in an optimum manner. For instance, if a user has got a computer with sufficient memory storage, and he /she is interested in assembling the sub-domains grid in one global volume grid, then the program will provide an option to do so. In contrast, if a user has a very restricted access to computing memory, such that some of the obtained sub-domains are still bigger than what can be gridded by the sequential generator, the program will provide an option by which a *local* re-partitioning procedure is used. Furthermore, the framework has been constructed taking into consideration future developments, such that individual sub-algorithms can be enhanced or replaced with minimum efforts and modifications in the rest of framework. For example, If a different gridding technique was to be implemented on the internal boundary or (and) in the sub-domains, the current form of the framework including all the other sub-algorithms would be applicable almost without any modifications at all.

The three main options that have been made available in the program are briefly highlighted in the following, whilst a flowchart of the overall framework is presented in Figure 6.1.

- Option 1 (*basic grid*): In fact, this option represents the 'basic' grid that is always generated whatever option is chosen, though it may appear in a very different form in the final output. However, the option still stands as an independent option. Such an option can be used in the following two cases: (a) The workload is already well balanced, or it is not a major concern of the user. (b) The user is planning to use the generated grid in conjunction with another re-partitioning algorithm, e.g. algorithm presented in [134]. The internal boundary point relaxation algorithm is optional in here, while the establishment of the inter-domain communication tables obviously is a must.

- Option 2 (*load redistribution*): Although the developed algorithm may suffer sometimes from high cost in the inter-domain communication, it can still provide more than a 'reasonable' solution in too many cases. However, this option is recommended whenever there is a demand for a highly balanced distribution of the number of elements into sub-domains, and with less concern about achieving an optimal communication cost. In fact, this option may simply become the only available load balancing procedure, particularly when access to advanced computing resources is limited. Clearly, this option would be more effective if the initial workload distribution was relatively more acceptable such that number of elements to be shuffled is small, and thus impact on inter-domain communication is minimum.

  In the case where achieving a *perfect* load balancing is considered, regardless of the inter-domain communication cost, this option can be invoked while the smoothing procedure on the sub-sub-domains internal boundary is deactivated, see section 5.3.2. Similarly as in option 1, establishing the inter-domain communication is a must. Obviously, the communication tables are indeed constructed for the 'new' boundary, i.e. after the load redistribution algorithm is complete.

- Option 3 (*one global grid*): Clearly, it is extremely unlikely to have such large size grids assembled in one global volume grid in order to use it with a traditional sequential program. However, this option was introduced mainly to give the user an opportunity to use his/her own way in partitioning the global grid if they wanted to. Thus, if the number of sub-domains or communication cost obtained from using the previous options were not satisfactory, or if there was an issue with the compatibility in data formats, the user can simply convert the 'partitioned grid' into one global grid. Then, a typical grid partitioning algorithm, e.g. Metis [75], can be implemented directly. Neither of the internal boundary point relaxation or the inter-domain communication tables algorithms are used in this option.

The construction of global volume grid

The developed technique is outlined briefly below, while an illustrated example is presented in Figures 6.2 and 6.3. The former figure shows the geometry in (I) (i.e. a CFD model of the Super Sonic Car Thrust inside a box) with the position of the planar cuts in (II). The original boundary surface grid on the resulted sub-domains are also presented in (III), whilst the later figure shows the surface grid on the internal boundary (I) with some details from the global

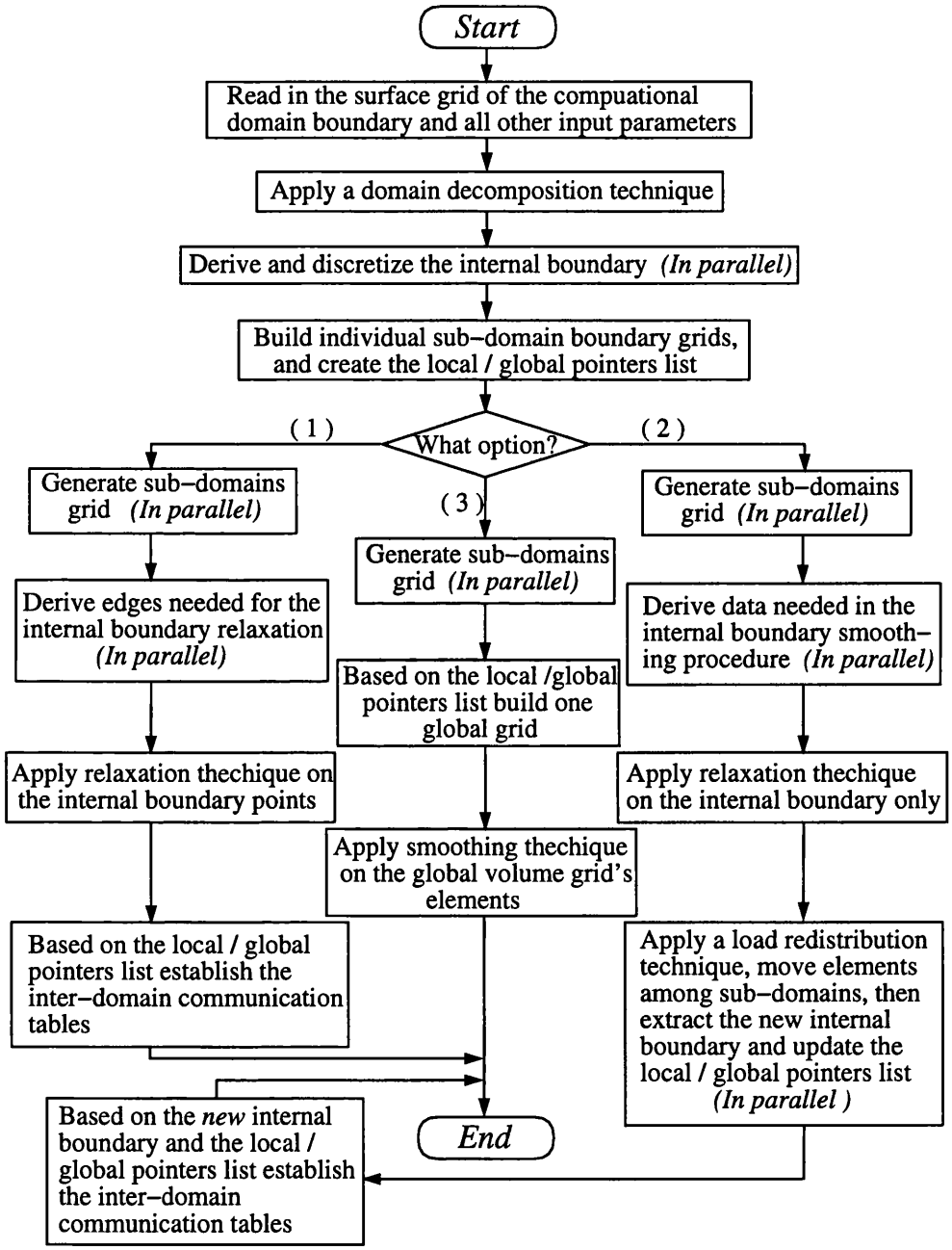


Figure 6.1: The general framework in practice, three different choices are available for the user: (1) generate the sub-domains grid and establish the inter-domain communication tables, (2) apply load redistribution on the sub-domains grid and (3) assemble the sub-domains grid in one global volume grid.

volume grid (II). Impact of the smoothing technique [2] can be seen as well, see *(II)* in both Figures 6.2 and 6.3 and notice the locations of internal boundary with the effect of the smoothing particularly in the coarse part of the grid.

Typical grid quality enhancements can still be implemented on the global grid elements [58], bearing in mind that an extensive demand of memory storage will be inevitable. In fact, although a point smoothing algorithm is available, as a post processing step in the assembly procedure, it is strongly recommended not to use it with large size grids. Very little improvement has been reported on the overall quality in global grids. Also it is to be noticed, that the original boundary grid is always reserved, though the order of the faces may change dramatically after the new global numbering system is established.

Layout of the procedure for constructing the global grid

```
1- Initialise the global volume grid with number of:

    Faces (NF_global)      = 0
    Tetrahedra (NT_global) = 0
    Points (NP_global)     = Total number of points on
                             internal and original boundary.

2- Loop over the sub-domains:

  2-1 Read a sub-domain volume grid with number of:
      Points (NP_Sub), Faces (NF_Sub), Tetrahedra(NT_Sub).

  2-2 Establish an access to the local-global pointers
      list with respect to the current sub-domain only.

  2-3 Initialise a temporary numbering list for (NP_Sub).

  2-4 Loop over the sub-domain points (NP_Sub)

     - If it is a boundary point, then access the pointers
       list using 2-2, else add it to (NP_global) as a new
       point and establish a link with the list in 2-3;
       (NP_global++;).
```

[2]Please note that the smoothing does not mean the relaxation of internal boundary points as presented in section 5.3.2, however, the same Laplacian equation, see 3.6 in page 50, is employed but operating on the entire volume grid points simultaneously.

```
    2-5 Loop over the sub-domain faces (NF_Sub)

       - If it is an original boundary face, then update the
         number of associated tetrahedron by adding the
         current value of (NT_global); otherwise ignore it.
         (NF_global++;).

    2-6 Loop over the sub-domain tetrahedra (NT_Sub)

       - Add a new tetrahedron to the global volume grid.
         All nodes are renumbered based on the local-global
         pointers list for the boundary points or on the
         numbering list established in (2-3) otherwise.
         (NT_global++);

3- Apply a smoothing technique on the global volume grid (optional).

4- Write out the final grid in one serial file.
```

It is been stated earlier that constructing the global volume grid option was introduced to be used only when a sufficient amount of memory is available. In fact, the assembly technique itself requires a very limited amount of extra memory, i.e. an integer vector of a length equals to the total number of points in the sub-domain only. Whilst, the 'real demand' of memory is associated with the storage of the global grid data itself. However, the same procedure as outlined above could still carry out the same task without the need for more memory by introducing a little modification. Simply, the updated version of every sub-domain grid (i.e. after all the sub-tasks in step 2 have been performed) can be written back on the disc instead of being accumulated to the global grid. Thus, having all sub-domains grid numbered according to the global volume grid numbering system and stored in individual files, the global grid can be created by simply reading individual sub-domain files and then appending them into one sequential file.

6.2.1 Local Partitioning of Large Sub-domains

Having the direct decomposition method implemented using planar cuts in one direction only, some limitations on the total number of sub-domains that can be created is expected. Also, recalling all the difficulties in controlling the load distribution between the sub-domains at the partitioning procedure in the indirect decomposition method. It is clear then, that there is a possibility to have a sub-domain assigned to a Worker to be gridded whilst it still

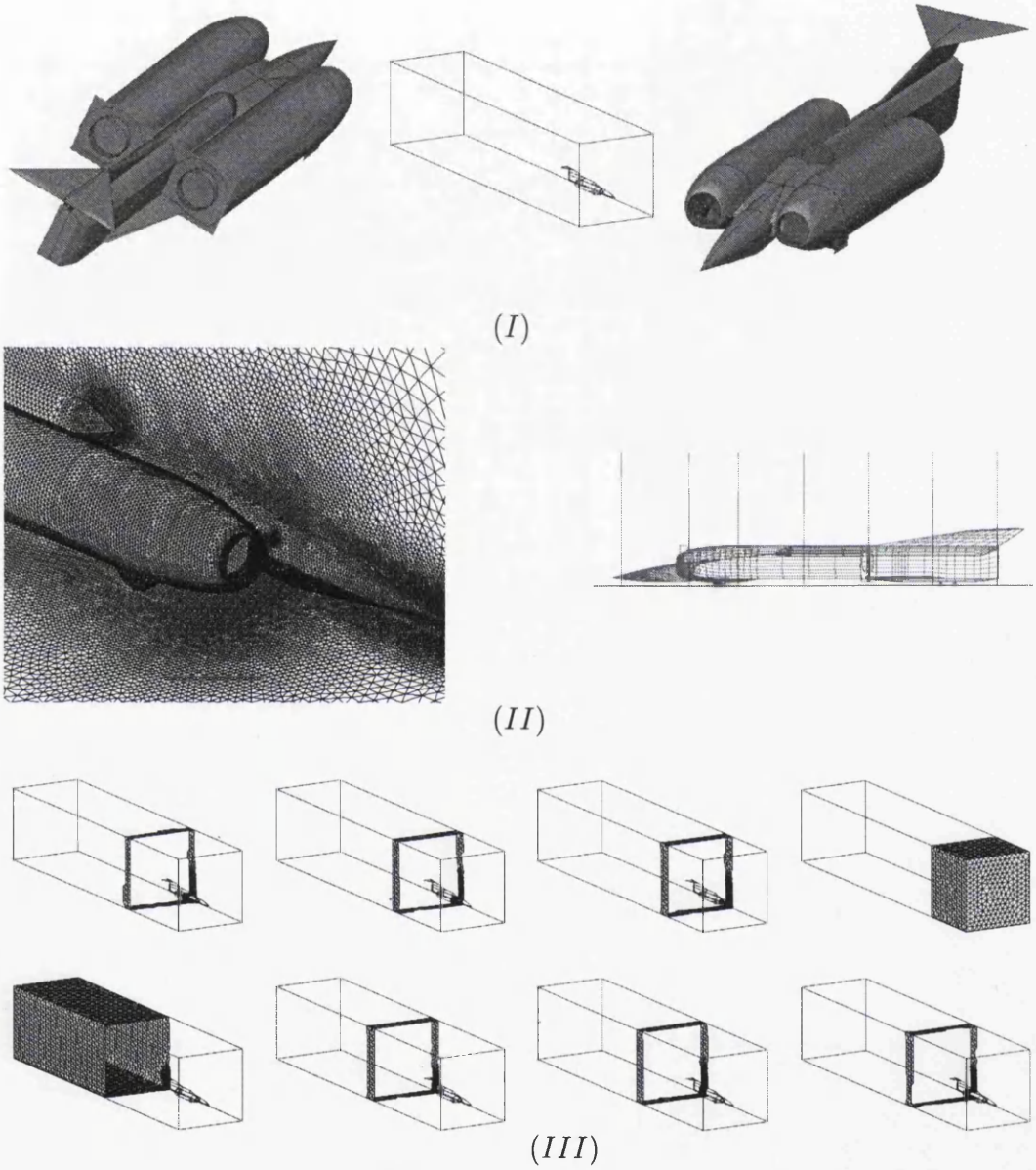


Figure 6.2: Super-sonic car example, a volume grid is generated in parallel within eight sub-domains then one global grid is constructed by merging the sub-domains grid. (I) Geometry definition, (II) Triangular grid on original boundary and positions of the planner cuts in the domain decomposition procedure. (III) Original boundary on the sub-domains.

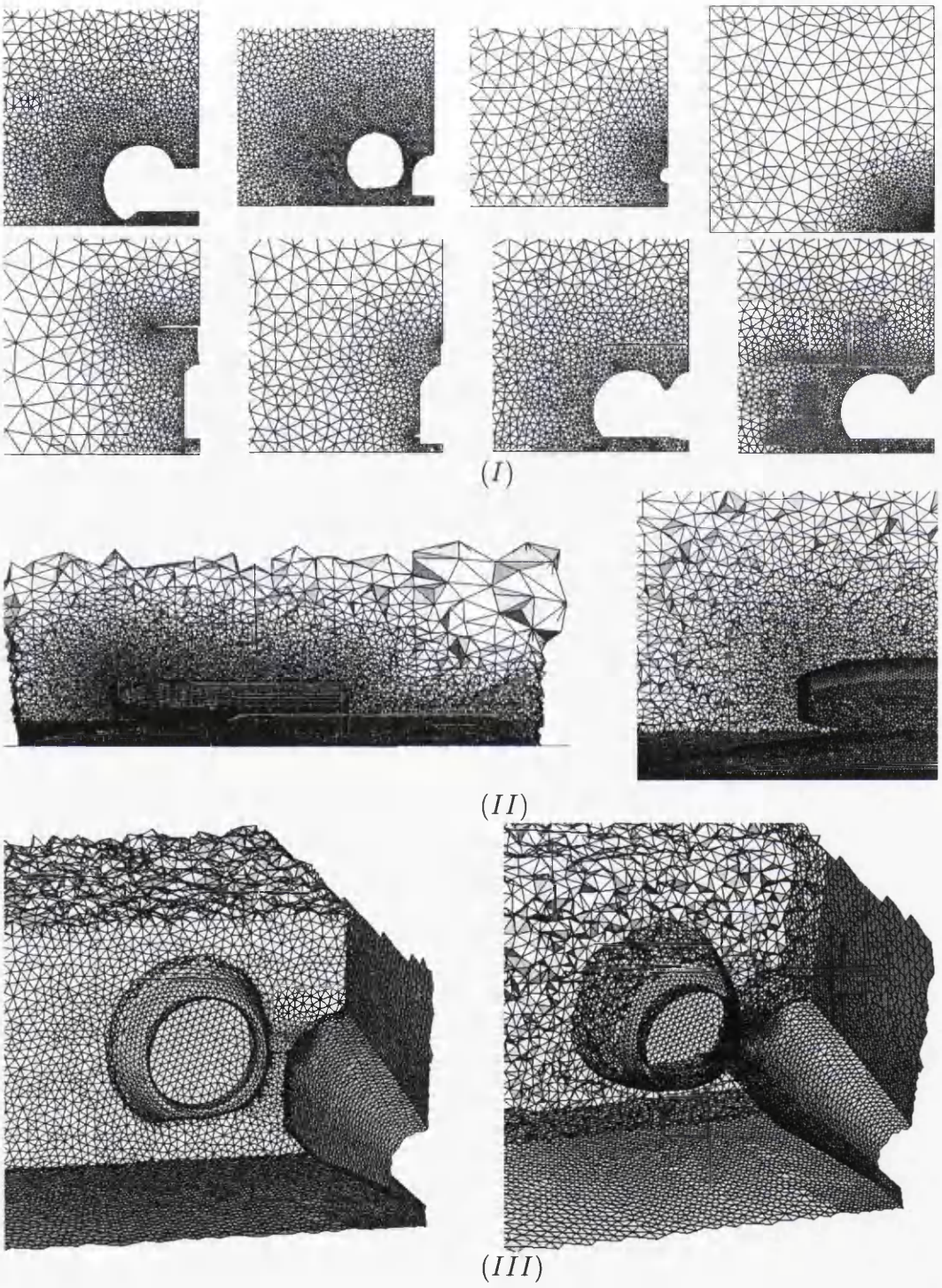


Figure 6.3: Super-sonic car example (cont.). (I) Details from the surface grids on the internal boundaries, (II) Cross section in the global volume grid where the location of internal boundary, and the effect of global grid smoothing on it, is noticeable. (III) Cross section around an interface region as an individual sub-domain and in the global grid.

requires computing memory larger than what is available. However, it has been observed that the algorithm presented above (i.e. the construction of the global grid) can be utilised effectively to overcome the problem of 'large' sub-domains. The overall algorithm can be enhanced by a local partitioning technique, which is an identical copy of the procedure in the global algorithm but employed on one sub-domain locally. The large sub-domain is divided into a number of smaller sub-sub-domains, which are then gridded independently on the same worker and assembled to form the sub- domain grid. The sub-sub-domains are recognised locally only and they simply do not exist at all from the point of view of the overall framework. Although, such local partitioning can be implemented in a recursive manner, ideally, the issue should be treated earlier at the initial domain decomposition step.

The local partitioning technique has been implemented in both indirect and direct decomposition methods. Table 6.1 illustrates the details of a grid generated using the former method [107]. The grid is for a configuration of two centralised hemispheres with different radii. The surface grid on the original boundary consists of 560,578 faces and 280,291 points. The domain is partitioned into 64 sub-domains initially and a number of sub-domains has been repartitioned locally, e.g. 55–57 in the Table.

Whilst employing the local partitioning in the indirect decomposition method can be very expensive and unreliable procedure, it has proved to be highly effective in the direct decomposition method case. Obviously, the main reason behind this is the fundamental difference in the domain decomposition technique itself. The complexity and irregularity of the sub-domain boundaries in the former method may increase the cost of the partitioning procedure dramatically. In fact, the boundary shape in the sub-sub-domains is expected to be even worse than the mother sub-domain, this will certainly amplify the risk of recovering the boundary faces after the Delaunay triangulation. However, with a better surface grid quality and more regular shape of the internal boundary in the direct decomposition case, the procedure has proved to be truly beneficial. A number of real world examples have been gridded utilising this procedure, an illustration example is presented.

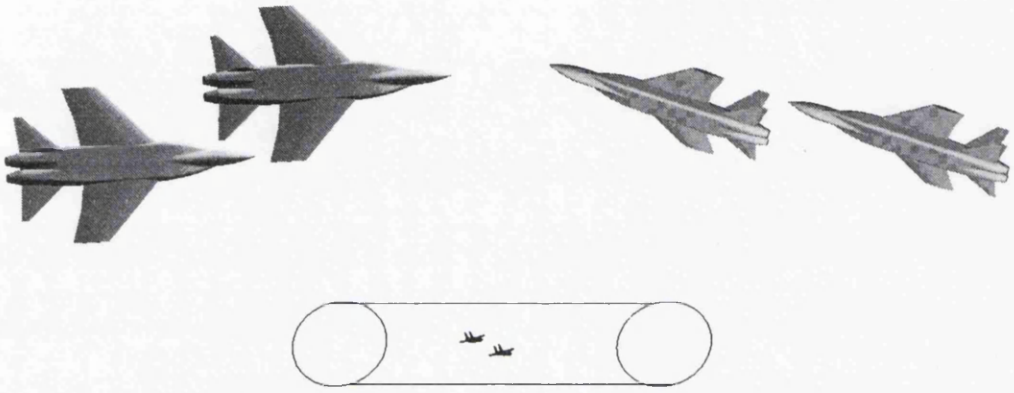
A configuration of two military aircraft inside a wind-tunnel has been divided into 8 sub-domains along the major axis of the tunnel, using the option of equal number of faces per sub-domain (see Figure 6.4). Subsequently, six sub-domains out of the original eight have been partitioned locally, each into two sub-sub-domains (see Table 6.2). Demonstration of the local partitioning applied on two sub-domains (i.e. 3 and 4) can be seen in Figures 6.5 and 6.6.

Domain	Elements.	Points	Faces
1	1,016,219	166,341	37,004
2	964,506	161,074	47,454
3	951,884	158,456	44,820
4	470,788	80,174	28,818
5	501,641	85,334	30,524
6	606,094	101,107	29,336
7	492,939	83,149	27,144
8	503,835	85,374	29,166
9	742,530	123,670	35,064
10	927,866	153,403	39,584
11	938,348	157,143	47,850
12	922,991	153,679	43,582
13	515,704	90,063	40,402
14	1,000,152	165,478	43,302
15	554,383	96,396	41,770
16	522,476	88,880	31,476
17	505,184	85,239	28,038
18	466,260	78,850	26,748
19	994,438	162,298	34,520
20	870,901	149,595	58,798
21	866,503	144,856	43,078
22	856,781	143,863	45,204
23	867,437	144,830	42,506
24	840,716	140,483	41,438
25	795,498	133,333	40,810
26	809,576	135,834	42,206
27	754,146	129,180	49,324
28	1,092,478	182,233	53,150
29	924,931	153,792	42,930
30-1	921,695	154,475	47,296
30-2	263,612	45,473	18,254
31	910,392	152,191	45,350
32	684,541	116,329	41,000
33	528,928	90,391	33,852
34	964,140	162,495	53,144
35	839,830	140,438	41,948

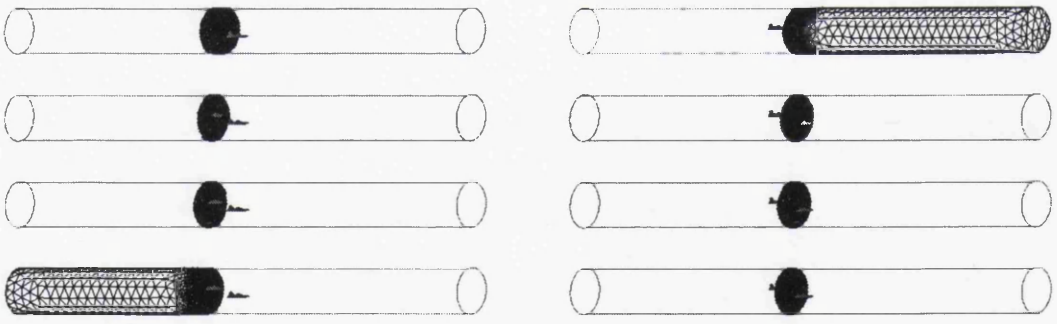
Domain	Elements.	Points	Faces
36	861,786	144,422	44,412
37	870,543	146,309	46,410
38	841,194	140,940	43,098
39	860,053	148,858	62,590
40	848,093	142,021	43,124
41	798,008	136,032	50,030
42	573,256	101,313	48,882
43	546,969	92,577	31,398
44	587,680	102,745	45,864
45	653,728	109,737	34,244
46	477,376	80,609	26,818
47	513,302	87,262	30,862
48	561,333	95,197	32,988
49	466,043	79,084	27,622
50	586,159	100,132	37,236
51	547,594	94,662	39,170
52	1,146,291	189,306	48,040
53	960,420	159,131	42,210
54	1,000,691	166,145	45,108
55-1	1,051,133	174,994	49,474
55-2	270,816	46,683	18,796
56-1	1,036,076	173,845	53,588
56-2	201,575	34,590	13,088
56-3	231,795	40,449	17,934
57-1	939,339	157,283	47,566
57-2	140,110	24,871	12,236
57-3	10,113	2,149	2,152
57-4	452,932	82,200	47,364
58	840,521	142,130	48,258
59	536,844	92,764	38,200
60	459,583	77,131	23,714
61	525,858	91,070	37,948
62	606,006	102,623	35,074
63	586,693	99,081	32,688
64	188,625	33,824	18,102

Total:	Elements	Points	Faces
	49,168,881	8,288,068	2,703,17

Table 6.1: Details of a grid of two hemispheres of different radii, generated using the indirect decomposition method. Notice that the local repartitioning used in a few of the sub-domains. The original boundary surface grid consists of 560,578 faces and 280,291.

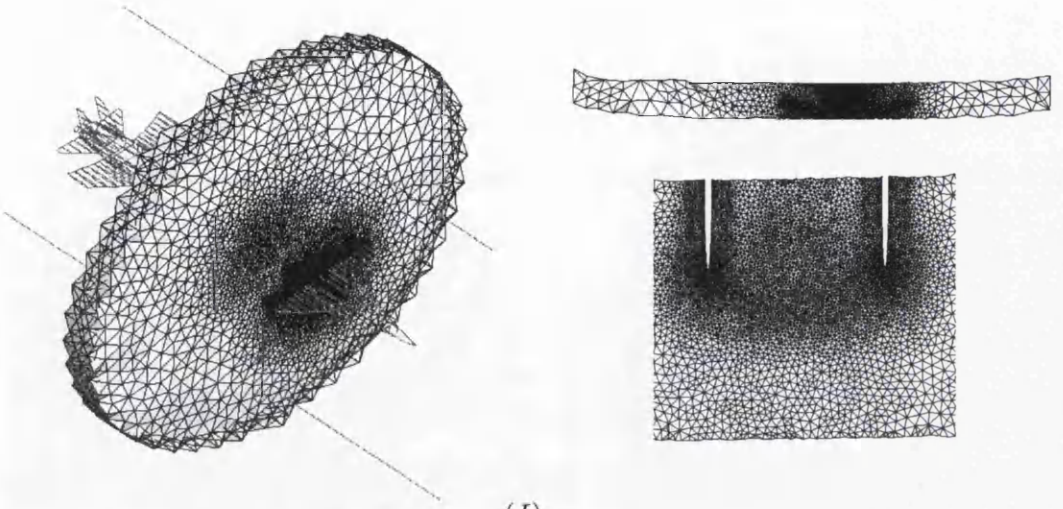


(I)

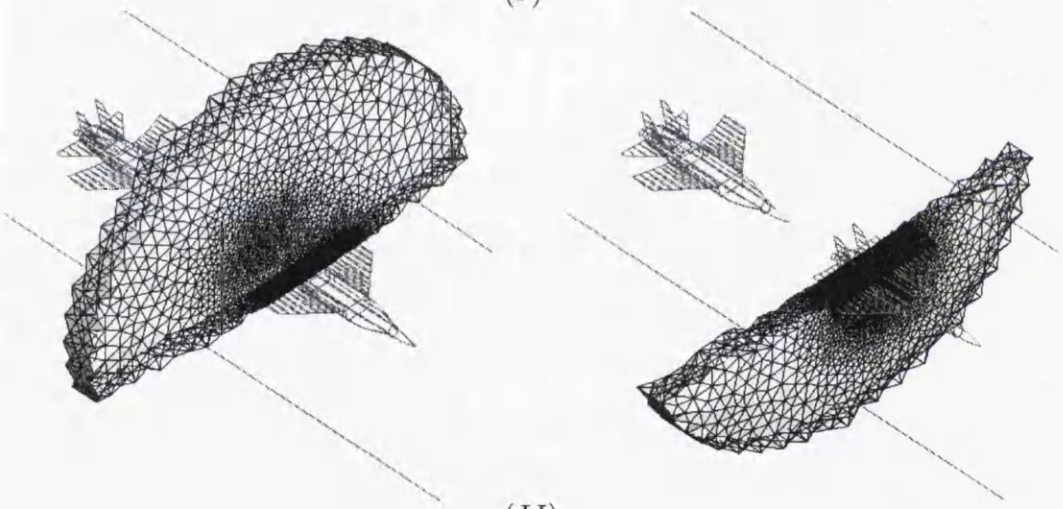


(II)

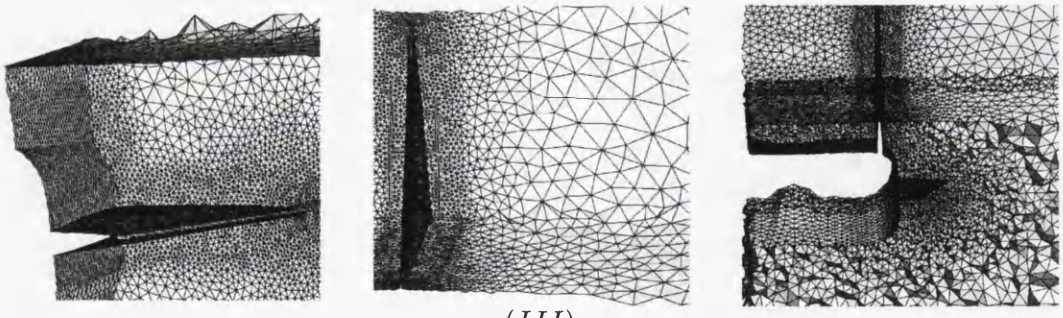
Figure 6.4: Two aircraft in a wind-tunnel example. One global grid is constructed by splitting the domains into eight sub-domains along X axis followed by 'local partitioning' for each domain using planar cuts along a different axis. (I) Different views of the geometry. (II) The boundary grids on the initial eight sub-domains.



(*I*)

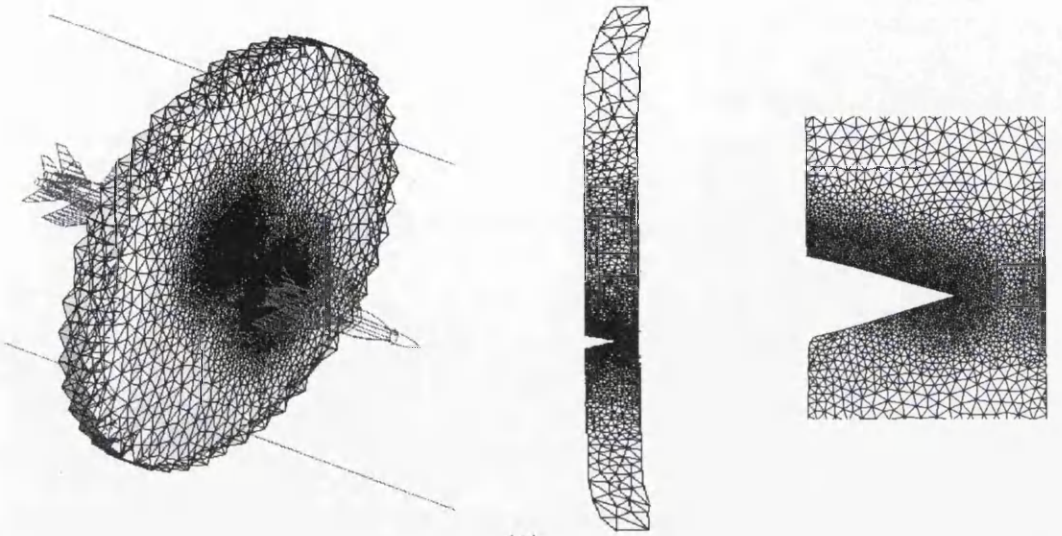


(*II*)

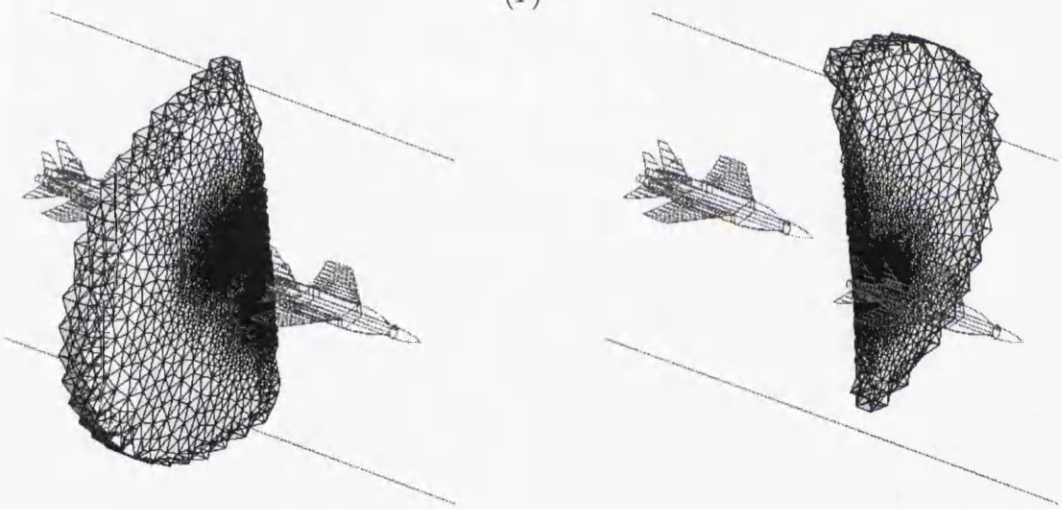


(*III*)

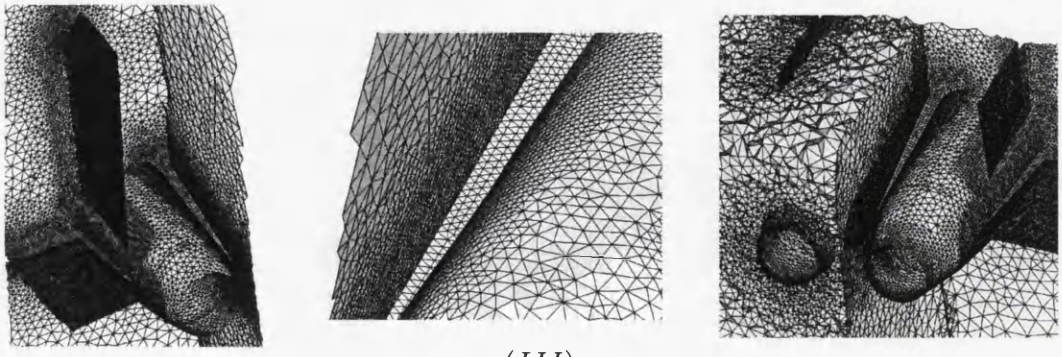
Figure 6.5: Two aircraft in a wind-tunnel example (cont.). (*I*) Surface grid on the closed boundary of Sub-domain 3, and the new local internal boundary with a close-up. (*II*) The closed boundary surface grid on the 'new local sub-domains'. (*III*) Details from the volume and surface grids.



(I)



(II)



(III)

Figure 6.6: Two aircraft in a wind-tunnel example (cont 2.). Similarly to previous Figure, details for Sub-domain 4 while the local repartitioning axis in here is Y (it is Z in the previous Figure and X the initial partitioning).

Domain	Elements.	Points	Faces
1-1	1436393	234718	69498
1-2	1403451	221597	68578
2	1500853	263163	120090
3-1	2134879	360099	117930
3-2	648709	110935	41954
4-1	1215588	205272	68200
4-2	1472833	248288	81356
5-1	1487321	260933	76668
5-2	1419587	258081	69876
6	1479156	259329	118636
7-1	598747	105080	33016
7-2	2200145	372906	106278
8-1	1989245	337388	65614
8-2	1884852	319683	64852

Total:	Elements	Points	Faces
	20,871,759	3,557,472	1,102,546

Table 6.2: Details of the two aircraft in a wind tunnel grid. The original boundary surface grid consists of 658,198 faces and 329,105.

6.2.2 Examples of Grids

Three grids of different realistic engineering problems have been selected in a manner that demonstrates the various options available in the framework. This involves, in addition to the post processing options that were highlighted earlier in this Chapter, the three different possibilities for implementing the domain decomposition procedure mentioned in Section 3.3.2 (i.e. equal number of faces per sub-domain, even intervals between the planar cuts and prescribed position for the planar cuts). The direct decomposition method is the only approach considered in here. Employing some of these grids by some parallel simulators is presented later on, see Section 6.4.

All grids have been generated on an SGI Challenge shared memory machine which has 8 (150 MHZ IP19, MIPS R4000) processors and 512 MBytes of memory. One processor only has been used in all the examples presented.

Duct of an aircraft engine (*The basic grid option*)

The aerospace industry over the last two decades has shown a great interest in the simulation of electro-magnetic wave propagation and scattering. Major modifications in the design of aircraft external shape, particularly in the military sector, have been introduced based on such simulations. However, investigating the propagation of single electro-magnetic wave through a par-

ticular part of the aircraft can be of interest. The configuration presented in Figure 6.7 is for an aircraft engine duct, which has been a subject of an extensive CEM study.

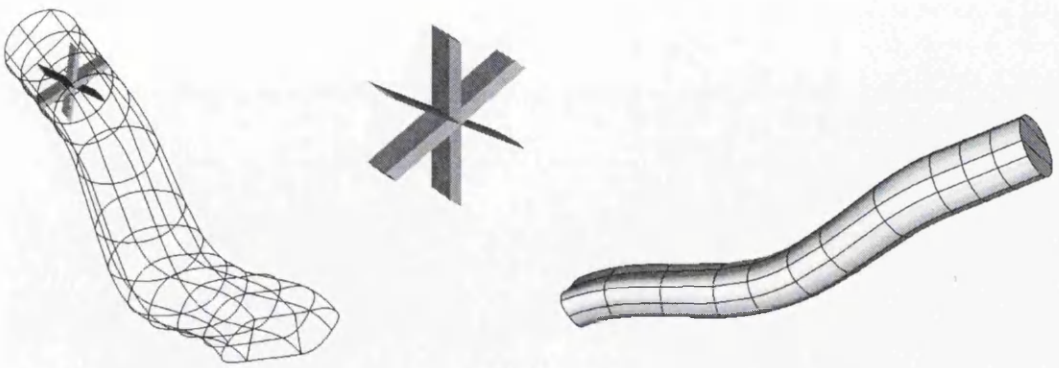
In a typical CEM simulation, unlike CFD, grids are normally very regular and may have fixed element size over the entire domain, including the scatter surface. The point distribution (element size) is determined based on the length of the wave (frequency), it is agreed that to achieve a good level of accuracy ten nodes at least are needed per wavelength. Nevertheless, having this uniform grid point density, and after inspecting the general shape of the geometry as presented in (I) in Figure 6.7, it has been decided to use the 'even intervals' option for partitioning the domain. Apparently, this option can still produce a very reasonable workload distribution in special circumstances such as the ones in this example.

A surface grid consists of 2,110,122 triangles and 1,055,065 points has been generated using a typical Advancing Front algorithm [99, 100]. The domain then has been partitioned into 64 sub-domains using 63 planar cuts spaced equally along the major axis of the duct. Typical examples of an internal boundary grid can be seen in (II) in Figure 6.7.

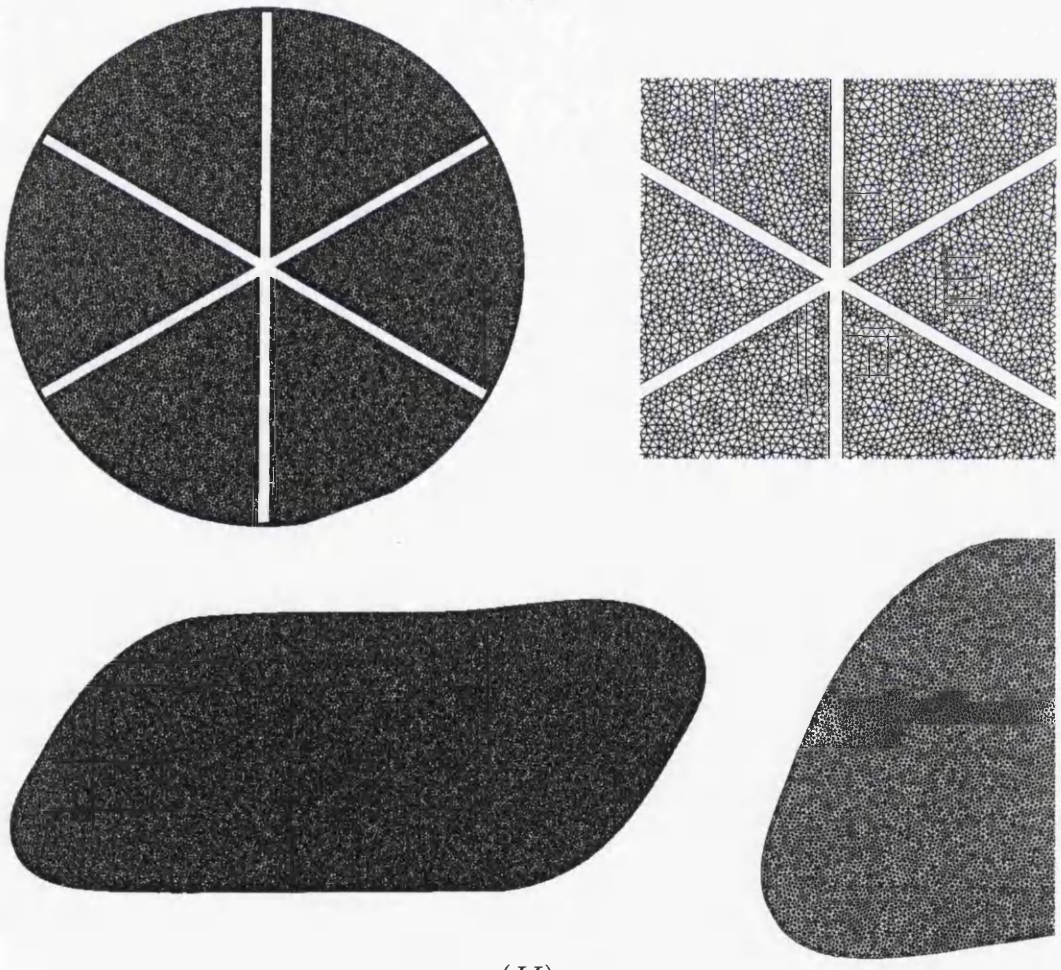
Inspecting the details of the sub-domains grid presented in Table 6.3 shows that a satisfactory level of load distribution is already available. In fact, though employing the load redistribution algorithm on such a grid will improve the load balancing more, it is recommended not to do so since very likely the cost of the inter-domain communication will increase. Such that, after all, the total time of idle processors might be very similar when the grid is been used by a parallel solver. However, a list of number of inter-domain communication points of this grid is presented in Table 6.4.

An example of a two adjacent sub-domains grid is presented in Figure 6.8, see Figure 6.42 for a general view of the entire domain grid. The total time required to generate the grid has been around 18 hours, whilst 90 minutes of this was spent on the partitioning procedure (including the surface grid generation on the internal boundary). Establishing the inter-domain communication tables and writing the relevant files was accomplished within less than 3 minutes.

Unfortunately, due to some out of hand administrative issues, it was not possible to access a parallel CEM solver that can operate on such grid size. However, some smaller grids with a very similar load balancing have been employed in conjunction with an in-house CEM program during the course of this research [94, 95].



(I)



(II)

Figure 6.7: Duct example. (I) Geometrical definition. (II) Details of two selected internal boundaries.

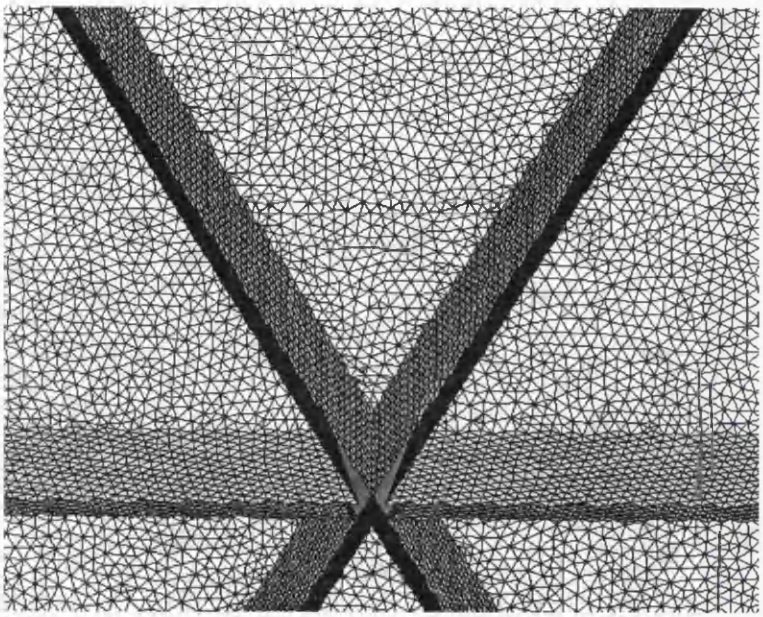
Domain	Elements.	Points	Faces	Domain	Elements.	Points	Faces
57	1928438	343554	174540	32	1742423	303501	132120
58	1950259	344974	168574	27	1735377	302306	131922
56	1982127	349026	165864	29	1760776	306157	131744
12	1972741	343571	149174	28	1727209	301023	131742
11	1956462	340950	149044	33	1730550	301590	131590
14	1974817	343627	148762	35	1743161	303395	131326
13	1967630	342608	148750	36	1732332	301661	131140
10	1943188	338825	148656	42	1749437	304242	131084
15	1952252	340108	148336	41	1750914	304486	130894
16	1937983	337838	147434	34	1733787	301880	130822
9	1936328	337301	147010	37	1738218	302506	130642
17	1949406	339338	146934	53	1750124	304297	130592
8	1918245	334132	145374	54	1753221	304805	130544
18	1925140	335027	144858	40	1749280	304126	130420
7	1884055	328376	143712	52	1749938	304227	130354
19	1880554	327590	142476	59	1762642	306166	130334
6	1861501	324300	141478	39	1740435	302622	130240
20	1862233	324266	141042	38	1732721	301487	130208
21	1848974	321941	139676	43	1724207	300045	130162
1	1905619	330510	139468	55	1937434	333007	129986
5	1836540	319949	139410	60	1753690	304696	129896
22	1832139	319058	138576	61	1758664	305275	129730
4	1793237	312664	137264	62	1745970	303343	129502
3	1779769	310359	136040	47	1734484	301528	129384
23	1810438	315090	136034	63	1704912	296943	129034
2	1766061	307911	134890	46	1729951	300708	128780
24	1784761	310721	134662	51	1705734	296785	128622
25	1766522	307656	133640	44	1711917	297795	128276
64	1849814	320157	132658	48	1707466	297007	128126
26	1753328	305373	132470	45	1703574	296471	127922
30	1753086	305214	132360	49	1692086	294411	127212
31	1764316	306836	132310	50	1692555	294421	127138

Total:	Elements	Points	Faces
	115,713,152	20,151,762	8,762,964

Table 6.3: Details of the Duct example. 64 Sub-domains using the Direct Decomposition method along the longest axis of the Duct. The original boundary surface grid consists of 2,110,122 faces and 1,055,065 points.

Sub-domains	Comm. Points	Sub-domains	Comm. Points
$1 \leftrightarrow 2$	25964	$32 \leftrightarrow 33$	26258
$2 \leftrightarrow 3$	26223	$33 \leftrightarrow 34$	26038
$3 \leftrightarrow 4$	26511	$34 \leftrightarrow 35$	26011
$4 \leftrightarrow 5$	26764	$35 \leftrightarrow 36$	26194
$5 \leftrightarrow 6$	27346	$36 \leftrightarrow 37$	26042
$6 \leftrightarrow 7$	27800	$37 \leftrightarrow 38$	25966
$7 \leftrightarrow 8$	28203	$38 \leftrightarrow 39$	25854
$8 \leftrightarrow 9$	28514	$39 \leftrightarrow 40$	25955
$9 \leftrightarrow 10$	28973	$40 \leftrightarrow 41$	26070
$10 \leftrightarrow 11$	29227	$41 \leftrightarrow 42$	26200
$11 \leftrightarrow 12$	29296	$42 \leftrightarrow 43$	26202
$12 \leftrightarrow 13$	29293	$43 \leftrightarrow 44$	25755
$13 \leftrightarrow 14$	29176	$44 \leftrightarrow 45$	25408
$14 \leftrightarrow 15$	29384	$45 \leftrightarrow 46$	25619
$15 \leftrightarrow 16$	29065	$46 \leftrightarrow 47$	25963
$16 \leftrightarrow 17$	29104	$47 \leftrightarrow 48$	25926
$17 \leftrightarrow 18$	29006	$48 \leftrightarrow 49$	25309
$18 \leftrightarrow 19$	28280	$49 \leftrightarrow 50$	25406
$19 \leftrightarrow 20$	28128	$50 \leftrightarrow 51$	25282
$20 \leftrightarrow 21$	27819	$51 \leftrightarrow 52$	26148
$21 \leftrightarrow 22$	27576	$52 \leftrightarrow 53$	26143
$22 \leftrightarrow 23$	27316	$53 \leftrightarrow 54$	26109
$23 \leftrightarrow 24$	26801	$54 \leftrightarrow 55$	26172
$24 \leftrightarrow 25$	26720	$55 \leftrightarrow 56$	25831
$25 \leftrightarrow 26$	26410	$56 \leftrightarrow 57$	24827
$26 \leftrightarrow 27$	26211	$57 \leftrightarrow 58$	25825
$27 \leftrightarrow 28$	26188	$58 \leftrightarrow 59$	25935
$28 \leftrightarrow 29$	26110	$59 \leftrightarrow 60$	26278
$29 \leftrightarrow 30$	26321	$60 \leftrightarrow 61$	25733
$30 \leftrightarrow 31$	26310	$61 \leftrightarrow 62$	26069
$31 \leftrightarrow 32$	26270	$62 \leftrightarrow 63$	25709
$\leftrightarrow$		$63 \leftrightarrow 64$	25812

Table 6.4: Duct example, a list of number of the inter-domain communication points.



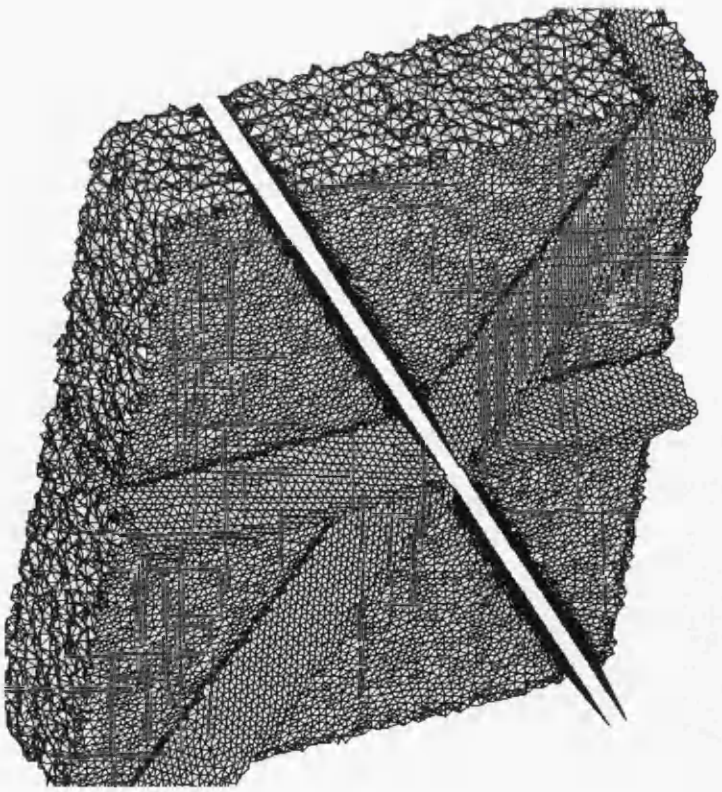


Figure 6.8: Duct example (cont.). Details from a sub-domain surface and volume grid.

A configuration of four military aircraft (*The load redistribution option*)

A very familiar procedure in the CFD community is to simulate aerodynamic problems considering half of the computational domain only, that is by exploiting the symmetry in the computational domain. However, with the ongoing developments in the CFD field and the computer technology, the interest in simulating the flow around more complex and non-symmetrical configurations will certainly grow more and more. Obviously, this is an area where parallel grid generation algorithms are expected to meet the challenge and deliver the inescapable *complete* large size grids. Such a configuration has been constructed in this example by arranging four military aircraft (i.e. B3 model) inside wind tunnel in a manner that mimics a real life flying position. Different views of the configuration are presented in Figure 6.9, point spacing sources are also shown, notice the *extra* sources used in the region of interest around the aircraft. An inviscid compressible flow simulation has been carried out on this configuration, employing a grid in the order of 62 million tetrahedra, is presented in Section 6.4.

A surface grid consists of 1,458,040 triangles and 729,030 points has been partitioned into 32 sub-domains, using the equal number of faces per sub-domain option. A few examples selected from the internal boundary grids are presented in Figure 6.10. Inspecting the details of the sub-domains grid presented in Table 6.5 shows how badly the elements are distributed among the sub-domains. In fact, the main goal of this example is to demonstrate the load redistribution option. However, it also ought to be mentioned, that number of faces listed in the table represent the *total*, i.e. after the internal boundary surface grids have been attached. Hence the original 'balanced' distribution of original boundary faces among the 32 sub-domains does not show.

By applying Hu & Blake algorithm, see Section 5.4.1, on the unbalanced sub-domains grid an element migration schedule is obtained. Details about this *recommended* schedule and the *actual* sub-sub-domains that have been extracted and transfered among the sub-domains are presented in Table 6.6. Continuing with the same parallel run, i.e. using one Worker only, two complete DPP loops only (first to extract and second to link) are needed to achieve the balanced elements distribution. A list of this sub-domains grid can be seen in Table 6.5, right hand side. Total time required by the redistribution procedure overall has been within 80 minutes, where only 5 percent of that is spent in the second DPP loop.

An example of the load redistributing algorithm action is illustrated in Figures 6.11 and 6.12, where three selected sub-domains and all the sub-sub-domains that are moved *from* or *to* them are presented. The impact of the load redistribution on the inter-domain boundary communication can be inspected by comparing the two lists of number of communication points, before and after in Table 6.7.

Domain	Elements.	Points	Faces	Domain	Elements.	Points	Faces
1	2234111	360386	67204	1	1487531	242924	56818
2	1950837	325920	96354	2	1487181	255848	104746
3	1221384	215018	104620	3	1487290	263298	132636
4	1155149	204587	103856	4	1487504	271676	171010
5	1093928	195977	105888	5	1487613	268770	156614
6	1738618	296547	113496	6	1487304	262301	132050
7	1366560	242548	125134	7	1487174	265124	142586
8	1718028	298822	133892	8	1487726	272911	171466
9	1818344	302038	88718	9	1486913	262508	132800
10	1720581	290255	95670	10	1486996	261053	127166
11	1078770	193943	104966	11	1487106	267622	148626
12	1087452	194568	104498	12	1487876	271972	164956
13	1461216	254319	114272	13	1487792	265188	143514
14	1399020	243601	110390	14	1487468	258280	115386
15	1216491	220224	127142	15	1488825	265948	141978
16	1914533	323364	112692	16	1485912	259726	120724
17	2309059	408845	106794	17	1486075	271542	118324
18	1403071	243477	103622	18	1487392	274395	123522
19	1059377	190948	104970	19	1487246	271676	163690
20	1061731	190980	105448	20	1487751	270908	161482
21	1582718	272115	111060	21	1487403	262594	133480
22	1066432	196957	122214	22	1491043	266995	142274
23	2054366	352444	142208	23	1484825	263358	135324
24	1302827	220555	79342	24	1485755	254776	101318
25	1872219	312736	97940	25	1487353	253166	96042
26	1106005	197307	104630	26	1487275	260758	125328
27	1126610	200485	104340	27	1489100	263176	132098
28	768620	142401	91860	28	1484629	266320	142914
29	2162747	365825	123670	29	1490508	262236	124410
30	920030	174068	120688	30	1484059	266586	142602
31	1512739	262837	113000	31	1487648	261650	126592
32	2110253	341228	66704	32	1487553	242817	56608

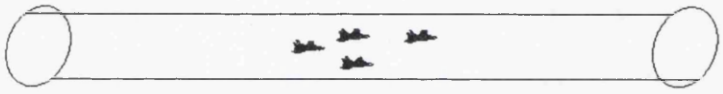
	Elements	Points	Faces
Total(Before):	47,593,826	8,235,325	3,407,282
Total(After):	47,593,826	8,429,262	4,189,084

Table 6.5: Details of the B3 example. 32 Sub-domains using the Direct Decomposition method with equal number of faces along the tunnel. Notice that total number of points and faces include the internal boundary grids counted with every sub-domain.

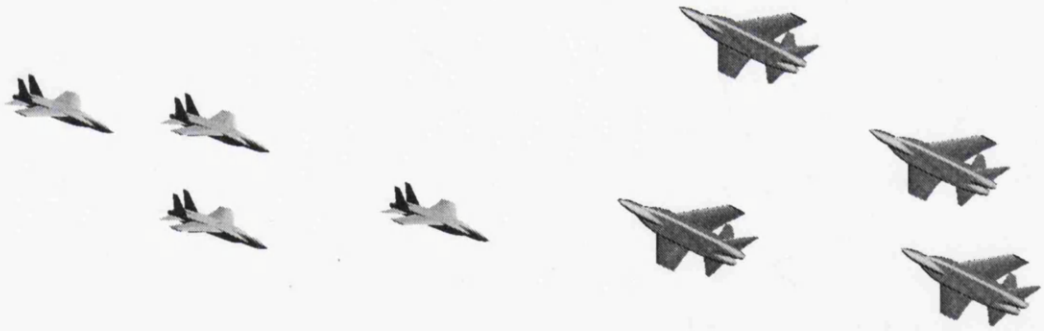
From_To	Recomm.	Actual
1 To 2	746804	746580
2 To 3	1210334	1210236
3 To 4	944411	944330
4 To 5	612253	611975
5 To 6	218874	218290
6 To 7	470185	469604
7 To 8	349438	348990
8 To 9	580159	579292
9 To 10	911195	910723
10 To 11	1144469	1144308
11 To 12	735932	735972
12 To 13	336077	335548
13 To 14	309986	308972
14 To 15	221699	220524
15 To 16	-49117	-51810
16 To 17	378109	376811
17 To 18	1199861	1199795
18 To 19	1115625	1115474
19 To 20	687695	687605
20 To 21	262119	261585
21 To 22	357530	356900
22 To 23	-63345	-67711
23 To 24	503714	501830
24 To 25	319234	318902
25 To 26	704145	703768
26 To 27	322843	322498
27 To 28	-37854	-39992
28 To 29	-756541	-756001
29 To 30	-81101	-83762
30 To 31	-648378	-647791
31 To 32	-622946	-622700

Table 6.6: A list of number of elements in the sub-sub-domains that are extracted and moved among the initial sub-domains in order to achieve a balanced elements distribution. The 'Recomm' column contains the numbers as recommended by Hu & Blake algorithm, while the 'Actual' column has the em exact number of elements. The difference between the two columns is due to the smoothing technique which is applied on the sub-sub-domains boundary while they are extracted.

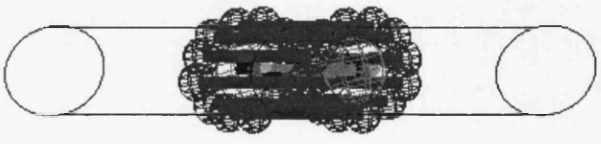
(I)



(II)



(III)



(IV)

Figure 6.9: Four aircraft configuration example. Different views of the geometry, general view of the four aircraft inside the wind tunnel presented in (II), and with the grid point spacing sources in (IV).

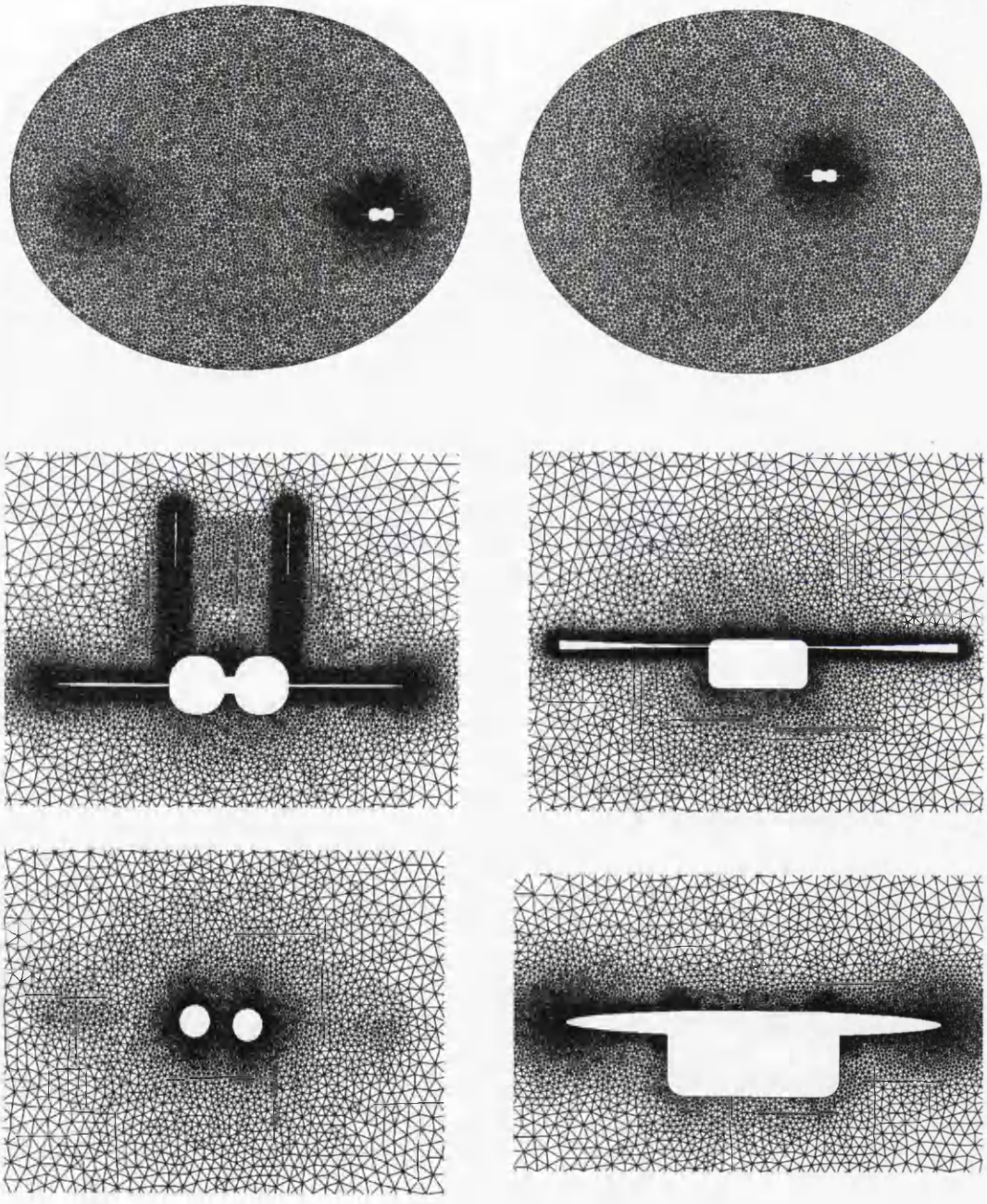


Figure 6.10: Four aircraft configuration example (cont.). Surface grids on selected internal boundaries.

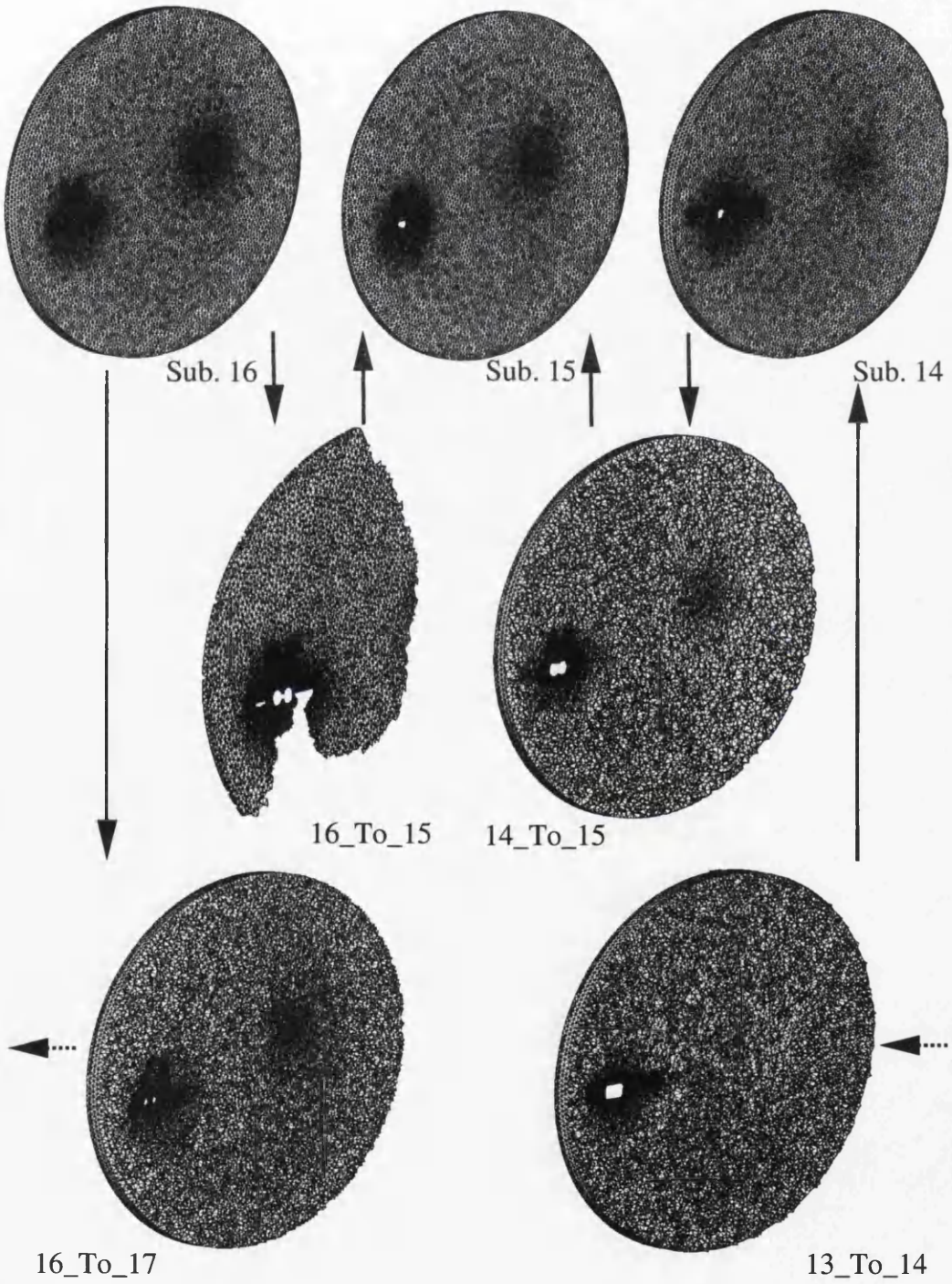


Figure 6.11: Demonstration of the load redistribution algorithm on three selected sub-domains. Surface grid of Sub-domains before moving elements (top), and the set of Sub-Sub-domains that to be moved out or linked to to each sub-domain.

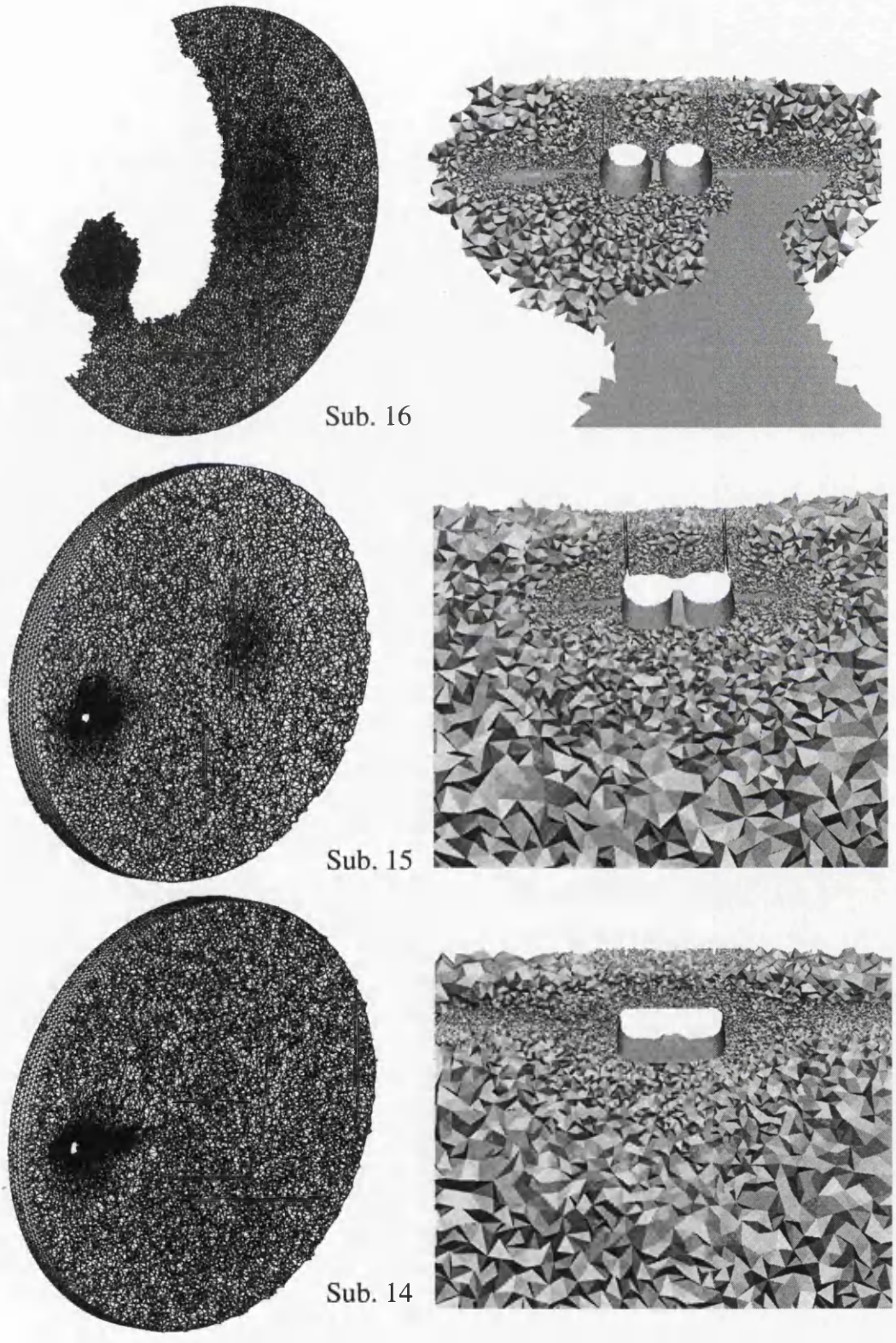


Figure 6.12: The same three Sub-domains, presented in the previous Figure, after the load redistribution algorithm has been implemented.

Between	Before.	Between	After
$1 \leftrightarrow 2$	10818	$1 \leftrightarrow 2$	11802
$2 \leftrightarrow 3$	14948	$2 \leftrightarrow 3$	22238
$3 \leftrightarrow 4$	15107	$3 \leftrightarrow 4$	26746
$4 \leftrightarrow 5$	14978	$4 \leftrightarrow 5$	29345
$5 \leftrightarrow 6$	15998	$5 \leftrightarrow 6$	21071
$6 \leftrightarrow 7$	18857	$6 \leftrightarrow 7$	24462
$7 \leftrightarrow 8$	21524	$7 \leftrightarrow 8$	24729
$8 \leftrightarrow 9$	17718	$8 \leftrightarrow 9$	31593
$9 \leftrightarrow 10$	10254	$9 \leftrightarrow 10$	21447
$10 \leftrightarrow 11$	15201	$10 \leftrightarrow 11$	25664
$11 \leftrightarrow 12$	15278	$11 \leftrightarrow 12$	27190
$12 \leftrightarrow 13$	14993	$12 \leftrightarrow 13$	24188
$13 \leftrightarrow 14$	16480	$13 \leftrightarrow 14$	19700
$14 \leftrightarrow 15$	19877	$14 \leftrightarrow 15$	19437
$15 \leftrightarrow 16$	21692	$15 \leftrightarrow 16$	22768
		$15 \leftrightarrow 17$	9148
$16 \leftrightarrow 17$	14918	$16 \leftrightarrow 17$	20213
$17 \leftrightarrow 18$	14490	$17 \leftrightarrow 18$	19316
$18 \leftrightarrow 19$	15079	$18 \leftrightarrow 19$	26627
$19 \leftrightarrow 20$	15417	$19 \leftrightarrow 20$	27929
$20 \leftrightarrow 21$	15457	$20 \leftrightarrow 21$	22284
$21 \leftrightarrow 22$	18117	$21 \leftrightarrow 22$	21931
$22 \leftrightarrow 23$	20981	$22 \leftrightarrow 23$	17369
		$22 \leftrightarrow 24$	9552
$23 \leftrightarrow 24$	16943	$23 \leftrightarrow 24$	20321
$24 \leftrightarrow 25$	11465	$24 \leftrightarrow 25$	12766
$25 \leftrightarrow 26$	15054	$25 \leftrightarrow 26$	17361
$26 \leftrightarrow 27$	15243	$26 \leftrightarrow 27$	20725
$27 \leftrightarrow 28$	14828	$27 \leftrightarrow 28$	18467
$28 \leftrightarrow 29$	15489	$28 \leftrightarrow 29$	29501
		$28 \leftrightarrow 30$	521
$29 \leftrightarrow 30$	18366	$29 \leftrightarrow 30$	11810
$30 \leftrightarrow 31$	19898	$30 \leftrightarrow 31$	30027
$31 \leftrightarrow 32$	12841	$31 \leftrightarrow 32$	12598

Table 6.7: A list of number of communication points between sub-domains before and after applying the load redistribution algorithm.

Vortex dynamic behind a civilian aircraft (*The construction of one global grid option*)

Demonstrating the framework with the option of 'constructing one global volume grid', and its implementation in the local partitioning of large sub-domains, is the main goal of this example. Having the grid designed to serve a certain task, i.e. investigating the vortex dynamic behind a civilian aircraft (see Section 6.4), a careful control of the grid point density and distribution is essential. Hence, a set of line sources that start from the wing tip and stretch back to a distance equivalent to 20 lengths of the aircraft body itself have been defined. See Figure 6.13 for the geometrical definition of the aircraft surface (i.e. Gulf Stream, EADS) and the grid point spacing sources as well.

In fact, the proposed example represents a very rare case as a volumetric grid. In particular, the relation between the size of the boundary surface grid and the closed domain volume grid. Since the sources have very limited effect on the original boundary so the size of the obtained surface grid has been, not surprisingly, as small as 223,932 triangles and 111,970 points. Subsequently, the number of sub-domains that can be created would be relatively small, or otherwise an extreme case of ill load balancing will be inevitable. On the other hand, dividing the domain using a small number of planar cuts along one axis is very likely to produce some large size sub-domains grid. Thus, implementing the local partitioning on such sub-domains had to be considered.

The computational domain has been divided initially into 8 sub-domains, using a set of planar cuts with locations already described along the major axis, see Figure 6.14. After gridding the internal boundary, constructing the closed sub-domains surface grid and establishing the inter-domain communication tables, each sub-domain would become a complete independent grid generation task. The total time needed for this operation, including the read-write activities on the hard disc, has been just about ten minutes employing one Worker only. Various number of sub-sub-domains have been used in the sub-domain local partitioning, which was obviously driven by the the size of the sub-domain volume grid. The 'equal number of faces' option has been adopted in the domain decomposition algorithm. Details of the final sub-domain volume grids with the number of sub-sub-domains used in the local partitioning are presented in Table 6.8, where a detailed description of one sub-domain grid is listed as well.

All the sub-domain grids have been generated and assembled using one Worker only using the same SGI Challenge machine used in previous examples. Although it would have been possible to construct the global volume grid on the same machine, the task was carried out on an SGI Onyx machine with 32(500 MHZ IP35, MIPS R14000) processors and 64 GBytes of shared memory. This is mainly due to the demand of the CFD parallel solver itself, see Section 6.4 for the use of generated grid in further simulations. The construction of the global grid has been completed within 8 minutes only, employing the same

program with the assembly algorithm is the only active procedure. Thus, the total time required for accomplishing the entire gridding procedure approximately reaches ten hours. Of course, this is an 'approximate' figure since the grid was not generated in one normal continuous run.

Two of the sub-domains (i.e. 1 and 2 as in the initial decomposition) have been selected to demonstrate this example. Where an illustration of the local partitioning in the first sub-domain is summarised in Figure 6.15, whilst various cuts in the volume grid are demonstrated in Figure 6.16. Similarly, Figures 6.17 and 6.18 present the second sub-domain. Also, see Figure 6.42 for a general view of a planar cut inside the global volume grid.

Domain	Elements.	Points	Faces	Local Part.
1	8778938	1560978	226510	2x2
2	18845430	3155838	139562	4x4
3	11725672	1961934	79764	4x4
4	6567154	1115057	110202	2x2
5	4406989	751929	88774	2x2
6	3865019	633373	73996	2
7	3482454	567059	62288	2
8	6399090	1067659	31838	2x2

Total:	Elements	Points	Faces
	64,070,746	10,666,263	223,932

Domain	Elements.	Points	Faces
1-1-1	1837303	332815	113220
1-1-2	2535691	444781	98066
1-2-1	1871442	338687	113938
1-2-2	2534502	444532	97956

Total:	Elements	Points	Faces
	8,778,938	1,560,978	226,510

Table 6.8: Statistics of the Gulf-Stream sub-domains grid. Details of the first sub-domain local repartitioning grids are presented. Notice the 'Totals' in here represent the grid after the assembly procedure, unlike the other examples where the internal boundary grids are counted more than once.

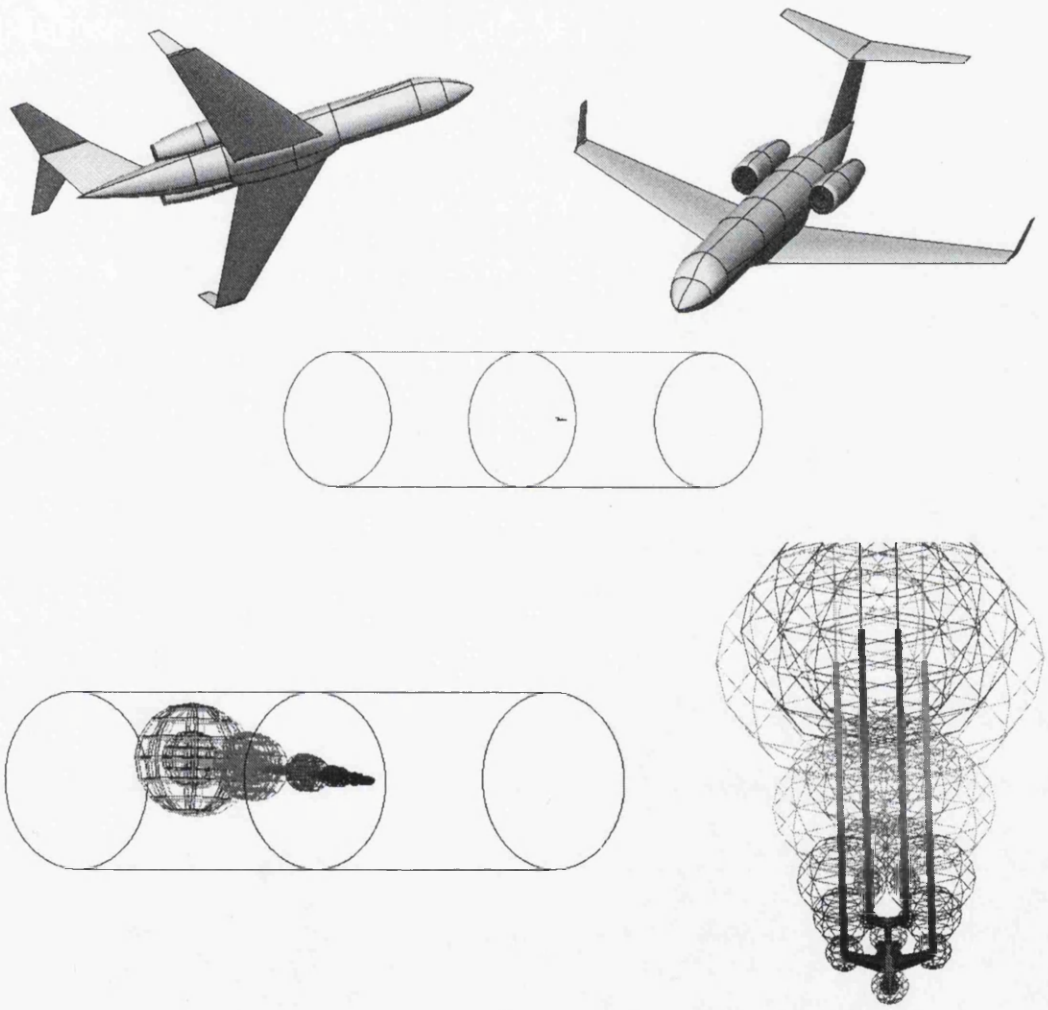


Figure 6.13: Geometry of the civilian aircraft (i.e. Gulf-Stream), and the grid point spacing sources defined in order to capture the vortices behind the wing tips.

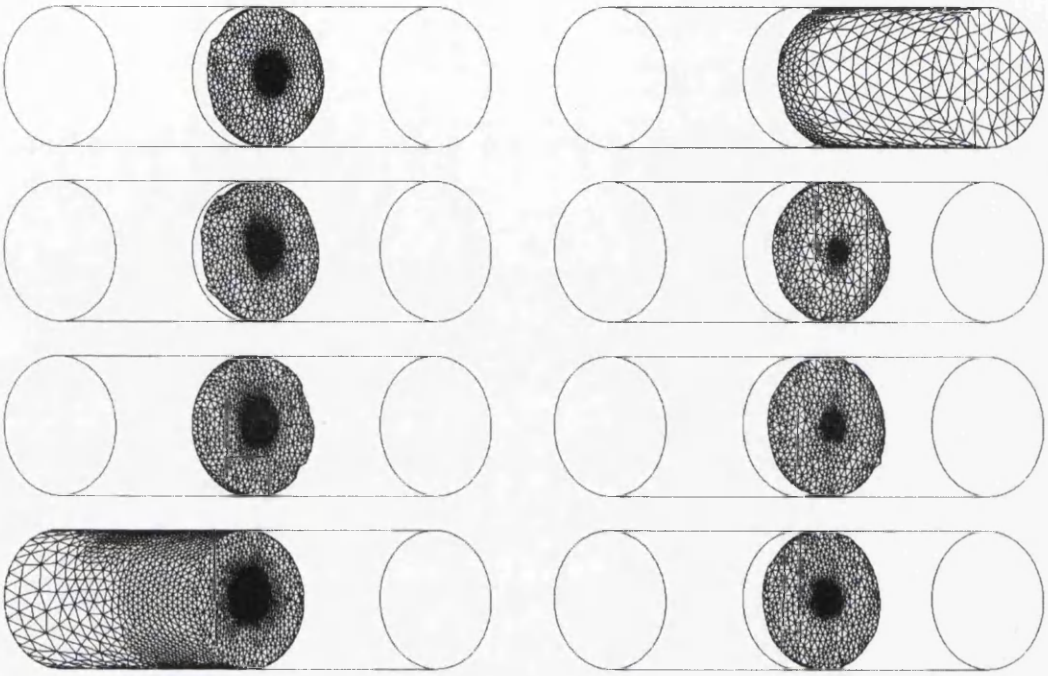


Figure 6.14: Surface grid on the sub-domains boundary, before carrying out the local partitioning.

6.3 Performance and Scalability

Whenever a parallel algorithm is presented in the literature an analysis of its performance is often presented as well. It is very common to see a couple of graphs/tables with some measurements like: the program is implemented on a parallel machine using P processors, and a speed up T is obtained for a problem size Q. Unfortunately, such analysis does not always, at least not in our case, offer an adequate overview of the algorithm behavior. What happens if there is a limitation on the number of available processors while the size of the problem grows?, how efficiently the available resources are used?, how does the communication cost change in respect to the problem size or number of processors?..etc. "a single performance measurement (or even several measurements) serves only to determine performance in one narrow region of what is a large multidimensional space, and is often a poor indicator of performance in other situations" [40].

Impact of DPP technique on the analysis policy

In fact, even for a reader who is familiar with more comprehensive analysis of parallel programs performance, some of the figures in this chapter may look rather 'strange'. For example, sometimes increasing the total number of processors N_{Proc} from 2 to 4 does not have any impact on the speed-up factor S_f

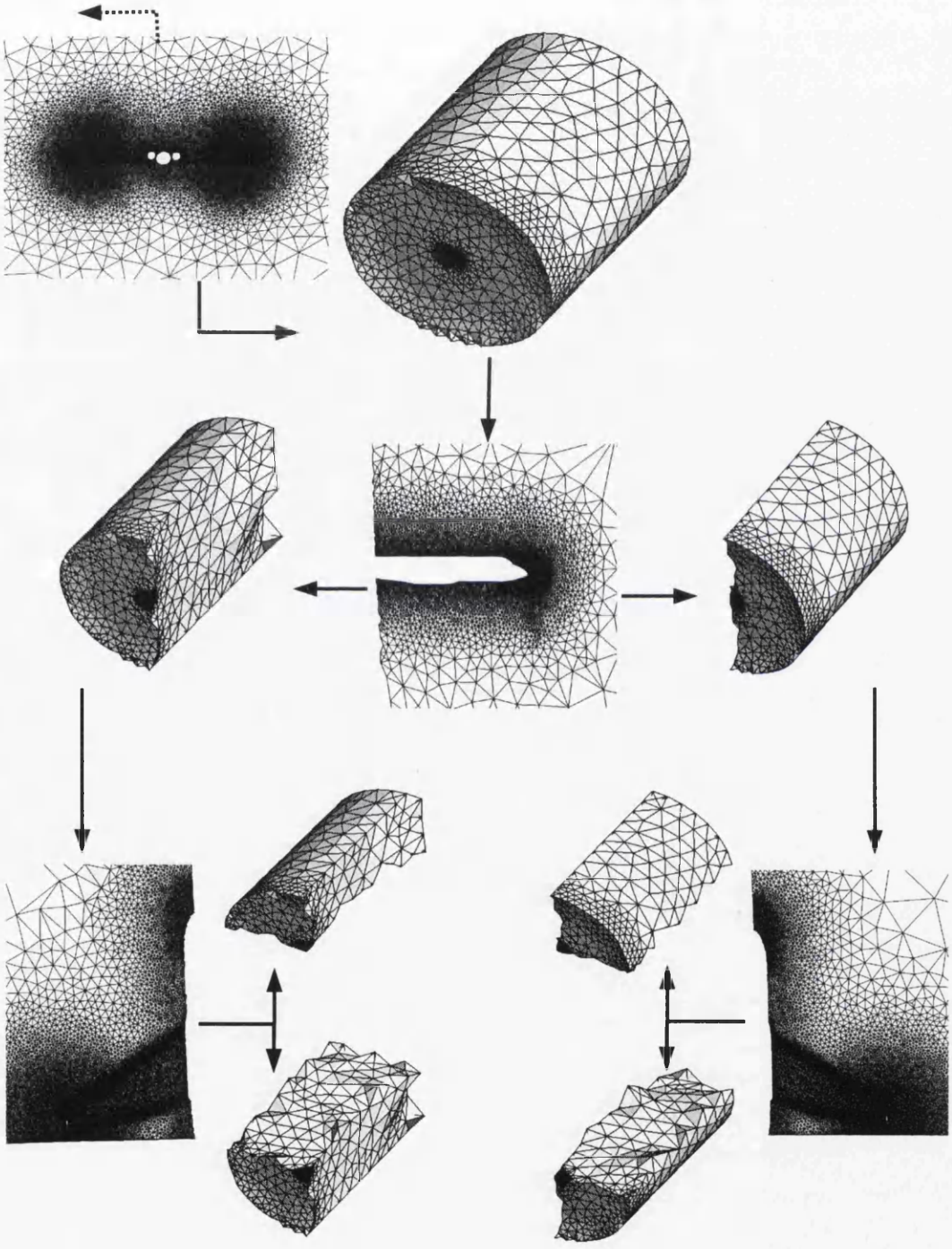


Figure 6.15: Illustration of the local partitioning of sub-domain (1) into (2x2) sub-sub-domains. Surface grids and some close-up on the internal boundary.

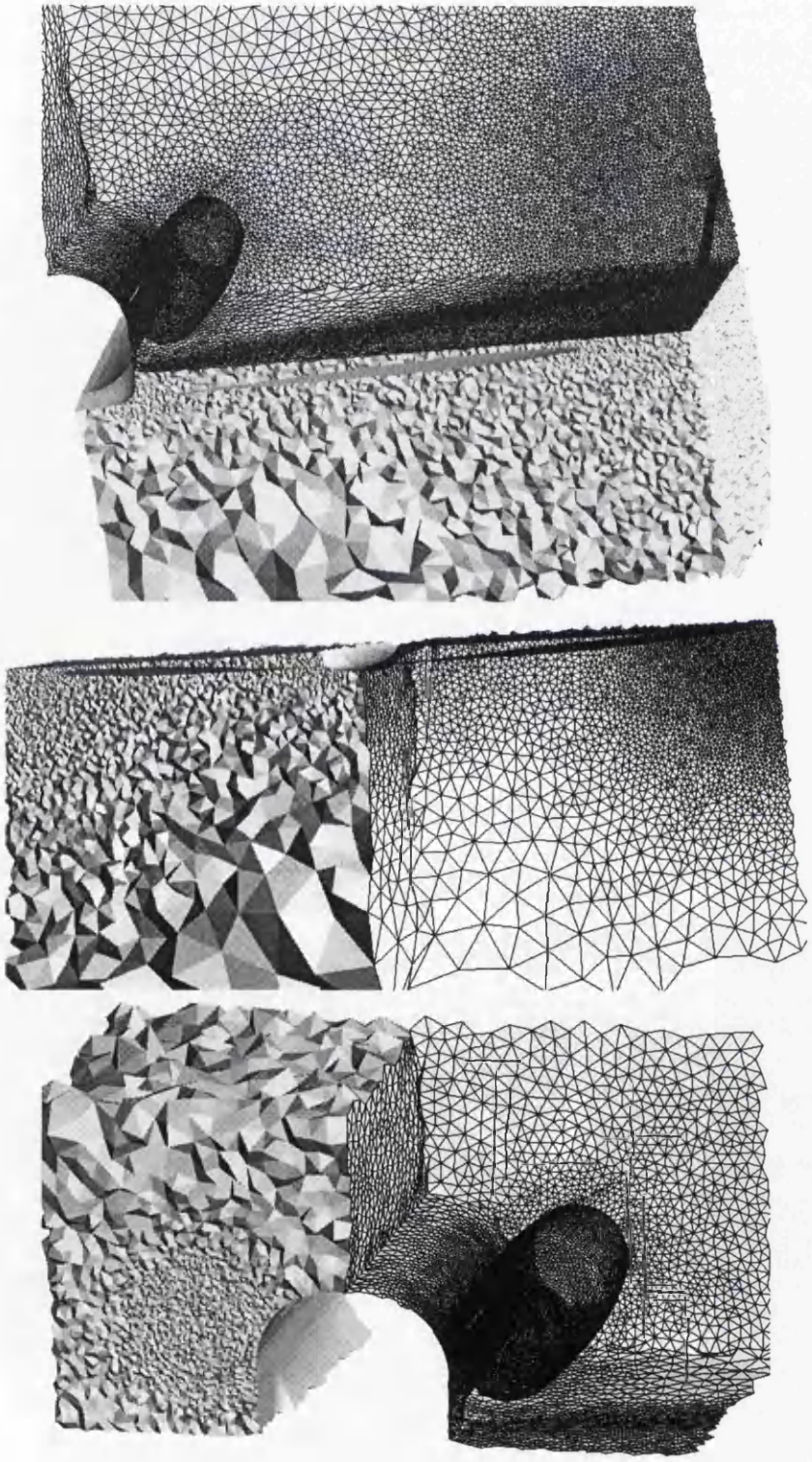


Figure 6.16: Details from sub-domain (1) volume grid.

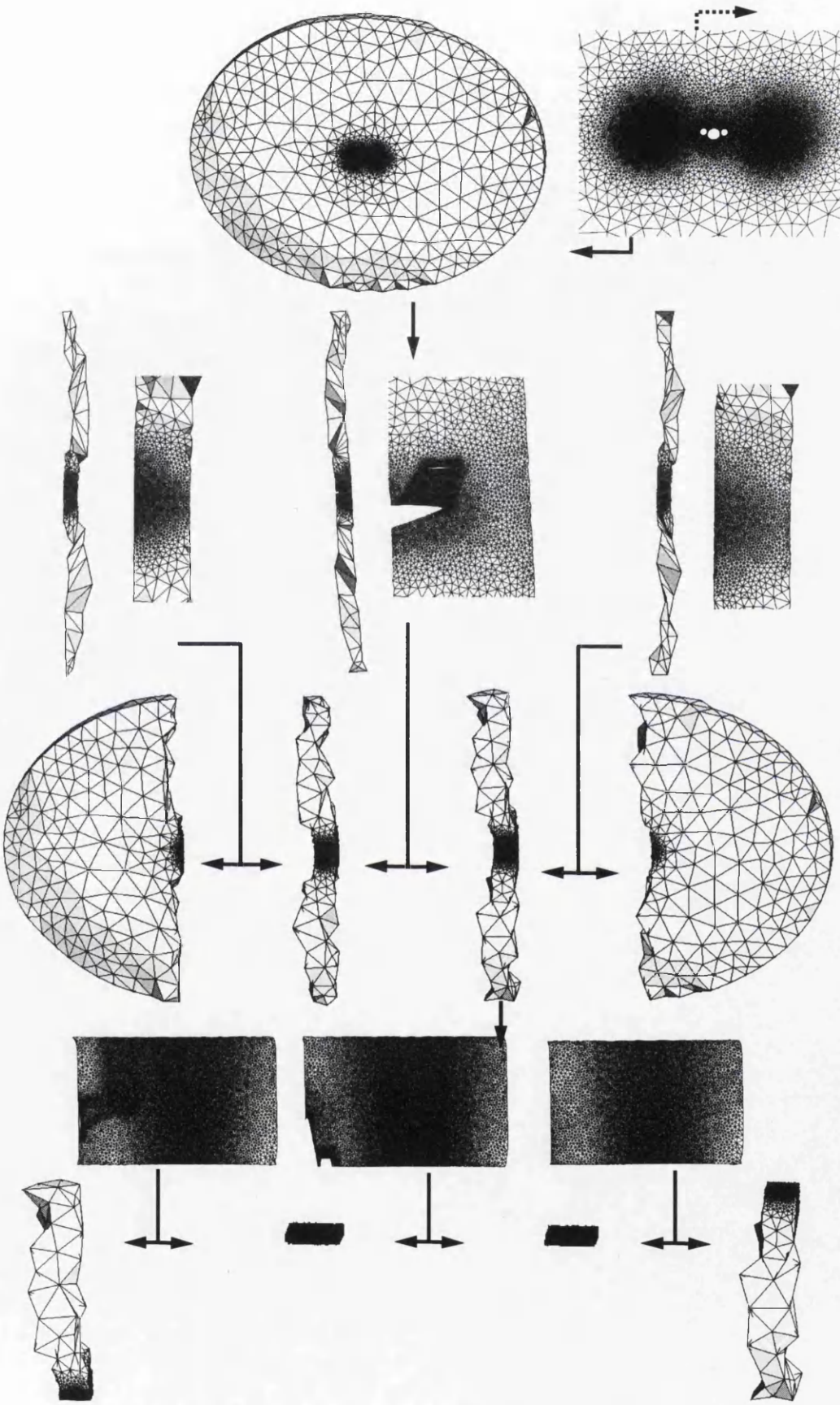


Figure 6.17: Illustration of the local partitioning of sub-domain (2) into (4x4) sub-sub-domains. Details of sub-sub-domains (2) only is presented.

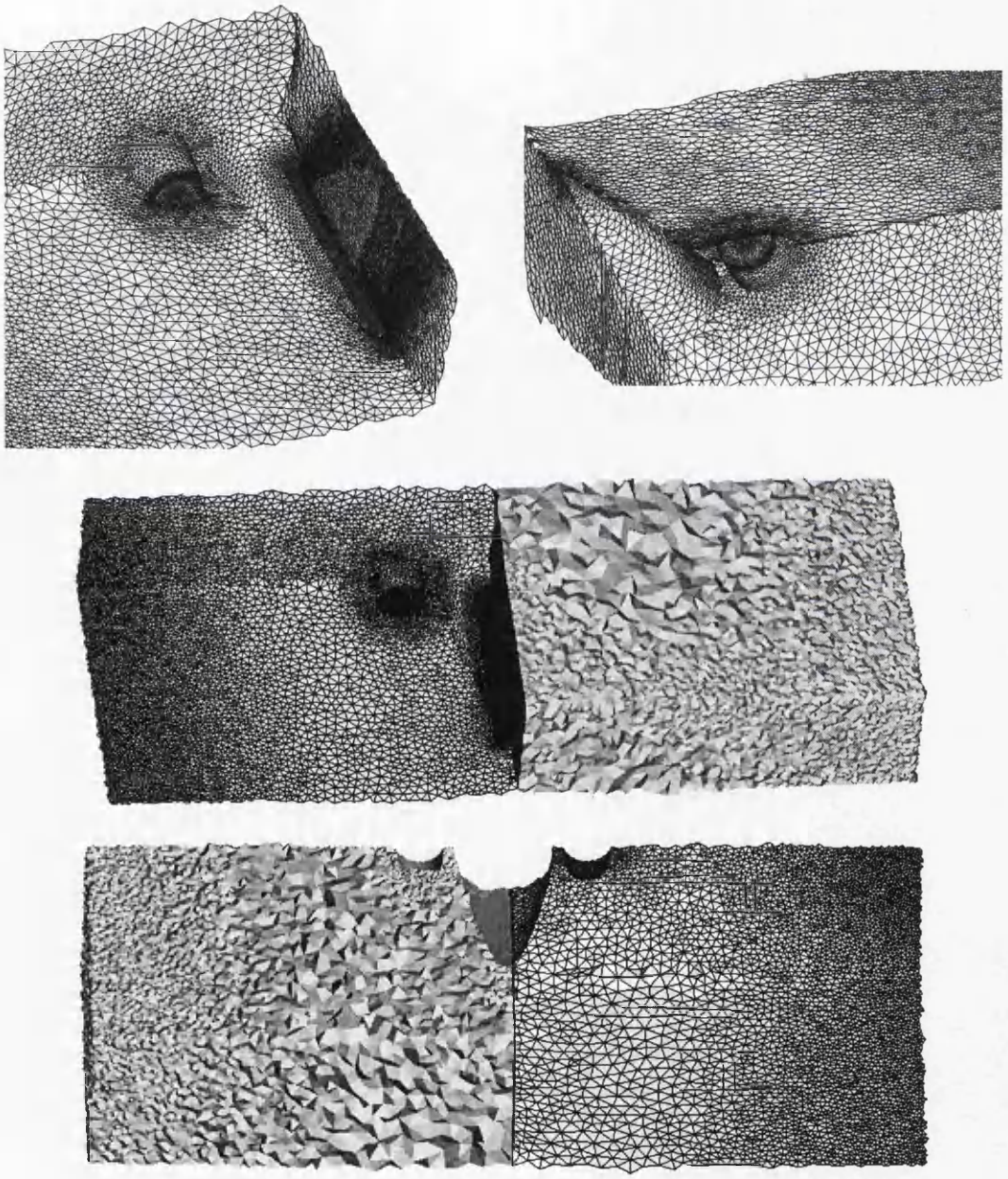


Figure 6.18: Details from sub-domain (2) volume grid.

whatsoever, whilst some other times adding one processor only gives a substantial increase. Furthermore, using a single processor a *parallel run* is still achievable, and it may give a speed-up factor of order 3!. That is all due to a simple fact: usually there are only two parameters that can effect the performance overall (number of processors and size of the problem), while here an additional parameter has to be considered (i.e. number of sub-domains). Apparently, the DPP technique allows the three parameters to emerge arbitrarily in any parallel run. Thus, an informative analysis must demonstrate the impact of changing each parameter, individually and combined, on the overall performance. A pattern of several cases can be observed, see Table (6.9)for a summary of these cases. References will be made to the relevant case number whilst the different measurements are discussed throughout the rest of this Chapter. In return, we believe this should help in forming a better understanding of the parallel framework performance and its dynamic nature.

Case No.	No. Proc.	No. Sub.	Size
1	F	F	F
2	F	F	C
3	F	C	F
4	F	C	C
5	C	F	F
6	C	C	F
7	C	F	C
8	C	C	C

Table 6.9: Pattern of the different cases discussed throughout this analysis. F = Fixed value for the parameter, C = Changing. Case No.1 represents one parallel run only, while any other case may have unlimited number of runs. Case No.8 will appear within a special form only, e.g. No. Proc. = No. Sub. while they both change in every run with the size!.

A few other factors can be brought into this study as well, such as the: variation in the involved processors performance, unbalanced workload distribution among the sub-domains, variation in the complexity of gridded geometries (between different examples, or different sub-domains within the same problem), ... etc. Not to underestimate any of these factors, but obviously any attempt to add one or two of them to the analysis parameters would have made the pattern presented in Table 6.9 far more complex.

To eliminate any other influence but the three parameters declared above, the same machine and geometry have been used throughout the analysis. The machine is an SGI Challenge with 8 processors and 512 MBytes of memory, and the geometry is a railway tunnel of length equals to 25*width. This example

provides: a well balanced workload distribution, same complexity of the sub-domains geometry and the possibility to create large number of sub-domains (See Table 6.10 and Figure 6.19). Certainly, the chosen geometry represents an 'ideal' case for the direct decomposition method, and therefore it does not reflect the status in the 'real world'. In addition, no reference is made to the indirect decomposition method throughout this analysis. Thus, it ought to be mentioned that: the main goal of this performance analysis is mainly to inspect the effectiveness of the adopted parallelisation strategy and the way it has been implemented.

Performance Analysis Tools

Unlike the traditional sequential programs, the performance analysis of parallel programs is a complex task, and some special tools may have to be utilised. UPSHOT is a very useful tool for understanding the behavior of parallel programs, it offers a graphical display of parallel time-lines. Each line is associated with a processor where coloured bars reflect on the state of the processor at any time. *Zooming* functionality is available and it can be utilised to find fine details about the time needed in every sub-task.

UPSHOT works on logging files that are generated by the Multi-Processing Environment (MPE) library [52]. Screen dumps of a typical output from UPSHOT can be seen in Figures: 4.5 and 6.21, whilst a description of the presented 'states' is available in Table 6.11. More information about UPSHOT and the MPE library are available in Appendix B. However, in short, by exploiting the MPE and UPSHOT a comprehensive performance analysis of the parallel programs could be achieved.

6.3.1 Speedup

The *exact* time needed by the processor(s) to execute a parallel program carrying out a certain task is not the most adequate way for evaluating its performance. Comparing this time to the time needed for a sequential program to carry out the same task may sound more satisfactory. Such a comparison has been adopted by the parallel processing community in a standard form, which is a very convenient measurement called as the 'Speedup factor'. The Speedup factor S_f, as defined in equation 6.1, represents the normalisation of the sequential time T_s to the parallel time T_p [3]. S_f is more commonly interpreted as: how many times the parallel program is faster than the equivalent sequential one.

[3]It can bee seen as the inverse of this in some publications.

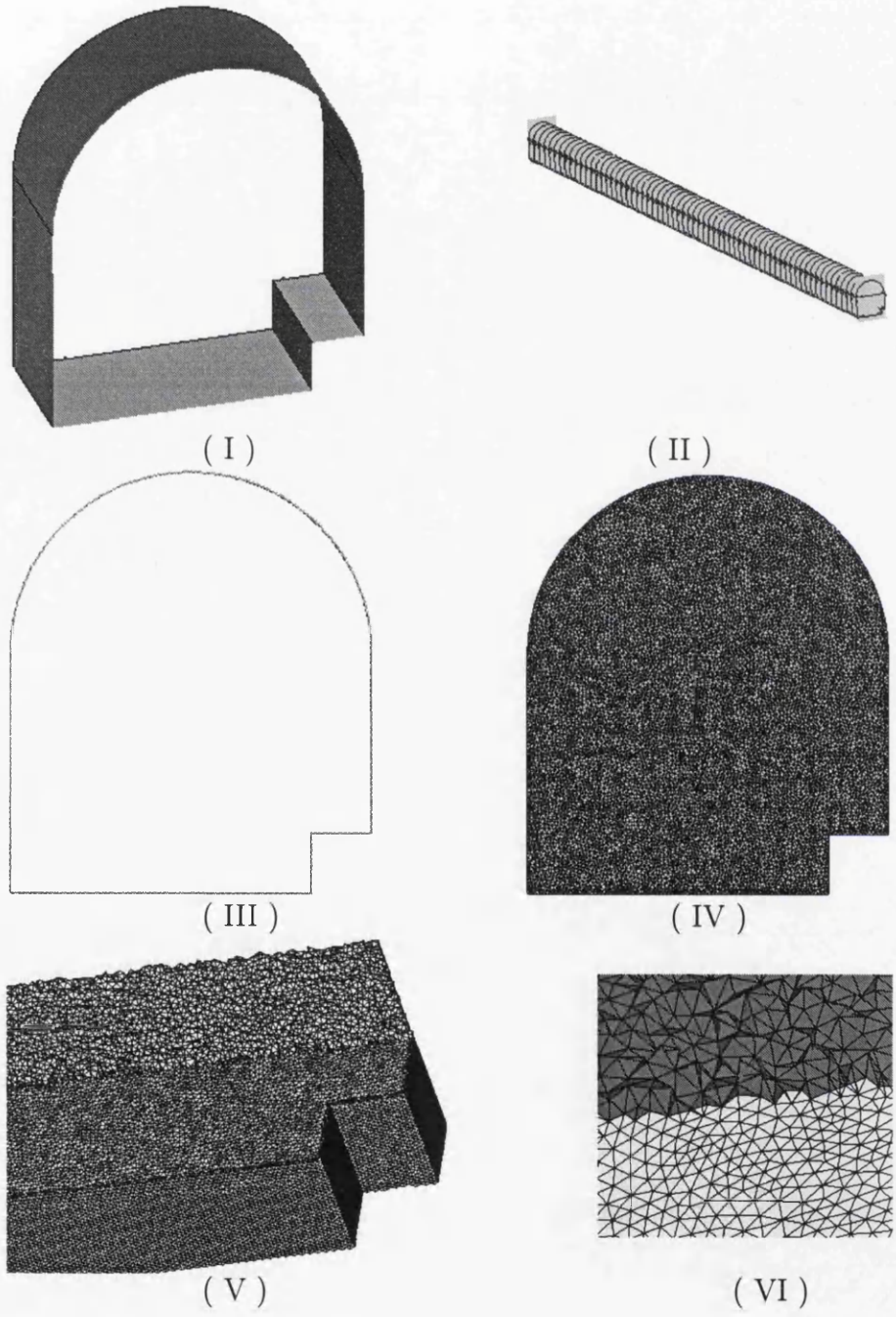


Figure 6.19: Details of the railway tunnel example: (I) and (II) Geometry, (III) Initial internal boundary, (IV) Surface grid of an internal boundary, (V) planner cut in a Sub-domain grid with the boundary of adjacent Sub-domain, (VI) Zoom in of (V).

Domain	Elements.	Points	Faces	Domain	Elements.	Points	Faces
64	1829418	300102	77384	11	1807715	299667	75628
1	1836366	303379	77314	36	1737793	288887	75622
3	1773821	294483	75818	26	1735908	288559	75616
60	1816536	301707	75806	33	1805151	299202	75608
4	1768546	293642	75790	23	1821067	301066	75604
50	1773814	294251	75762	2	1740045	289258	75602
5	1778260	295179	75756	49	1733174	288021	75598
27	1764884	291835	75756	24	1802687	298767	75598
18	1796701	297947	75754	47	1717361	285579	75596
31	1764290	292912	75754	39	1788062	296702	75596
37	1728684	287496	75754	14	1774602	294550	75594
51	1767397	293478	75748	9	1806050	299467	75594
63	1774606	294604	75736	48	1802515	298790	75558
40	1771716	294160	75710	45	1813354	300523	75558
30	1776533	294880	75710	52	1785645	296210	75556
57	1833528	303593	75692	42	1812662	300380	75556
12	1812614	300429	75690	32	1780623	295418	75552
62	1776321	294795	75686	46	1752352	291104	75536
7	1792454	297274	75682	17	1778693	295225	75536
21	1810653	300065	75680	35	1787402	296455	75530
56	1775575	294663	75674	13	1784198	296008	75522
38	1791530	297145	75674	54	1738609	288956	75520
43	1784728	296010	75658	8	1789699	296816	75520
6	1792276	297342	75656	44	1819724	301465	75516
41	1782546	295652	75654	34	1794325	297546	75510
58	1722605	286497	75650	15	1835894	304064	75508
59	1805064	299199	75648	19	1749150	290552	75502
22	1776574	294887	75644	28	1767042	293304	75476
10	1766003	293262	75644	29	1797168	297999	75466
55	1771500	294048	75642	20	1770999	292812	75442
61	1806495	299517	75638	16	1805870	299322	75396
25	1806572	299360	75638	53	1787114	295251	75392

Total:	Elements	Points	Faces
	114,151,263	18,931,718	4,843,210

Table 6.10: Details of the Railway Tunnel example, partitioned into 64 Sub-domains. Surface grid on the original boundary consists of 2,285,330 Triangles and 1,142,667 Points.

$$S_f = T_s/T_p \tag{6.1}$$

As stated earlier, *speed* wasn't considered as the major motivation behind this research, and a small value of S_f is still expected from the developed programs in some cases. That is very likely to be more frequent in the indirect decomposition method than in the direct decomposition method; mainly due to the time needed in generating the initial volume grid in the former method. Also, a rather disappointing speed up will be inevitable in the direct decomposition method whenever the total workload is poorly distributed on the sub-domains. However, as it will be presented shortly, a very good speedup factor can be achieved when an 'acceptable' workload distribution exists.

Since the evaluation of the speedup factor requires the sequential run time to be known grids presented in here will be smaller than 2.6 million elements, which is due to the limitation of the sequential grid generator on the default machine. Graph (I) in Figure 6.20 demonstrates a set of grids of size within the range [0.38 million - 2.53 million]. Each grid is divided into 8 sub-domains and generated 8 different times using [1 - 8] processors (i.e. `Case No.7`). It is observable in Figure 6.20.(I) that the achieved S_f varies irregularly in respect to the grid size. In fact, this is because of the time needed to generate these grids sequentially was not purely linear in the first place (see the sequential time needed for each grid in Table 6.12, page 180). However, for each individual grid, as expected, the speedup increases when more processors are added. But, apparently, the same speedup is obtained from using 4, 5, 6 and 7 processors!, thus an 'explanation' is needed.

Utilising the UPSHOT screen dumps, presented in Figure 6.21, details about one of the curves in the graph (i.e. the 2.17 million grid in Figure 6.20. (I)) can be examined. Image (IV) in Figure 6.21, presents the run when 4 processors are used and (V) for 5 and so on. It is recognisable in the four different runs mentioned above that the total time of each is heavily controlled by the time required to finish generating two sub-domains grid by one processor. Thus adding more processors (i.e. cases from 4 to 7 processors) cannot reduce the total time of the run significantly and the speedup factor remains almost the

State name	The associated procedure(s)
Part_ing	Partitioning, and deriving the internal boundary
Surf_Grid	Generating a surface grid on an internal boundary
Vol_Grid	Generating a volume grid in a sub-domain
Post_Proc	Post-processing operations,

Table 6.11: A list of 'states' defined in the framework, they are used by MPE functions while creating the 'logfile'.

same. Whilst adding one processor only (i.e. 8 processors in total) introduces a big jump in the curve. Graph (II) in Figure 6.20 demonstrates the relation between the speedup factor and number of utilised processors whilst the runs 5, 6 and 7 are omitted. However, the *efficiency* of parallel programs is a vital point, and it will be discussed thoroughly in the next section.

Speedup achieved employing one processor!

One of the most attractive features of the program developed in this study is its capability of generating large size grids using one single processor. Clearly, one may have a number of justified questions such as: how effective is this feature, is it possible to get a reasonable (if any at all) speedup with one processor only?, ... etc. However, an example of a grid generated several times on one processor is presented. The grid has been divided into a different number of sub-domains, which makes this example belong to the (`Case No.3`) category. Total number of tetrahedra is 2,199,070 and total time of each runs is presented in Figure 6.22.

It is recognisable in Figure 6.22 that a speedup factor greater than 1 was always achieved!. Which means that the time required to carry out all extra tasks introduced by the parallelisation procedure, e.g. partitioning and gridding internal boundary, is much less than the time gained by splitting the 'big' grid generation problem into a set of 'smaller' ones. In other words, generating unstructured grids in parallel using the developed program on a *single processor* can be faster than generating the same grid sequentially. Clearly, the obtained speedup will vary with the number of sub-domains, as it is associated with the ratio between time needed for the parallelisation work and the time saved in the volume grid generation. However, generally speaking, generating relatively small size grids in a very large number of sub-domains reduces the speedup obtained dramatically. In the presented example, the run associated with 32 sub-domains stands as a 'saddle point'. It is expected to have such a saddle point associated with a higher number of sub-domains for larger grids.

Speedup and the SPP solvers demand

To the best of our knowledge, most of the present parallel solvers still use the SPP (Static Parallel Processing) as their main parallelisation strategy. Hence, in order to match the total number of processors available every time, it is very likely to have the need for generating the same grid within a different number of sub-domains. The investigation of the previous example above shows the impact of changing the number of sub-domains on the speedup factor while number of processors is fixed, i.e. behavior of the program with respect to `Case No.3` in Table 6.9. Here we will examine the program performance with respect to `Case No.6`, which adds to the previous example the option of employing various numbers of processors. In fact, a careful reading of the two cases, i.e. `No.3` and `No.6`, gives a complete picture of the algorithm behavior in respect to the SPP parallel solver demands in general.

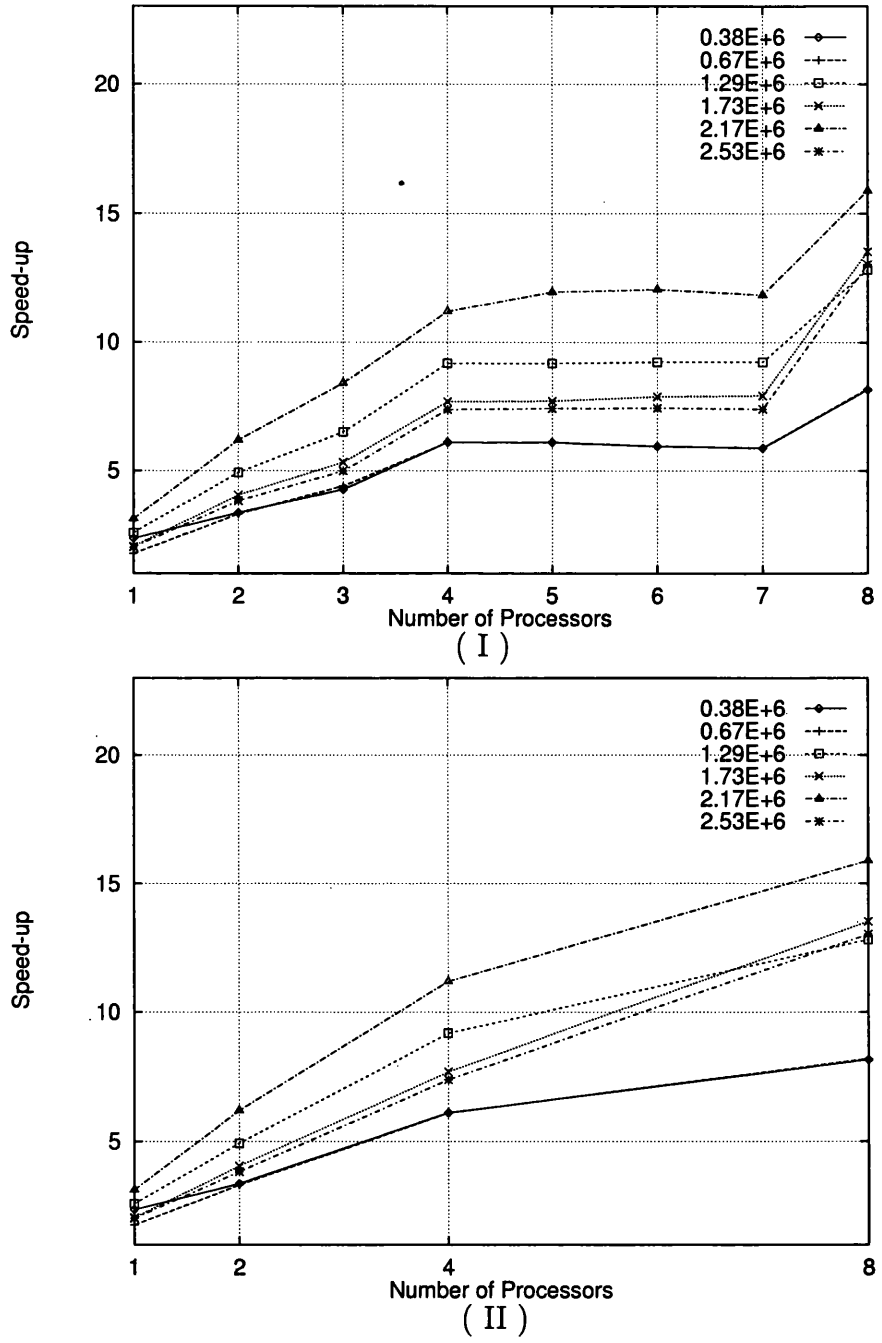


Figure 6.20: (I) The Speedup factor for a set of grids, each divided into 8 sub-domains and generated 8 different times on [1 - 8] processors (Case No.7). In (II) values related to the runs where equation 6.3 is satisfied.

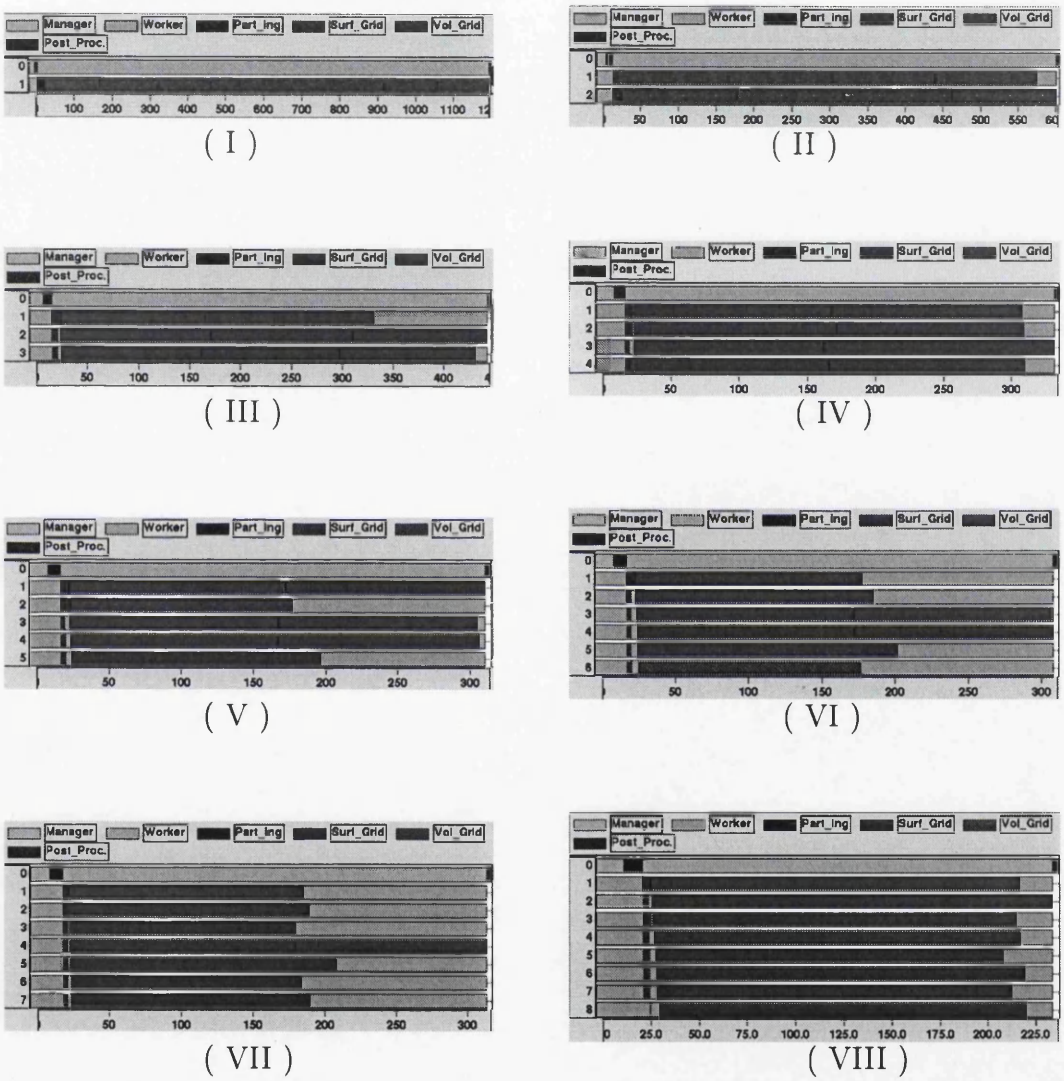


Figure 6.21: Dynamic Parallel Processing for a grid divided into 8 subdomains. Timing for 8 different runs as in: (I) using one processor, (II) using two processors, and so on.

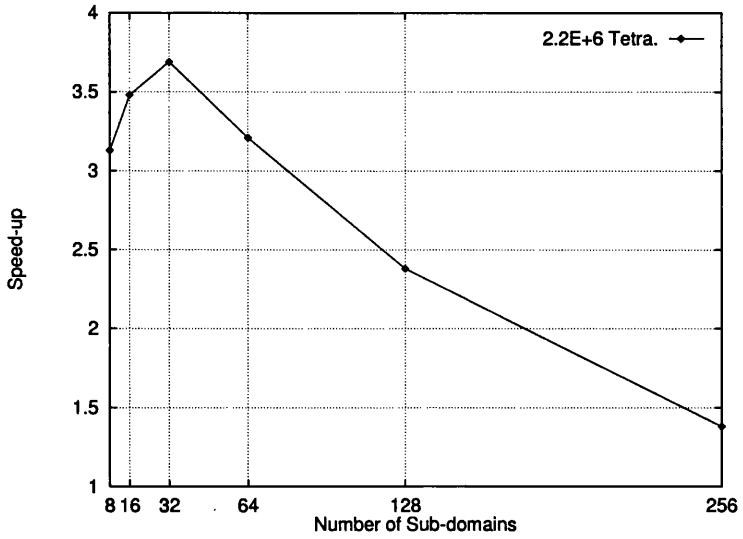


Figure 6.22: The Speedup factor obtained in generating the same grid several times within [8 - 256] sub-domains and on one processor only (Case No.3).

A grid consists of 1,311,711 tetrahedra is divided into n sub-domains, where n varies within the range [2 - 8]. Different runs for each n is carried out employing a number of processors in the range [1 - n]. Graph (I) in Figure 6.23 illustrates the speedup achieved for each run. Broadly speaking: a better speedup factor can be obtained if a sensible choice of number of sub-domains and number of processors is made. Observing Graph (I) shows that generating the same grid within 6 sub-domains on 5 processors is slower than generating it within 4 sub-domains on 4 processors, but generating it within 8 sub-domains on 4 processors is faster than within 6 sub-domains on 6 processors!. Although, this may sound rather confusing at the moment, it just reflects the fact that a good understanding of the dynamic nature of the program is essential in order to benefit from all its potentials. However, in Graph (II) of the same Figure, i.e. 6.23, only the runs that have number of sub-domains equals to number of processors are presented, where the 'linear' growth in the speedup factor with respect to the total number of sub-domains becomes more visible.

6.3.2 Efficiency

It is stated in Section 4.2.1 that the Dynamic Parallel Processing (DPP) technique provides a very efficient use of the computing resources available. In order to demonstrate this point, information is presented in graphs and tables such as Figure 4.5 and Table 4.1 In addition, expressions such as "the reduction in the total time of idle processors" had to be used. Apparently, this is

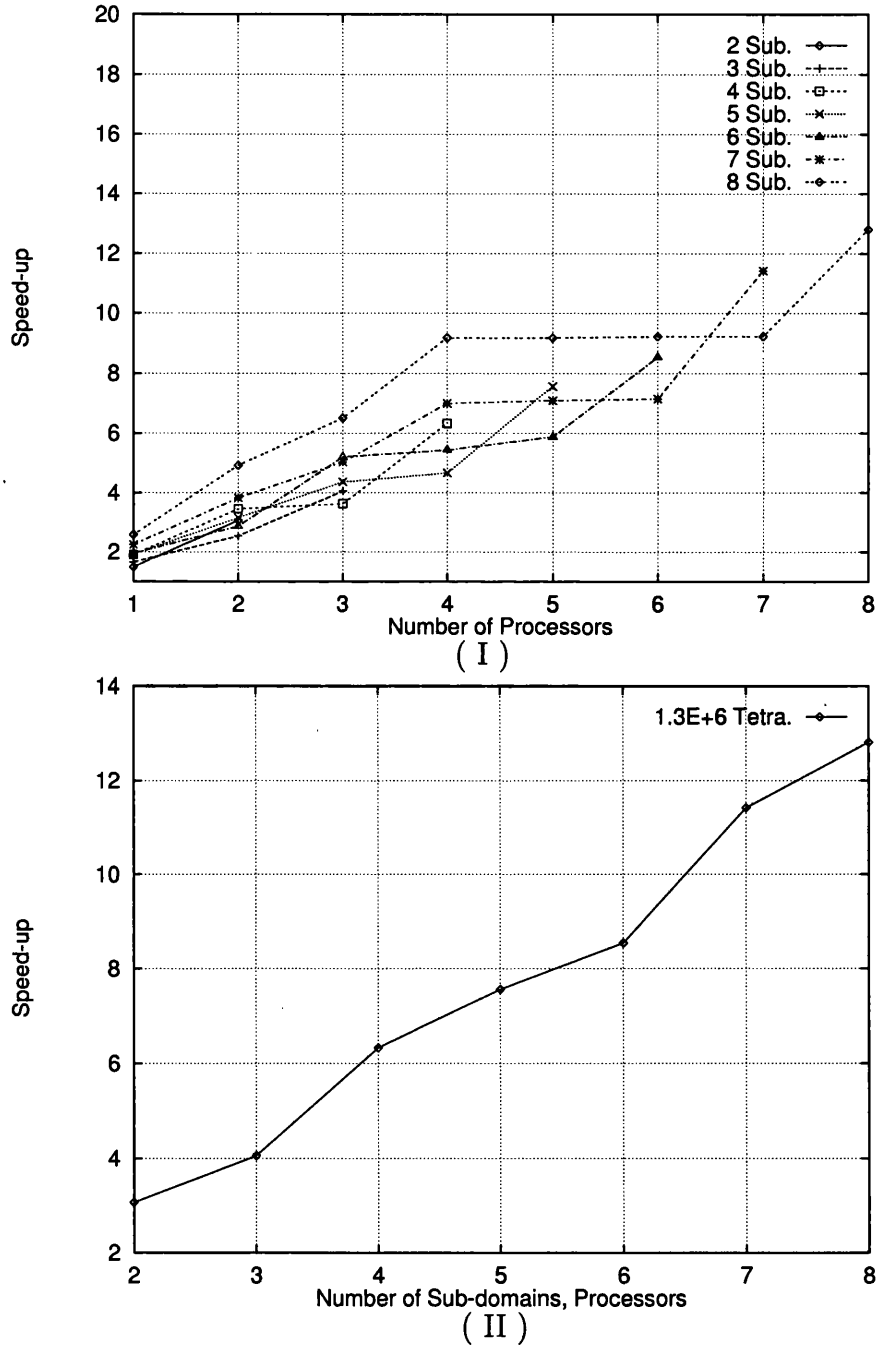


Figure 6.23: (I) The Speedup factor for one grid divided into a different number of sub-domains [2 - 8] and each generated few times using the range [1 - 8] processors, (Case No.6). In (II) values related to runs where No. sub-domains = No. processors.

a rather complex and 'inefficient' way to examine efficiency of parallel programs. Alternatively, a very convenient measurement that characterises the effectiveness of an algorithm in using the available resources can be adopted, it is the Efficiency factor (E_f), which is a well known measurement in the parallel processing community. E_f as defined in equation 6.2 shows: that a parallel program maintains the same level of efficiency when the Speedup factor S_f improves as in the order of the utilised processors N_{Proc}, (e.g. if J more processors are used, then S_f should increase in the order of J).

$$E_f = S_f / N_{Proc} \tag{6.2}$$

Recalling Graph(I) in Figure 6.20, where the same speedup is achieved using [4 - 7] processors, may suggest that the program has a major problem in respect to its efficiency. The discussion about the obtained speedup in one of the runs, as presented earlier utilising Figure 6.21, indicates that a 'careful' choice for the number of processors can eliminate most of the *inefficient* cases. However, it has been observed that equation 6.3 is a very effective criterion in choosing number of processors N_{Proc} to be employed for generating a grid within N_{Sub} sub-domains. Graphs (II) in Figures 6.20 and 6.24 illustrate the speedup and the efficiency, respectively, which were obtained from runs that satisfies equation 6.3.

$$N_{Sub} = N_{Proc} * K; \qquad \text{where} \quad K = \text{an integer.} \tag{6.3}$$

Efficiency with well balanced workload distribution

It is also mentioned in Section(4.2.1), that the DPP technique is implemented, more or less, to enhance the efficiency when a case of poor workload distribution occurs. Actually, a very well balanced workload distribution is reserved in all grids used throughout this analysis, details of one typical grid is presented in Table 6.10 [4]. However, the main interest is to show that even with such a well balanced workload the DPP technique still utilises the available resources more efficiently than the traditional SPP.

A grid consisting of 2,199,070 tetrahedra is divided into 8 sub-domains and generated 8 times employing [1 - 8] processors (i.e.`Case No.5`). An E_f in the order of 3.13 is achieved when the DPP technique is exploited (1 processor), while it drops to 1.98 if the SPP is used (8 processors), see (I) in Figure 6.24. In fact, the screen dumps of UPSHOT presented in Figure 6.21 correspond to this example. Thus, observing the total time spent by idle processors in (VII) in that Figure, i.e. Figure6.21 in page 172, explains the poor efficiency obtained when 7 processors are employed.

[4] Obviously presenting a similar table for each grid used in this analysis would have over crowded this chapter.

Efficiency of the program in respect to the two DPP loops

To study the effect of having *two* DPP loops on the efficiency of the program overall we reintroduce the same example of `Case No.6`, which has been investigated earlier (See Figure 6.23 on page 174 and the speedup and SPP solvers demand in page 170). The E_f obtained in this example is presented in Figure 6.25, whilst Graph (I) includes all runs, Graph (II) demonstrates the runs that satisfy equation 6.3 only. The extreme irregularity visible in Graph (I) must be well know to the reader by now, particularly in the light of the speedup result presented in Figure 6.23 and its discussion. However, what may still need an explanation is the sudden 'sharp' change at some of the runs presented in the Graph (II):

Apparently, equation 6.3 recommends that the number of processors is based on the desired number of sub-domains only, regardless of the number of the associated internal boundaries. All the runs presented in Figure 6.25 (II) satisfy the equation in their second DPP loop (gridding the sub-domains), but not necessarily in the first DPP loop (gridding the internal boundary). Consequently, an inefficient use of the recourses may occur in the first DPP loop.

Focusing on two of the curves only (4 and 6 sub-domains in Figure 6.25. (II)), it is noticeable that a lower level of efficiency is associated with the case where two processors are employed!. In fact, in both cases there is an odd number of internal boundary surfaces and one of the processor must have been 'waiting' while the other is dealing with the last internal boundary grid. However, it has been observed that the efficiency in the overall program is not effected unless number of employed processors and the achieved speedup are very small.

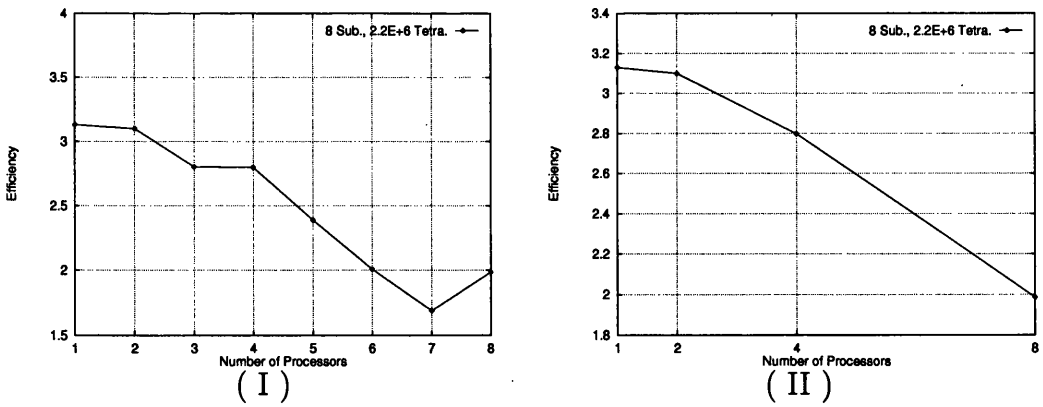


Figure 6.24: Changes of the Efficiency factor when various number of processors is employed to generate the same grid within the same number of sub-domains (Case No.5). All runs in (I), and only the ones satisfy equation 6.3 in (II).

A comparison between the efficiency achieved whilst employing the program in the SPP mode and DPP mode is presented in Figure 6.26, in which a typ-

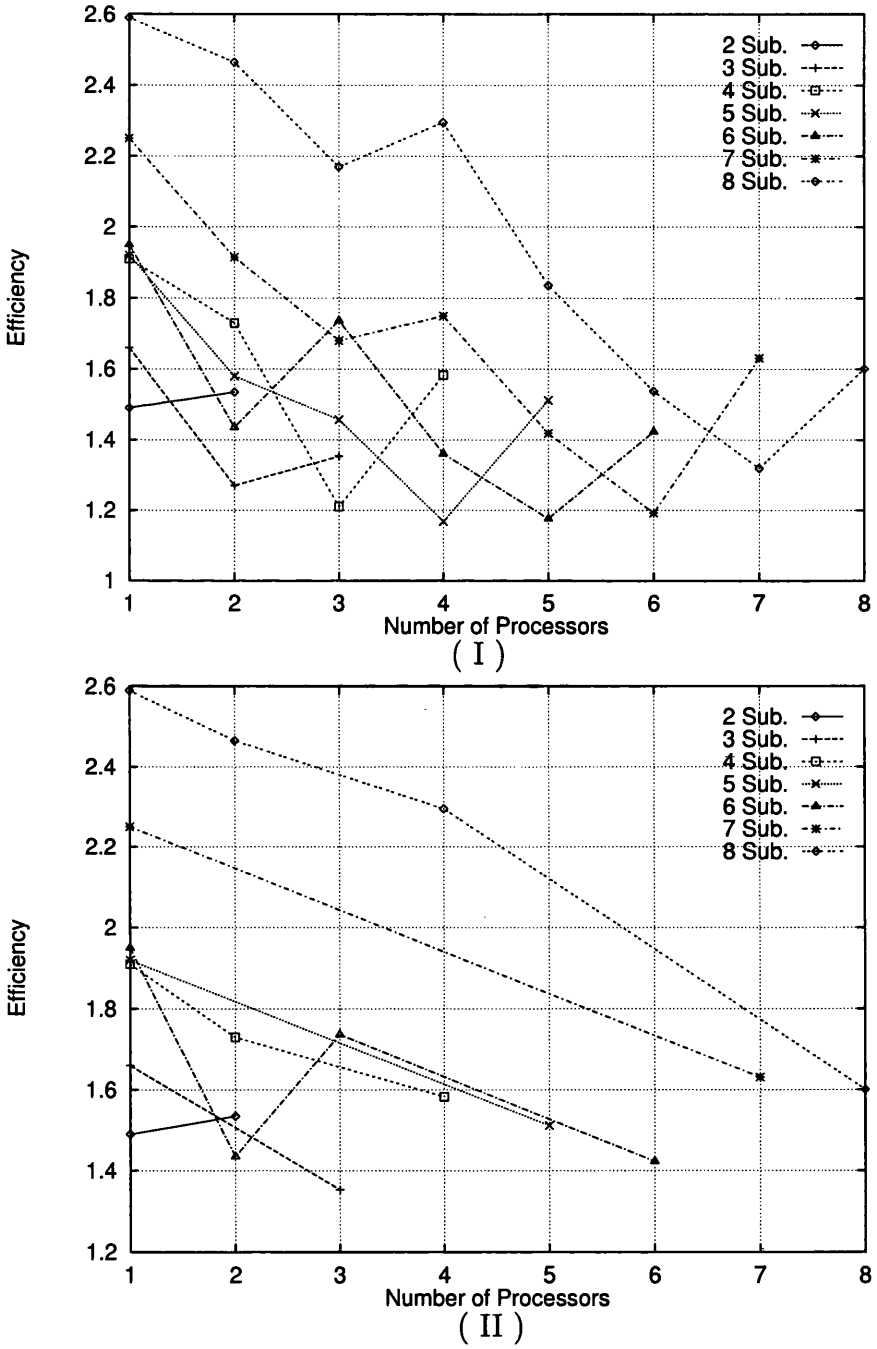


Figure 6.25: A grid of 1.3 million Tetrahedra divided into [2 - 8] sub-domains and generated on [1 - 8] processors, Case No.6. In (I) values of the efficiency factor achieved in each run, while in (II) the runs satisfy equation 6.3 only.

ical example of (Case No.7) is demonstrated. Recalling that (Case No.7) describes a case of fixed number of sub-domains and changing in number of processors and grid size, see Table 6.9 page 165; the graph in Figure 6.26 presents runs of several grids of size in the range of [0.18 million - 2.53 million], partitioned into 8 sub-domains and generated employing 1 (DPP) and 8 processors (SPP). In fact, observing this case as presented in Figure 6.26, in addition to the other cases presented in this section (Figures:6.24, and 6.25) a straightforward conclusion can be drawn: *although employing the developed program in the SPP mode may provide a faster option it is always recommended to use the DPP technique whenever the efficient use of available recourses is considered.*

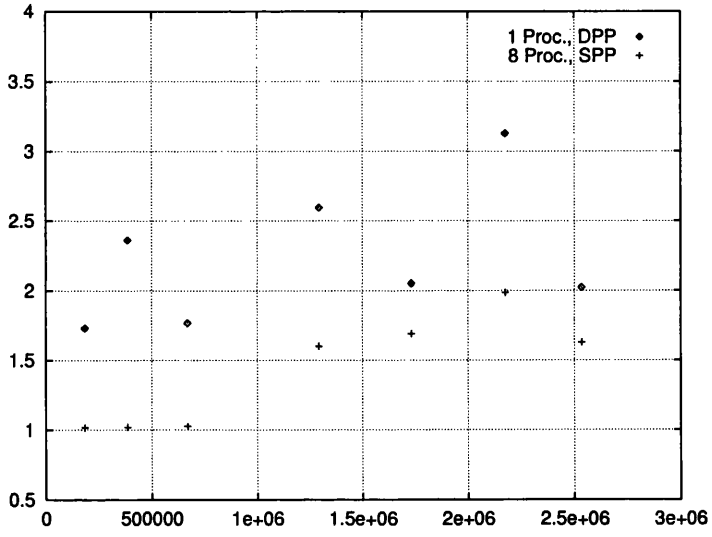


Figure 6.26: Comparison between the DPP and SPP efficiency using an example of Case 7: No. Processors =1 and 8, No. Sub-domains = 8, Size of grid within the range [0.18 million - 2.53 million].

6.3.3 Scalability

A wide spectrum of computers are used in engineering industries. Whilst, small companies rely on networked workstations and PCs, large firms intend to rely more on powerful servers and different type of workstations/PCs for the end users. On the other hand, it is easy to find a parallel algorithm that is expected to be used by both parties with a significant difference in the type of applications and size of computations. Recalling that a parallel algorithm is described usually as scalable when it enables the solution of large problems on large resources as well as it does for solving small problems on small resources.

Thus, the scalability of parallel algorithms designed to be used within the computational engineering discipline in general is a vital point.

In the early days of parallel computing, at the vector supercomputers generation, scepticism about scalability of parallel algorithms with large-scale engineering problems was widely spread. However, with the progression of the distributed memory parallel systems, practical experience shows that solving problems involving several million equations is a daily routine. Z. Johan et al. in [70] demonstrates how the parallel processing for finite element applications can be scalable on distributed memory machines. Performance analysis of two applications (computational aerodynamics and solid mechanics), in addition to a data decomposition algorithm, is presented. This analysis claims that a high scalability in the finite element parallel applications can be achieved on massive parallel computers. Apparently, most of the studies likewise [70] have missed out a simple fact, that is 'scalability must be available in all components of the numerical simulation discipline before we can consider it as truly scalable environment'. Thus, without a scalable grid generation algorithm the numerical simulation of engineering problems in general will have a constant bottle neck in handling large-scale problems.

Examining the scalability of the developed framework is carried out by investigating few cases in Table 6.9. In order to put the analysis of these cases in a more informative format, they are grouped in two different categories: scalability with the problem size and scalability with the computing resources. The first one reflects on the framework capabilities in generating large size grids even when the available resources are limited. Whilst the second demonstrates its great potential in exploiting any growth in the computing resources. The two following sections, 6.3.4 and 6.3.5, will reveal some relevant information to the scalability of algorithm in general as well.

Scalability With Problem Size

A set of grids of size in the order of [0.38 million - 20.5 million] Tetrahedra, each divided into 8 sub-domains and generated employing the same number of processors (`Case No.2`), are presented in Table 6.12. The time needed for two different parallel runs (1 and 8 processors), as well as the time needed to generate *some* of them sequentially are listed. The same information is represented in Figure 6.27(I), where the 'linear' growth in the time needed to generate the various grids in parallel is visible. Such a linear relation between the grid size and parallel run time is a clear demonstration of a highly scalable parallel algorithm.

To demonstrate the capability of the developed algorithm in generating large grids on the same platform we investigate `Case No.4`, where both the number of sub-domains and grid size can vary. Graph (II) in Figure 6.27 illustrates the time needed to generate several grids of size within the range [2.2 million

No. Tetrahedra	Sequential	1 Processor	8 Processors
385132	465	197	57
670095	525	297	64
1293429	1588	612	124
1730186	1839	896	139
2174469	3749	1198	236
2534687	3971	1963	305
3764040	NA	2527	451
4771702	NA	2806	532
6091983	NA	3931	572
7762985	NA	4969	708
10491519	NA	7824	1156
14003731	NA	10888	1463
20479678	NA	16878	2645

Table 6.12: Time in seconds needed to generate each grid sequentially, in parallel on one processor and on eight processors. Number of sub-domains in the parallel cases is fixed to 8.

- 114.2 million] generated on a single processor, each grid is divided into various numbers of sub-domains. The first thing to be noticed in this graph is: the difference in number of runs carried out among the grids. For instance, there are only two runs (64 and 128 sub-domains) for the grid of 114.2 million tetrahedra, and four runs (32 - 256) for the grid of 85.4 million tetrahedra. In fact, one may wonder if it is just a 'choice' or there is more into it?. Actually, the graph contains *all* the grids that could be generated on the platform [5]. Attempts made to generate any of the grids within smaller or larger number of sub-domain were not successful. Partitioning into a smaller number may produce a set of sub-domains where the grid size per sub-domain is bigger than the limitations of the sequential generator [6]. Whilst partitioning into a larger number may demand more memory storage than what is available.

Most of the *individual* sub-algorithms used throughout the partitioning procedure are not expensive, neither in memory requirement nor processing speed, but the real bottle neck emerges when number of internal boundary grows. The developed program, currently stores all the internal boundary information in the Manager before constructing the sub-domains boundary. Storing the internal boundary grids in the Workers memory where they are generated may minimise the effect of such bottleneck, but unfortunately not when a single processor machine or a shared memory machine is used. Obviously, this is

[5]Number of sub-domains not listed in the set presented in the Graph are not considered

[6]Obviously the local re-partitioning mechanism presented earlier in this chapter is not considered as a standard option in this analysis at all.

because in both cases Manager and the Workers use the same physical memory. However, modifying the developed program to implement this method for storing the internal boundary grid must be straightforward, though introducing some extra inter-processors communications activities will be inevitable.

The reader can consult Figure 6.22 (page 173) and the discussion of `Case No.3` in Section 6.3.1 in order to understand the behavior of the individual curves in Graph (II) in Figure 6.27.

Scalability With Computing Resources

An extensive study of the scalability, from the computing resources point of view, should cover the influence of the three major factors: processing speed, available memory and number of processors. It must be straightforward to understand the impact of the first factor, i.e. processing speed, on the program developed in this study. Clearly, improving the processing speed of *all* Workers homogeneously by s percentage will keep the same scenario of Manager-Workers interactions and advances the overall run time by the same s percentage. Whilst dramatic changes in such interactions are expected if the performance of *some* of the Workers were improved. Thus, although some workers may become much more effective the performance of the overall program may remain the same as before.

The analysis of `Case No.4` as presented above in Graph (II) in Figure 6.27, reflects on the impact of the second factor, i.e. the memory storage. It is recognisable that the 'limitations' of sequential grid generator and partitioning algorithm are strongly connected to the memory storage available on Workers and Manager respectively. More memory on the Workers increases the size of the grid per sub-domain, and more memory on the Manager increases the possibility for partitioning the domain into larger number of sub-domains.

It is appropriate to emphasis that the program does not inspect the computing resources available on each processor at all. The Manager synchronises the job processing on the Workers regardless their capabilities. Currently the Manager sorts the jobs based on their estimated workload and then sends the first job in the queue to the first available processor. But, an enhancement can be introduced by 'ranking' the Workers based on their capacities, and thereafter the Manager matches the first job to the most appropriate available Worker. However, the following discussion focuses on investigating the impact of the third factor only, i.e. number of available processors. The main goal in here is to find out if the algorithm can utilise effectively any increase in number of processors or not.

Recalling all cases defined in Table 6.9, page 165, shows that number of processors may change within Cases number 5, 6, 7 and 8. To avoid being repetitive, we recommend detecting the program scalability in respect to `Case No.7` and

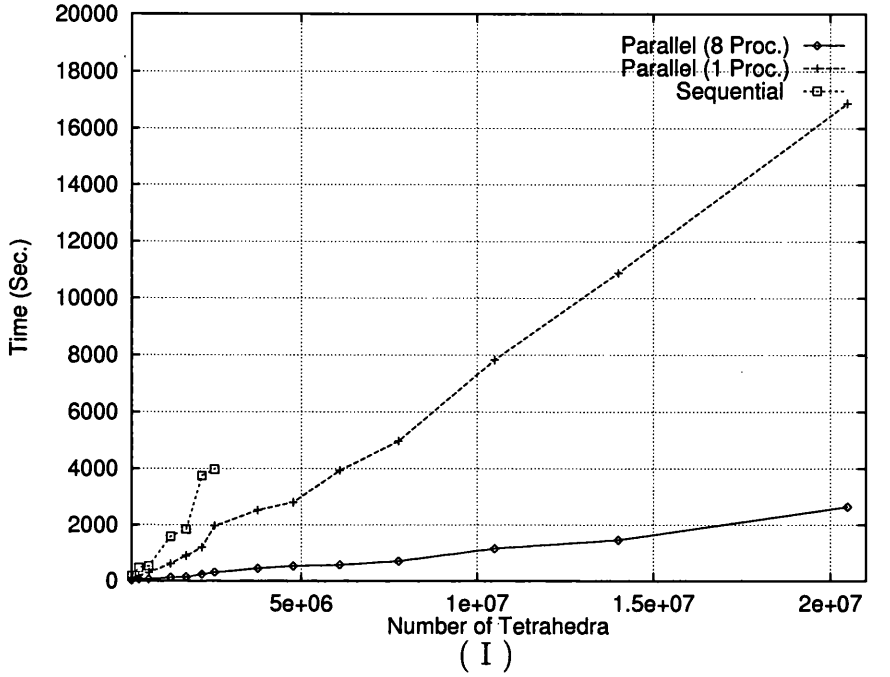


Figure 6.27: In (I), two sets of Case No.2 employing 1/8 processors, also the time needed by the sequential generator. In (II) Several grids generated on one processor within various number of sub-domains, Case No.4

`Case No.6` by observing Graphs (II) in both Figures 6.20 and 6.23 respectively. It is worth emphasising that a linear relation between speed-up S_f and number of processors N_{Proc} defines the ideal scalability status. Thus, the 'linearity' that is visible in Graph (II) in Figure 6.23 reflects a very healthy situation in the program. Similarly, `Case No.5` can be exposed by examining Figure 6.24. Thus, `Case No.8` is the only one left to be explored in here.

The following three different parameters: number of processors, number of sub-domains and size of grid may vary in `Case No.8`. Apparently, this represents the 'ultimate' condition for observing the program behavior, and it is a rather complex case to be summarised in one graph. However, the reader might be able to make his/her own judgment about this case by monitoring the analysis of all other cases presented so far. Nevertheless, an example of a 'simplified' form of case No.8 can be built utilising some already existing results. A brief description of this is presented as:
Recalling that the set of parallel runs presented in (II) in Figure 6.27 corresponds to `Case No.4`, in which number of sub-domains and grid size may change. Reproducing the same set of runs on various number of processors would then mimic the general form of `Case No.8`. In fact, the 'simplified' form mentioned above is constructed by 'calculating' the time needed in several runs in which number of processors is equivalent to the number of sub-domains. Details of the exact time needed in each individual sub-task were obtained from the MPE log-files, which had been produced during previous parallel runs operating on one processor. Utilising this break-down of the processing time on one processor a new set of 'projected time' can be evaluated. Graph (I) in Figure 6.28 represents the time of a set of runs for a set of different grids in which number of sub-domains is always equal to the number of processors.

A quick comparison between Graph(I) in Figure 6.28 and Graph (II) in Figure 6.27, where only one processor is used, demonstrates how effective the program would be in utilising any increase in number of processors. Of course, achieving a better timing for each run is not a surprise at all, however, what is more interesting is the dramatic changes in the characteristics of each curve. In fact, results presented in (I) in Figure 6.27 reconfirms what already has been stated "generating a small size grid within large number of sub-domains may not be effective at all". Graph (II) in Figure 6.28 represents the 'best' time achieved in every grid regardless the number of sub-domains used. One way to read this graph is: *the program developed in this study allows solving large problems on large resources as well as it does for solving small problems on small resources.*

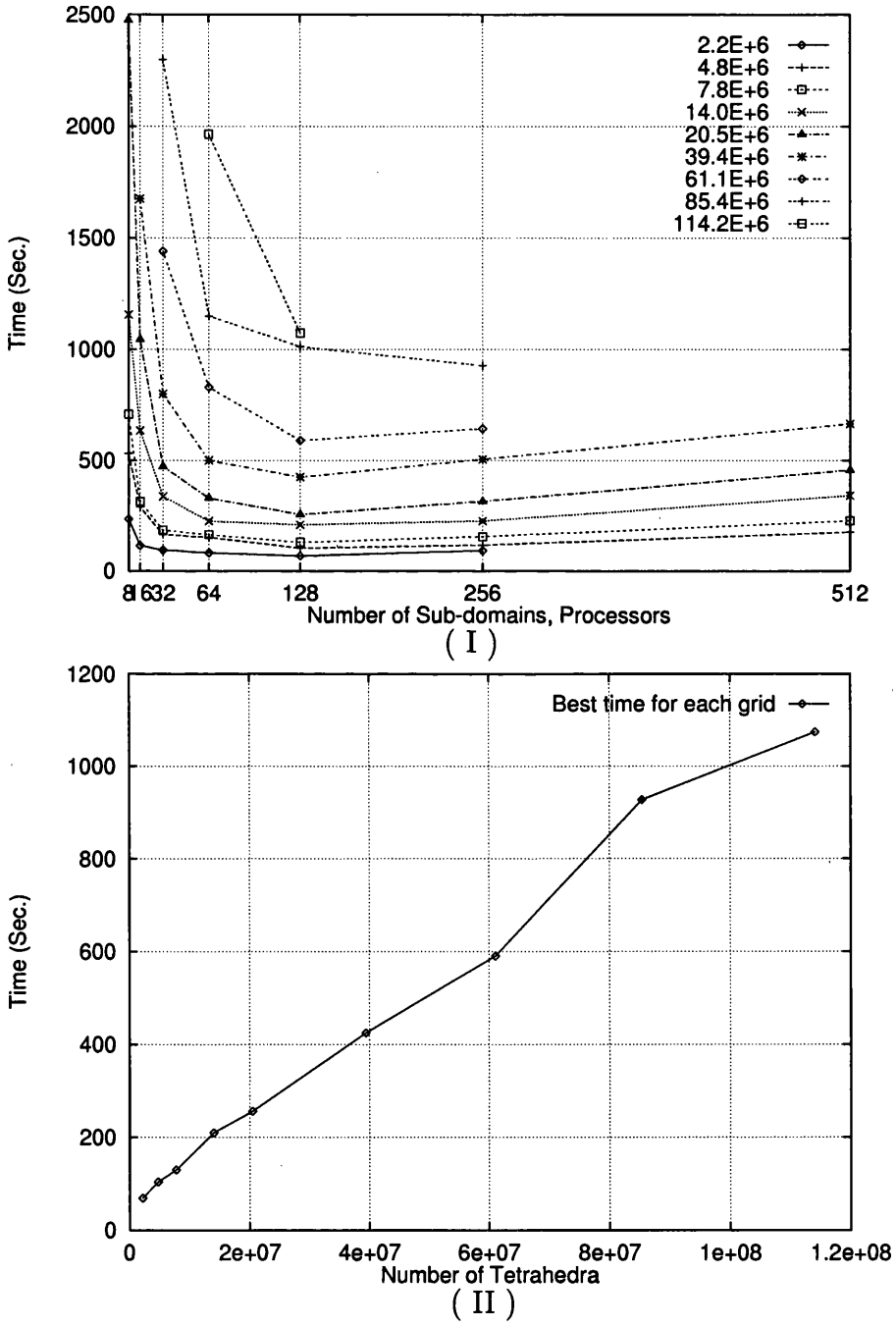


Figure 6.28: (I) Projected time for some parallel runs within Case No. 8, with the condition $N_{Sub} = N_{Proc}$. (II) Best time achieved for each grid presented in (I)

6.3.4 Amdahl's Law

It is very common to have in every parallel algorithm some components that are not possible to parallelise. These sequential components will eventually limit the speedup factor that can be achieved. Amdahl's law is a well known function that describes this issue [68]: Recalling the definition of the speedup factor in equation 6.1, and rewriting the total parallel time as:

$$T_p = T_{p_p} + T_{p_s}$$

where T_{p_p} is the time needed for the parallel components of the program, and T_{p_s} for the sequential components. The maximum speedup factor $S_{f_{max}}$ which can never be exceeded defined in equation 6.4.

$$S_{f_{max}} = S_f; \qquad \text{where } S_f \rightarrow \frac{T_s}{T_{p_s}} \qquad \text{when } T_{p_p} \rightarrow 0 \qquad (6.4)$$

In the early days of parallel computing, it was widely believed that this effect would limit the utility of parallel computing to a small number of specialised applications. Practical experience shows that this inherently sequential way of thinking is of little relevance to real problems [40]. However, this all can conclude as: "a *good* design of parallel algorithm must keep the sequential parts in it as minimum as possible".

The parallelisation strategy adopted in this research has left out two parts of the general algorithm, they are the partitioning of the domain and the construction of the inter-domain communication tables. The latest is an identical task in both direct and indirect decomposition methods but the former differs dramatically. Obviously, it requires more time in the indirect decomposition method, since it involves the generation of an initial volume grid. Nevertheless, to estimate the impact of these two sequential parts on the parallel algorithm overall, two examples of the direct decomposition are demonstrated.

Both examples illustrate the percentage of the time needed in the sequential parts, to the total time of the run overall employing one processor only. The first example reflects the case where the size of the grid changes with a fixed number of sub-domains (see Graph (I) in Figure 6.29), and the second reflects on the case where number of sub-domains in the same grid changes (see Graph (I) in Figure 6.30). Graph(II) in each of these two figures demonstrates the same result as in graphs (I) presented as a 'projected time', under the condition of the number of employed processors is equal to number of sub-domains. In each column of the histogram, total sequential time is presented at the bottom whilst the upper part represents the time needed to generate a volume grid in one sub-domain and a surface grid on one internal boundary.

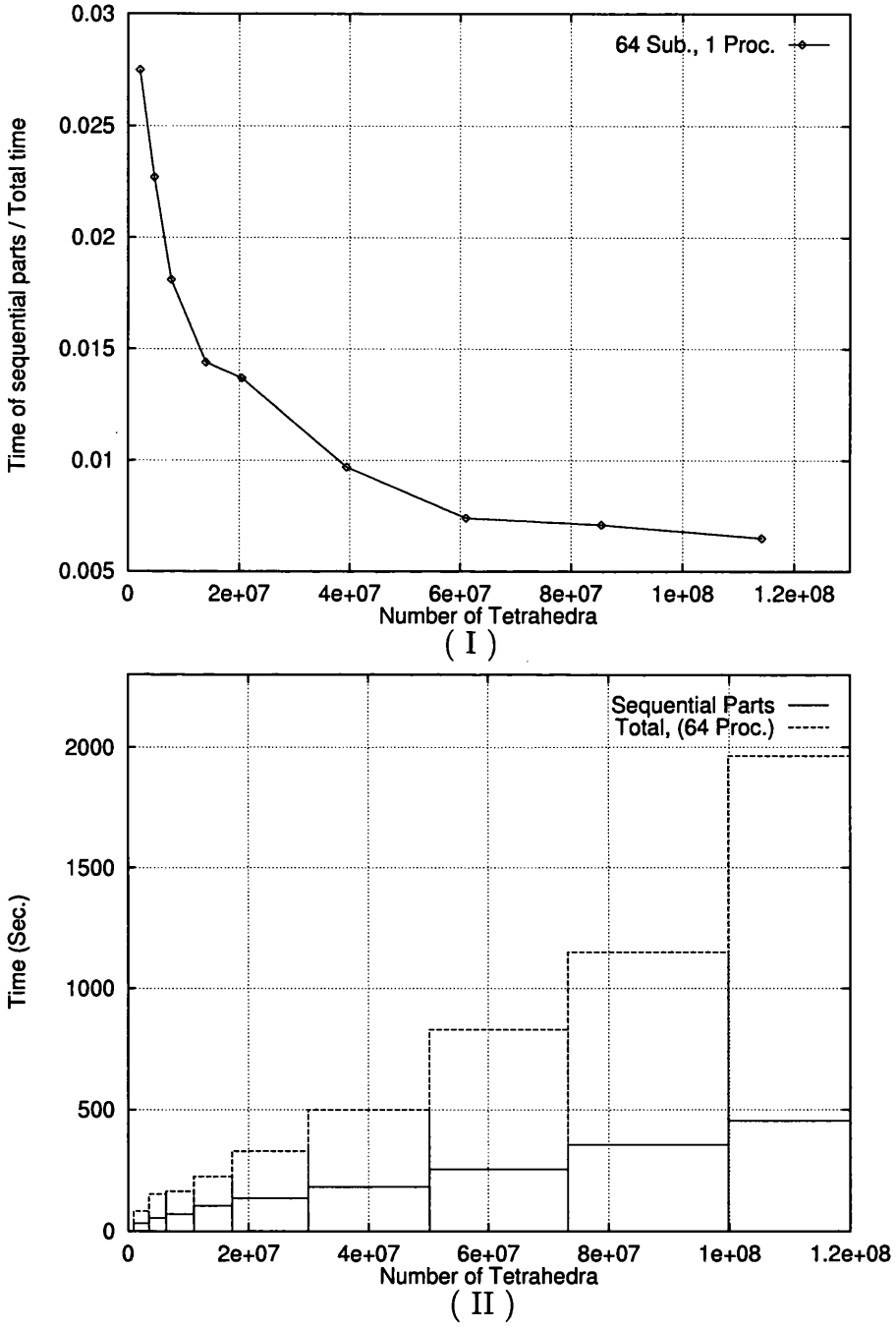


Figure 6.29: Time required by the sequential parts in respect to the total time of the parallel run. Grid size in the range [2E+6 - 114E+6] tetrahedra and partitioned into 64 of sub-domains. (I) Exact timing using one processor, (II) projected timing by assuming number of sub-domains is equal to number of processors.

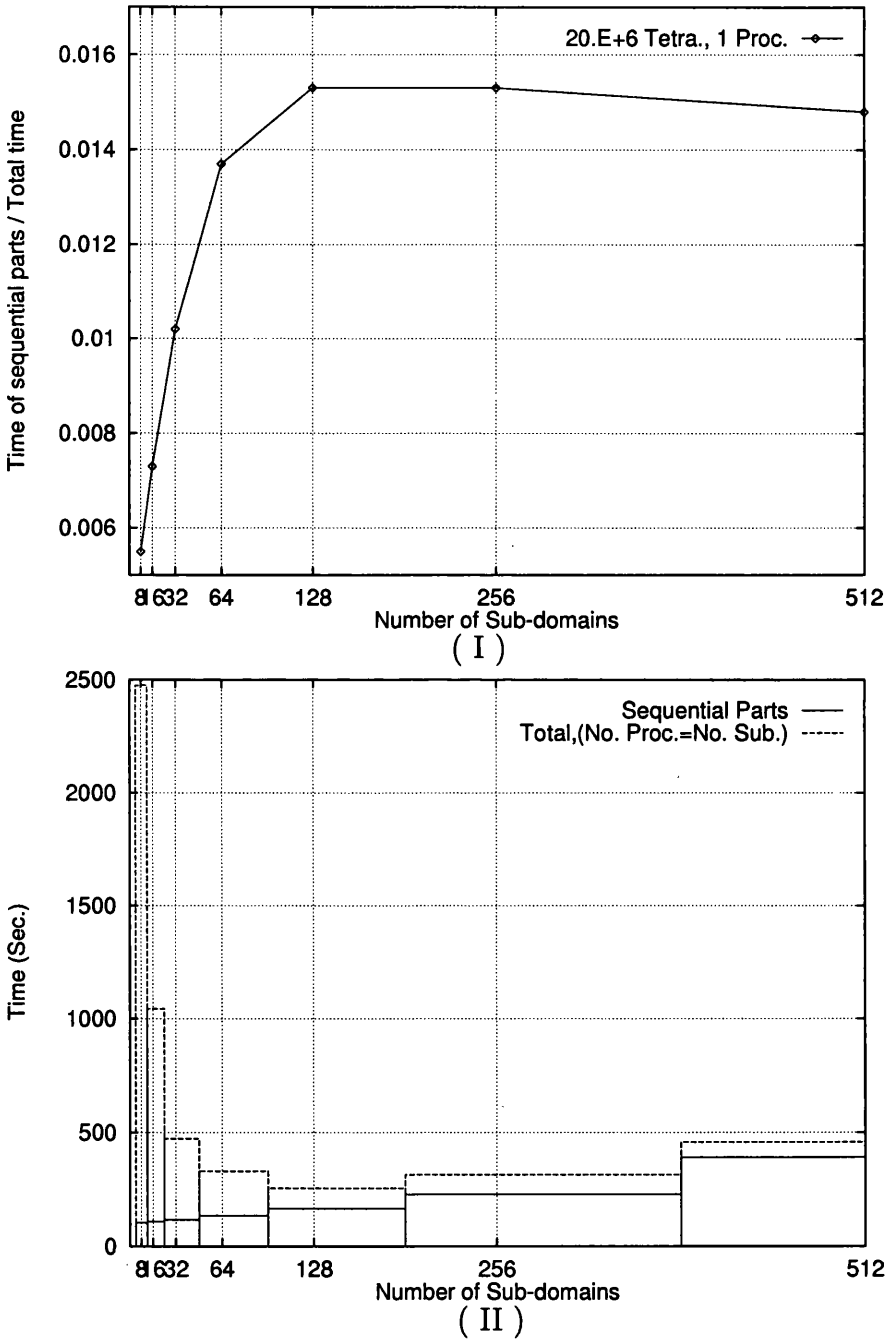


Figure 6.30: Time required by the sequential parts in respect to the total time of the parallel run. Grid consists of 20.0E+6 tetrahedra and partitioned into different number of sub-domains in the range [8 - 512]. (I) Exact timing using one processor, (II) projected timing by assuming number of sub-domains is equal to number of processors.

The cost of partitioning the domain, in the direct decomposition method, and establishing the inter-domain communication data, in both methods, is problem independent. In other words, the very small percentage of the sequential time presented must remain the same when more complex geometries are considered. However, it has been observed that the most expensive sequential operation is writing the inter-domain communication output files on the disc by the Manager. Actually, a minor modification to the program can transfer this task to the Workers, but introducing some extra interprocessor communications will be inevitable.

6.3.5 Inter-Processor Communication

The speed of transferring data among processors, particularly when it is done over a network, is still considered as a very slow procedure compared to the speed of processing. In the 'idealised network' the communication cost is independent of the processors location and other network traffic, but it does depend on the message length [40]. Also, there is always an overhead cost associated with every sending/receiving operation whatever the contents of the message is. On the other hand, parallel algorithms designed to operate on distributed memory model rely heavily on transferring data among processors. Therefore, minimising the size and the frequency of the transfered data is a very crucial point for such parallel algorithms performance.

The inter-processor communication operations are discussed extensively in Section 4.2, but some of their features, from the performance point of view, are highlighted herein.
The number of sending/receiving operations is always kept to a minimum as:
- Whenever there is a set of data to be transfered more than once, from the Manager to the Workers or vice versa, it is then transfered once only and stored in the local memory of each Worker (or the Manager).
- Whenever there is a noncontiguous set of data, a 'grouping' operation is carried out before the sending.
Thus, in both cases, one sending/receiving operation is employed instead of several one at least. Also, the size of the transfered data is kept to a minimum as well, by always extracting the required data only and imposing a new compact numbering scheme. Obviously, these features provide an optimised communication strategy, which in turn gives a better performance of the parallel programs.

The two examples presented in Figure 6.31 show that the time needed in the inter-processor communication operations is extremely small comparing to the total time in any run. So, even if the communication speed becomes much slower than it is on the used platform, the performance of the program overall mustn't be influenced badly. Graph(I) in Figure 6.31 illustrates how the time

spent on communication varies with the grid size, whilst Graph(II) illustrates that with the number of sub-domains. Although the communication cost may grow substantially when a high number of sub-domains is used, it is always negligible in comparison to the cost of the entire procedure. Considering the noticeable increase associated with the 512 sub-domains case in Graph(II), it is appropriate to highlight that such a number of sub-domains is considered to be rather high for grids with size as in the presented ones. It is also essential to mention that the communication cost is independent from the complexity of the gridded configuration.

6.4 A Comprehensive Parallel Processing Environment

The continuous increase in the complexity of applications is already pressing for a combined solution that promotes an extensive use of parallel processing technology. An environment that provides a highly efficient and scalable parallel algorithm for every procedure involved in the design-analysis loop (i.e. geometrical and topological definition, grid generation, numerical analysis and post processing) is very likely to be the main stream demand in the very near future [89, 95, 146, 144].

Although, developing such an environment may sound a straightforward task of software integration, in practice, achieving advanced level of portability, scalability and efficiency can be very challenging. Employing highly sophisticated programming tools enhanced by object-oriented database may become essential [116]. Nevertheless, a simplified form of integration can still be an effective alternative; particularly when the data that transfered among different modules is well established and very unlikely to be changed.

In fact, such a comprehensive environment of parallel computational engineering is already been under development in the Civil Engineering Department at the University of Wales Swansea, into which the program developed in this research has been integrated. It is called the PSUE (Parallel Simulation User Environment), developed by a research group under the supervision of professors N.P. Weatherill, O. Hassan and K. Morgan. The PSUE contains some basic CAD functionalities, parallel CFD and CEM algorithms, a limited parallel grid enrichment algorithm and an advanced parallel visualisation tools [144, 149]. Obviously, discussing the PSUE, or the topic of parallel environment in general, is beyond the objective of this thesis. However, it is still appropriate to demonstrate the interaction between the parallel grid generator and other various models inside the PSUE.

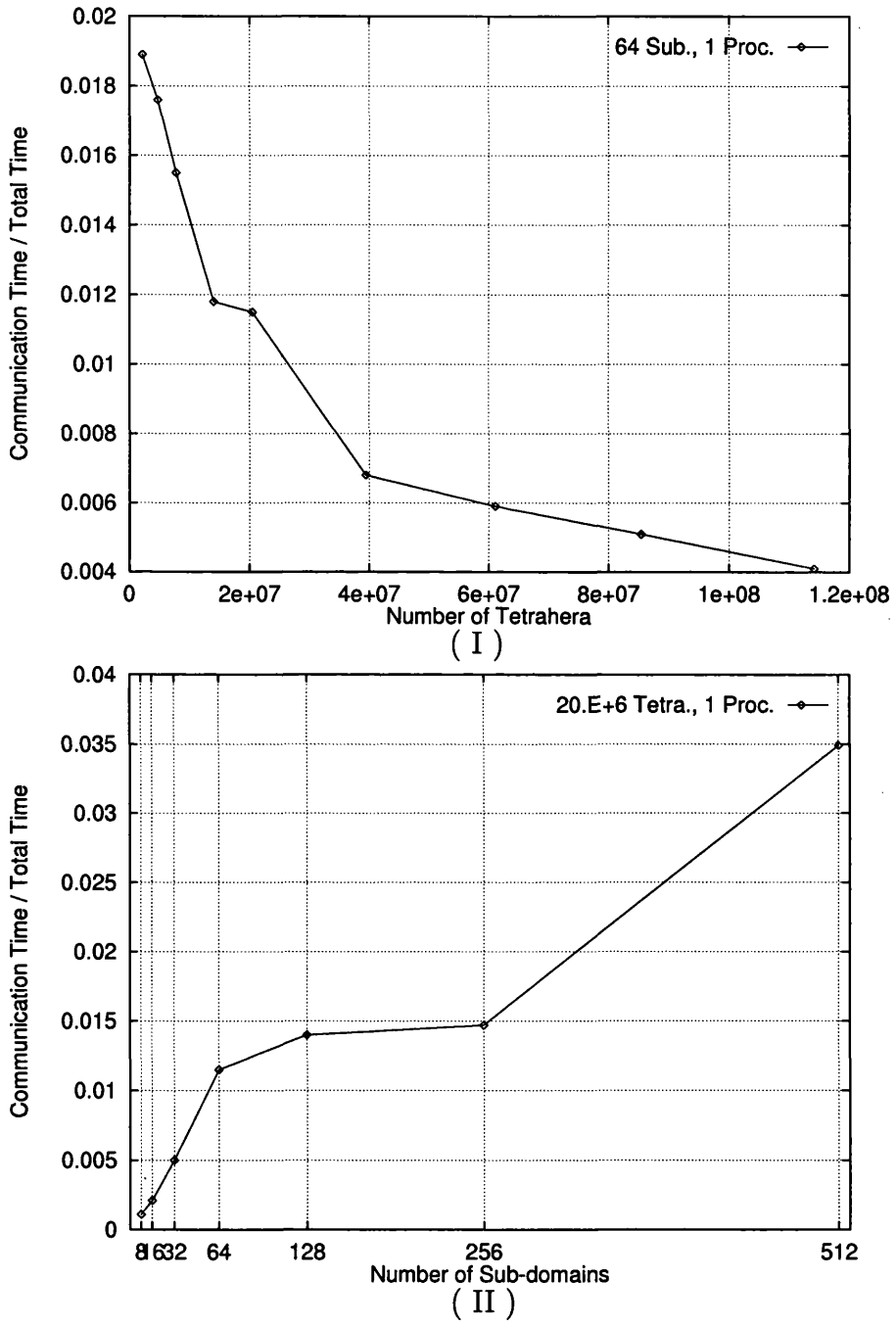


Figure 6.31: Time required by the communication operations in respect to the total time of the parallel run using a single processor. (I) Grid size in the range [2E+6 - 114E+6] tetrahedra and partitioned into 64 sub-domains, (II) Grid consists of 20.0E+6 tetrahedra and partitioned into [8 - 512] sub-domains.

Parallel simulation

Two different simulators are available in the PSUE; the first one is for fluid dynamics (i.e. Euler and Navier-Stokes equations) and the second is for electromagnetics (i.e. Maxwell's equations). Both solvers are based on the Finite Element method with Galerkin approach. An explicit finite difference procedure is used to discretize the time dimension [149]. The overall data structure is edge-based, which requires the edges to be extracted from the 'element based' grid as it is generated and stored by the parallel grid generator.

Having the grid already been distributed onto sub-domains, and the inter-domain communication established, the parallel simulators would require no further preparations. In fact, whilst the CEM solver accepts the same data format as produced by the parallel grid generator, there is still an incompatibility issue with the CFD solver. Hence, the option of 'building one global volume grid' is certainly needed such that a compatible grid partitioner can be employed. Unfortunately, due to some out of hands circumstances no electromagnetic simulation could be presented in this thesis, however, some previous work can be seen in [91, 94, 95]. Alternatively, we present the simulation of two different fluid dynamic problems. The work has been carried out using a CFD solver developed by Sorensen and uses an agglomerated multigrid method [119].

Simulating aerodynamic in a configuration of four military aircraft: Details of this configuration with a grid generated in 32 sub-domains have been presented in section 6.2.2, see Figures 6.9 and 6.10. Another grid that consists of 61,586,153 tetrahedra and 10,569,214 points has been generated in 16 sub-domains employing one Worker on the same SGI Challenge machine. A cross section in the volume grid of one the sub-domains is presented in Figure 6.32, with two different close ups. In order to improve the performance of the simulator, grid quality enhancement technique was applied on individual sub-domains before constructing the global volume grid, [58]. Statistics of the minimum dihedral angle per element, collected after the quality enhancement procedure been implemented, show that only 14 elements have a dihedral angle of less than 10 degrees.

The global volume grid has been re-partitioned into 16 sub-domains by Metis [75, 74] whilst operating sequentially on an SGI Onyx shared memory machine, which has 32 processors and 64 GBytes of memory. The simulation was carried out in parallel employing 16 processors on the same machine. Considering the airflow as inviscid and compressible, with the assumption of Mach number as 1.8 and zero angle of attack, the total time needed by the simulator was about 20 hours. Results are presented in a set of figures, each figure has a few pictures of the pressure distribution contours. See Figure 6.33 for the contours on the aircraft surface, whilst for a set of planar cuts inside the wind tunnel

and perpendicular on the main axis see Figures 6.34, 6.35 and 6.36.

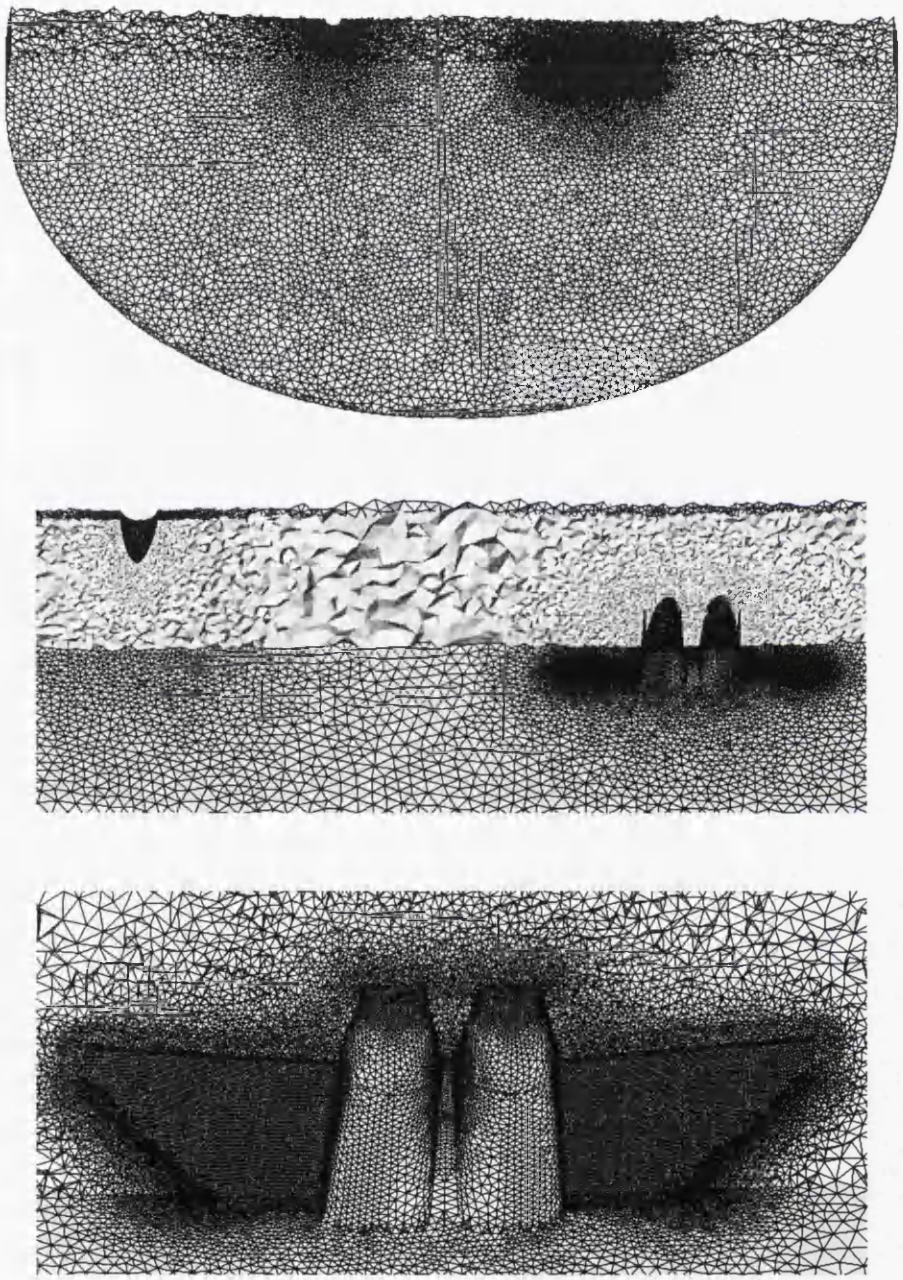
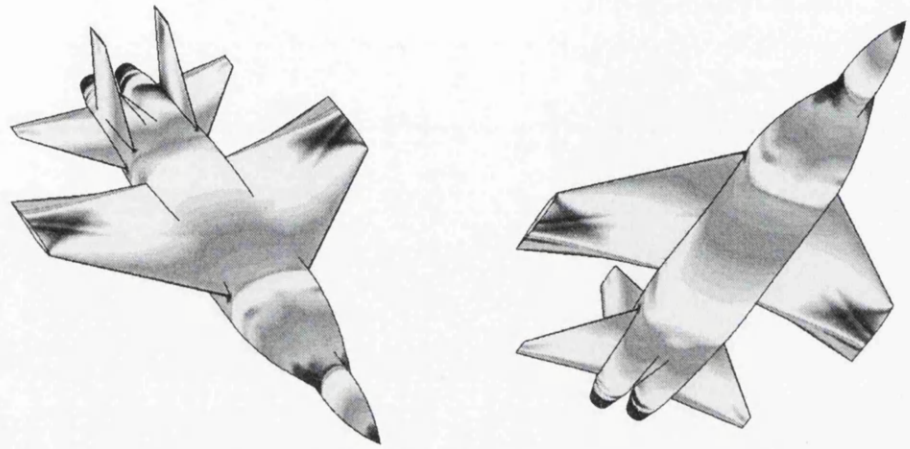
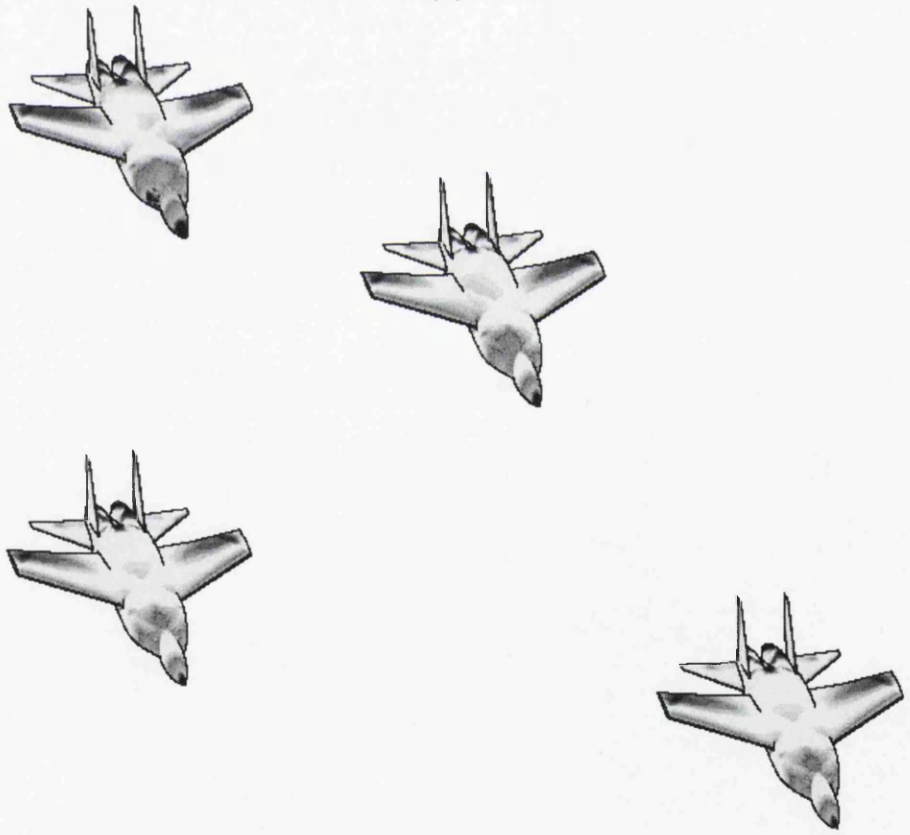


Figure 6.32: Details from a Sub-domain volume grid, total number of elements is 61,586,153.



(*I*)



(*II*)

Figure 6.33: Four aircraft configuration example (cont.). A pressure contour on the aircraft surface, (I) top and bottom view of one aircraft, (II) top view of the four aircraft.

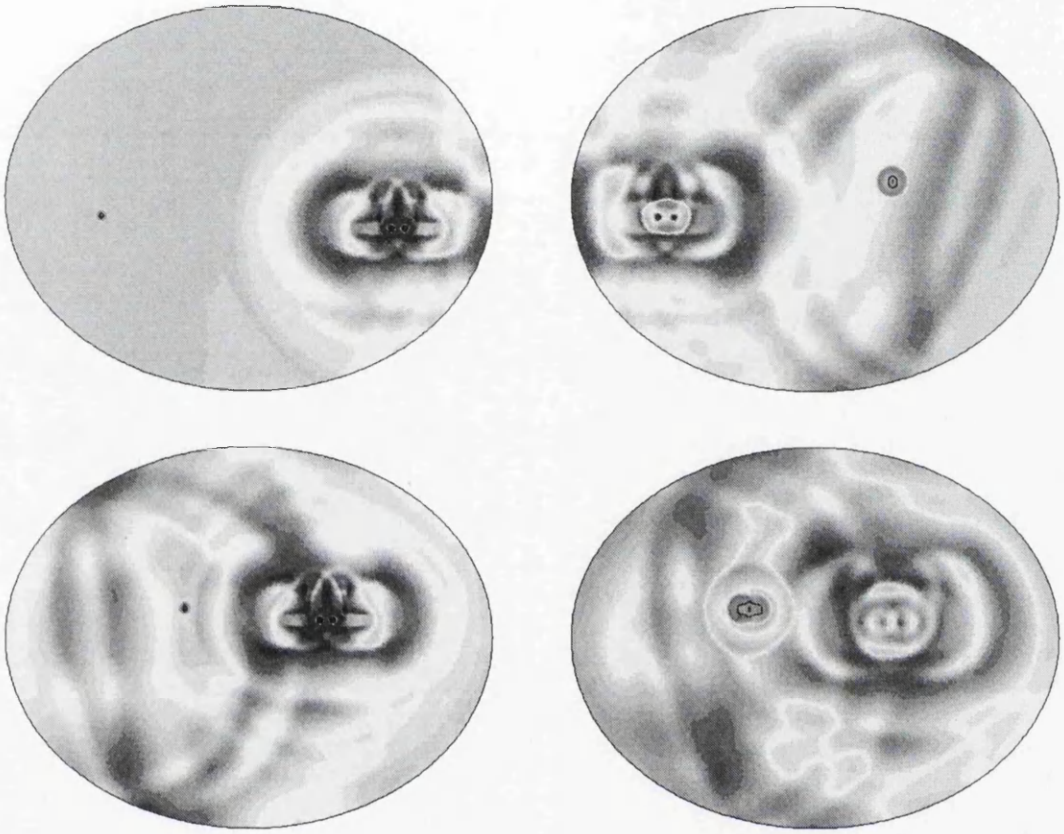


Figure 6.34: Pressure contours on a set of planar cuts that perpendicular to the main axis of the tunnel.

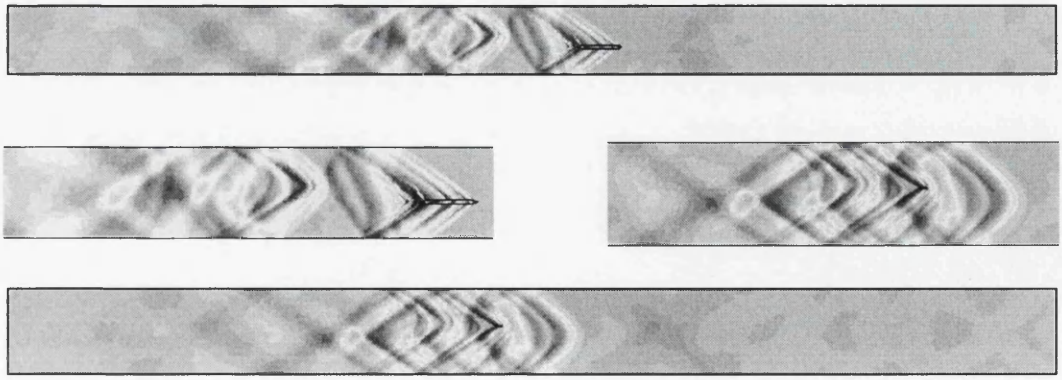


Figure 6.35: Pressure contours two planar cuts parallel to the main axis (side view) of the tunnel.

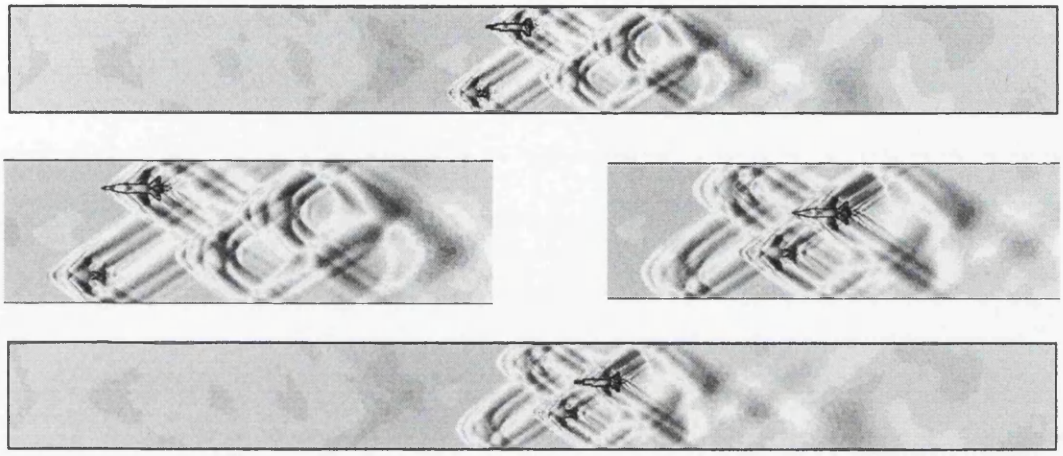
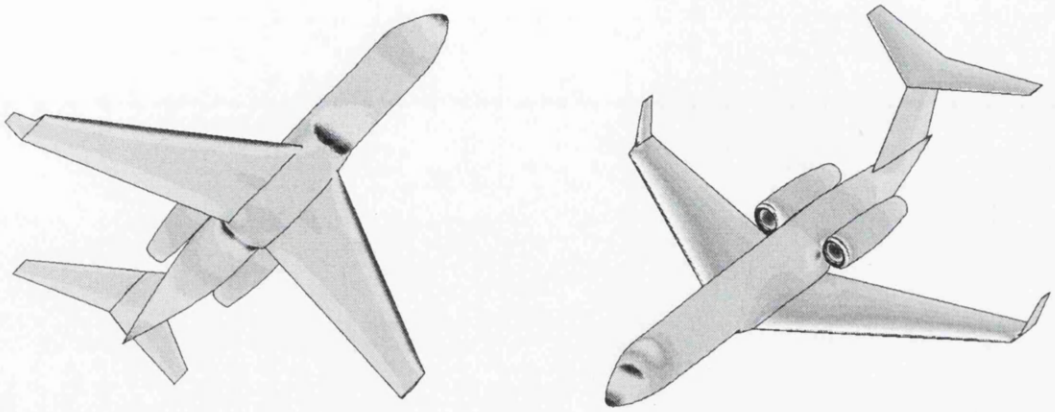


Figure 6.36: Pressure contours two planar cuts parallel to the main axis (top view) of the tunnel.

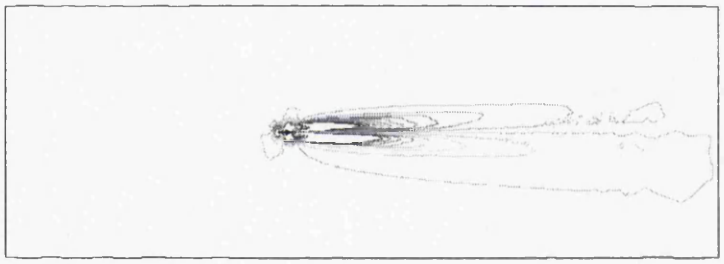
Vortex dynamic behind a civilian aircraft

Shortening the gap between airplanes queuing over an airport space for landing has a massive commercial interest, and it seems that aerospace industry has already acknowledged the issue. Obviously, a comprehensive understanding of the vortices dynamic and how they are formed behind aircrafts is a very essential part in addressing the problem. However, our main interest is to demonstrate the implementation of grids generated by the parallel grid generator developed in this research. The grid used in this example has been presented earlier in Section 6.2.2, see the Figures from 6.13 to 6.18 and Table 6.8. An interesting observation in this grid is that the total number of elements without the 'extra' sources (i.e. the ones stretched behind the wings tip) would be about 6 million tetrahedra only, which means that more than 57 million tetrahedra are distributed along the vertices path in order to capture an accurate solution.

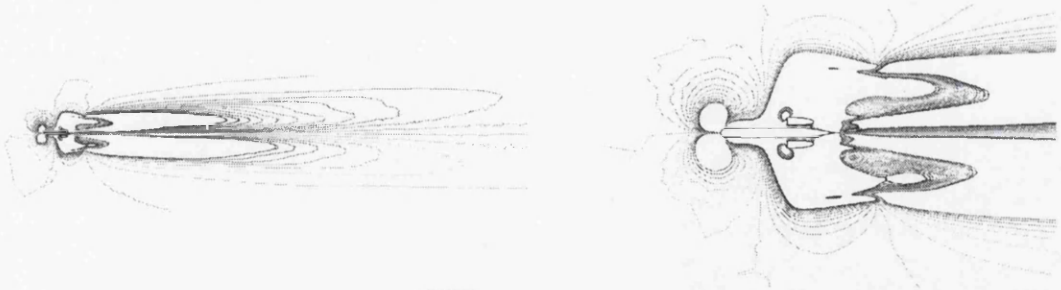
Grid quality enhancement algorithm has been applied on individual sub-domains before they were merged together to form the global volume grid, which has subsequently been partitioned into 16 sub-domains using Metis. The two operations of constructing and partitioning the global volume grid have been carried out sequentially on the SGI Onyx machine mentioned before. The same solver as in the example above has been used to carry out the analysis, with the angle of attack equals 5 and Mach number equals 0.85. Total run time for the solver was about 18 hours using 16 processors on the Onyx machine. Distribution of pressure on the entire aircraft surface is presented in the top images in Figure 6.37. Whilst, various images of the velocity distribution contours are presented in the Figures 6.38 and 6.39.



(I)



(II)



(III)

Figure 6.37: Contours of the pressure distribution on the airplane surface, (I), and line contours of the velocity (Y component) on a planar cross section parallel to the aircraft at the edge of the wing tips in (II) with two different closeups in (III).

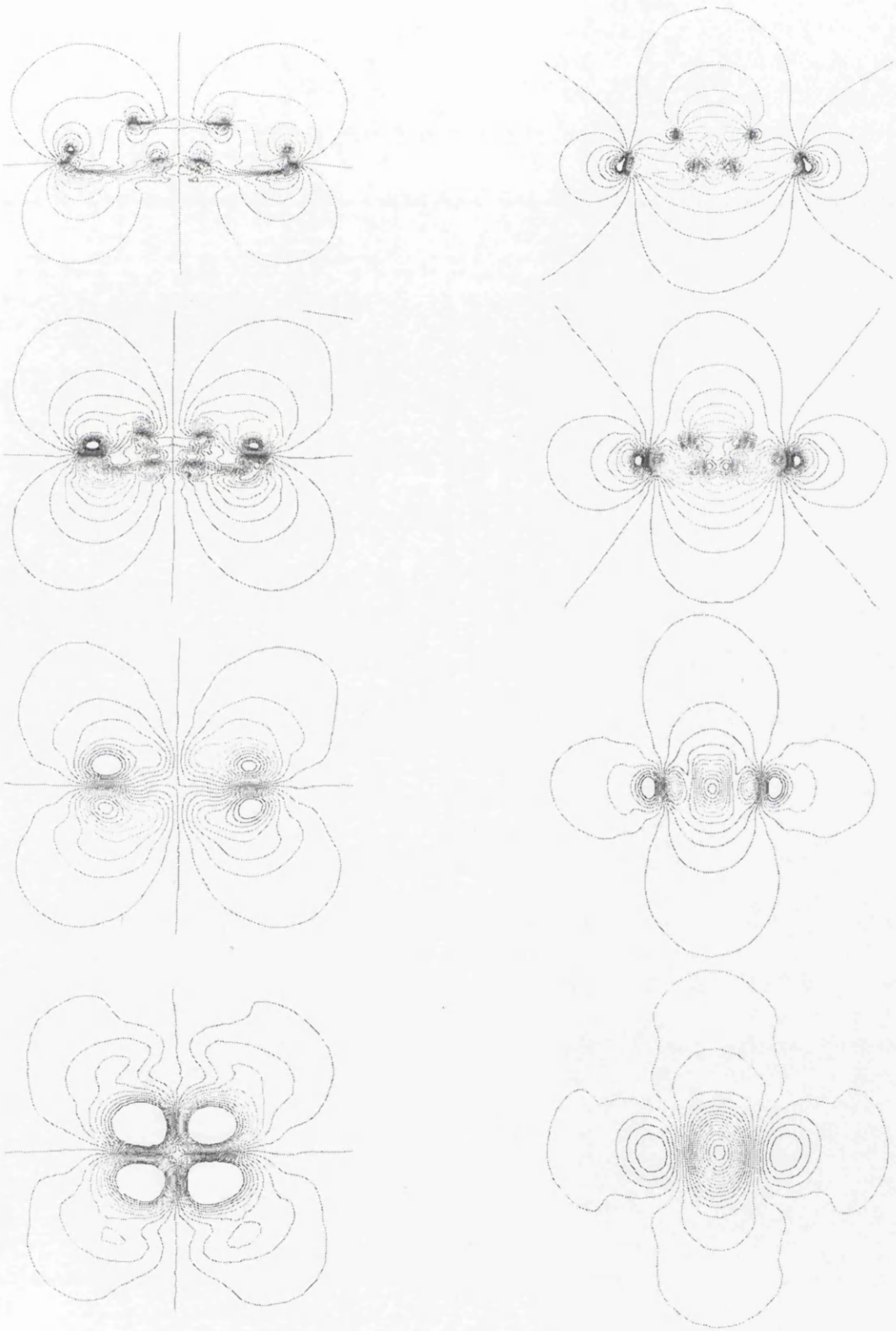
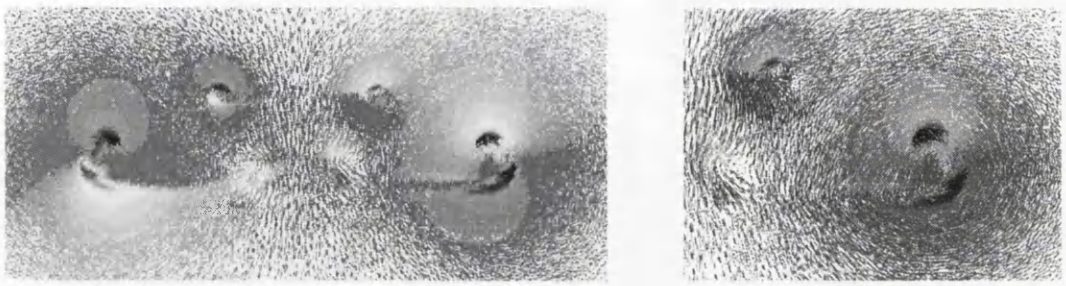
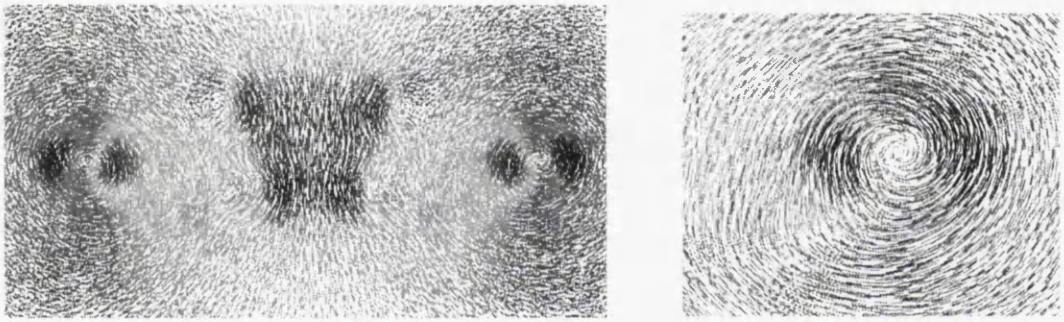


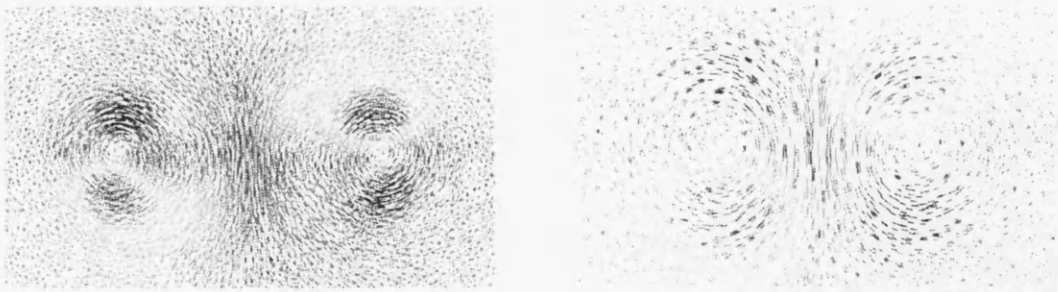
Figure 6.38: Line contours of two velocity components (Y direction on LHS and Z direction on RHS) on a set of planar cross sections along the main axis. All the cuts are located behind the airplane, the nearest plane is on the top.



(I)



(II)



(III)

Figure 6.39: The velocity vector at the same cross sections presented in Figure 6.38, (I) and (II) present the first two sections with a close up of each on the RHS, whilst (III) present the third (LHS) and fourth (RHS) cross section.

Parallel simulation with grid adaptivity

In general, sequential simulators enhanced by grid adaptation techniques have proved to be very useful as more accurate solutions can be achieved rather efficiently [58, 101, 140]. The same type of grid adaptation techniques can also be used by parallel simulators, though some extra care is needed in the implementation procedure. Given a solution on an initial grid *h-refinement* technique can be employed *inside* sub-domains independently, whilst, any modifications of elements that associated with the inter-domain boundaries have to be carried out in co-ordination between the adjacent sub-domains. Obviously, this requires extra inter-process communication to be introduced during the simulation. Such that inter-domain communication tables [7] need to be updated after every iteration of grid adaptivity.

Achieving a high level of accuracy and smoothness in interpolating new points on the boundary elements requires access to the geometrical description of boundary surfaces. Thus, an easy and efficient way to access the geometrical description of surfaces under the sub-domains boundary grid should be provided. In fact, the parallel grid generator reads in such information at the same time when the original boundary triangular grid is read. The surface number associated with each triangle on the original boundary is maintained, using the global numbering system, throughout the entire partitioning and gridding procedures.

An illustrated example for the grid refinement procedure impact is presented. The problem involves aerodynamic analysis of a military aircraft configuration. An initial grid is generated in 8 sub-domains operating on two processors in the SGI Challenge machine; see Figure 6.40 for some details of a typical grid in one of the sub-domains and its internal boundary. The total number of elements in the initial grid is 6,345,709 tetrahedra, whilst it is 18,020,126 in the final grid. Typical contours of the pressure distribution on the aircraft surface are presented in Figure 6.41, differences between the two solutions (i.e. associated with the initial grid or refined grid) can be seen. Typical examples of the grid refinement procedure impact on the surface grid can be seen in (*III*) in Figure 6.41.

[7] See section 5.2.2 for more information on these tables.

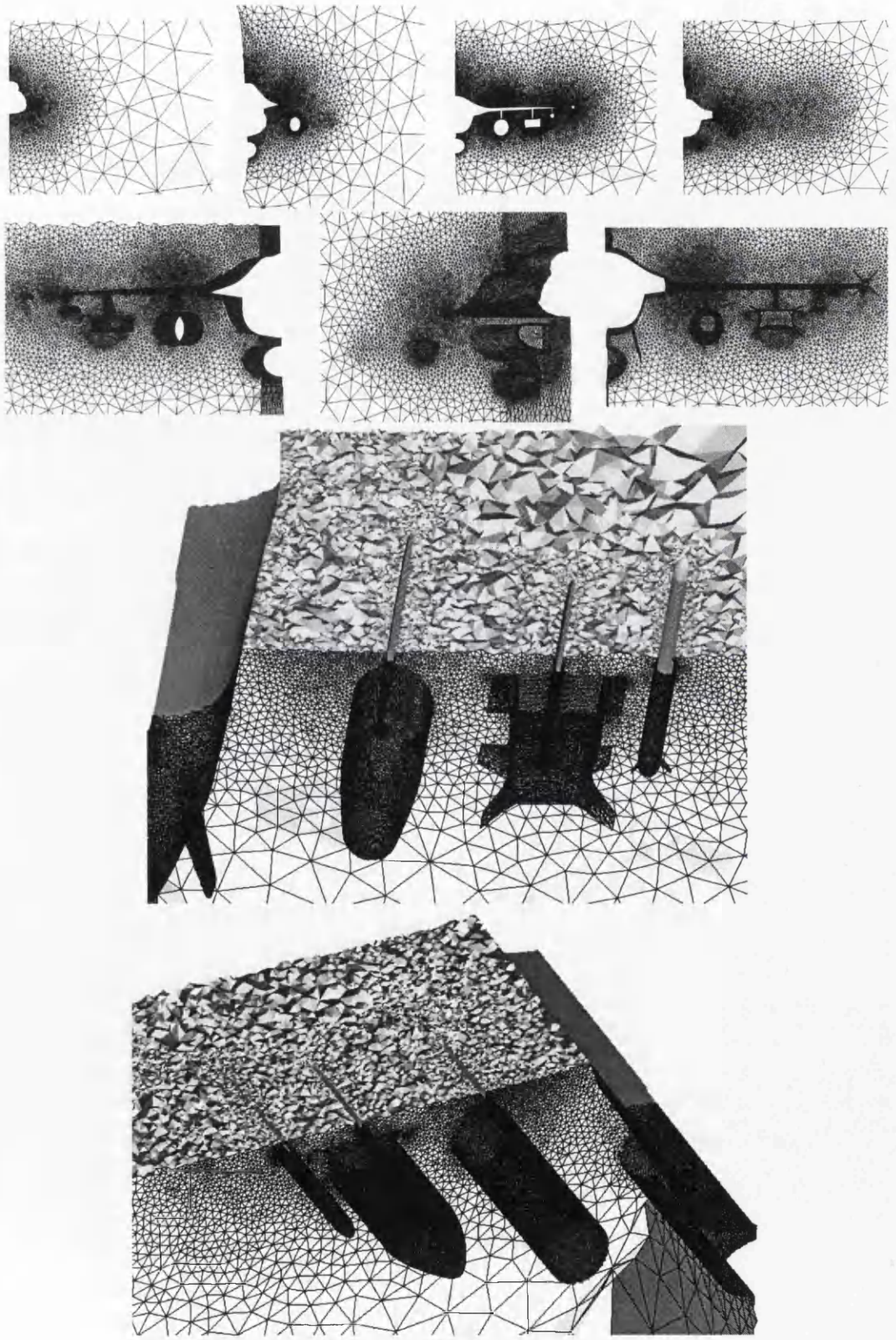
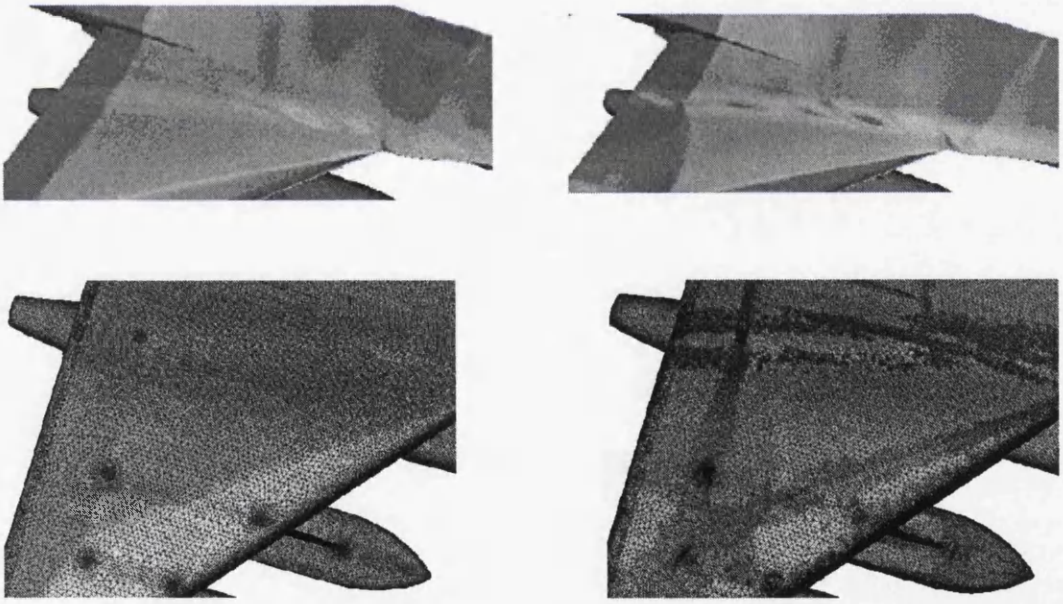


Figure 6.40: Details of the F16 example, surface grid on the internal boundary with cuts inside some of the sub-domains grid.

(I)



(II)

Figure 6.41: Simulation of the airflow around a military aircraft utilising the grid adaptivity technology. Pressure distribution contour in (I), and a comparision between the solution obtained on the initial grid (left) and the refined grid (right) in (II). Impact of the refinement procedure on the boundary triangular grid can be seen in the second row in (II).

Parallel Visualisation

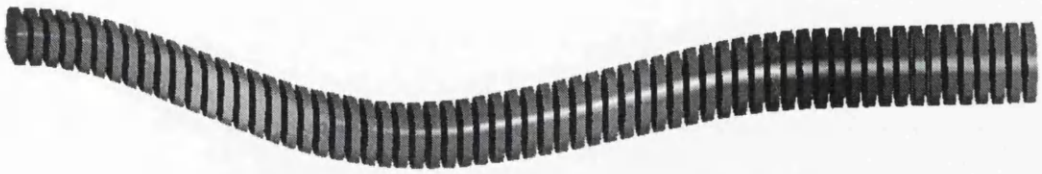
The interest in using parallel processing technology with volume rendering algorithms has grown considerably during the last decade,[87, 88]. Advanced parallel visualization tools known as ViPar have been designed and integrated into the PSUE [71, 149]. The core idea of ViPar is that all of the searching and computationally expensive tasks are performed on the server computer and only the data required to be rendered sent back to the end user workstation. ViPar can read grids directly as generated and stored by the parallel grid generator, i.e. the grid is already distributed on sub-domains and all inter-domain communication data been established. It also exploits other available information that associated with the sub-domains boundary grid the in order to speed up data search and inter-process communication operations. ViPar uses the SPMD parallel programming model operating on distributed memory architecture with Manager/Workers mechanism. The Manager is always associated with the end user graphic workstation, whilst, all Workers are distributed on the server processors.

Typical outputs of ViPar for a grid in the order of 115 million tetrahedra (i.e. the grid of the aircraft engine duct, which is presented earlier this chapter see Figure 6.7 and Table 6.3) is presented in Figure 6.42. A general overview of the sub-domains surface grid is presented in (I), a close up of a planar cut in the volume grid of a few sub-domains around the fan is presented in (II) and a planar cut in the global volume grid with the surface grid on the fan in (III).

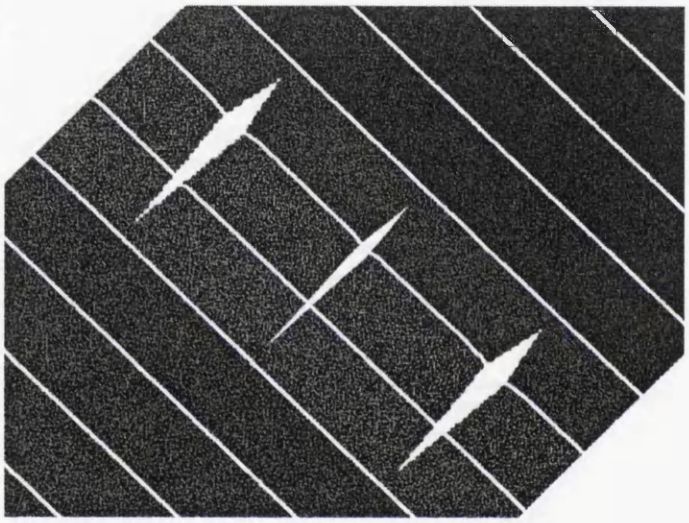
6.5 Concluding Remarks

A flowchart of the general parallel framework has been presented very early in this chapter. Various options available in the framework (i.e. the basic grid, load redistribution and the construction of one global grid) have been addressed. The option of building one global grid has been discussed in detail. A layout of the general procedure has been discussed, an example of a grid generated in eight sub-domains and then assembled in one global grid has been presented. Implementing this option in order to carry out *local partitioning of large sub-domains* has also been illustrated in a greate detail. A number of tetrahedral grids of realistic engineering problems have been presented. Some of these grids are in the order of more than 100 million elements, furthermore, all presented grids have been generated on a platform has only 512 MBytes of available memory.

An extensive numerical analysis of the program performance and scalability has been presented. Number of processors, number of sub-domains and prob-



(I)



(II)

(III)

Figure 6.42: Different views of the Duct grid, see Figure 6.7 and Table 6.3, generated by the ViPar tools. A general overview in (I), a close up of a planar cut in volume grid of a few sub-domains around the fan in (II) and a planar cut in the global volume grid with the surface grid on the fan in (III).

lem size, have been considered as the main parameters that would influence such analysis. A set of 'Cases', that defined by allowing these parameters to change individually or combined, has been established and used systematically during the inspection of various measurements. The inspected measurements included: speedup, efficiency, scalability with problem size and scalability with resources. Sequentially processed tasks have been identified and Amdahl's law has been examined. The impact of the time spent in the inter-processor communications on the performance overall has also been explored.

The scalability of the indirect decomposition method is seen to be questionable, mainly because of the limitations introduced by the construction of the initial volume grid. Whilst the direct decomposition method suffers from the limitations on total number of sub-domains (i.e. planar cuts) that can be created along one axis. The enhancement by the local repartitioning procedure has proved to be effective, but an octree based decomposition with better initial load distribution must be sought. In general, this chapter has demonstrated that the developed framework can provide an efficient scalable parallel algorithm for unstructured grid generation. Also, most interestingly, generating massive grids can be accomplished by employing a single processor machine only.

A comprehensive computational engineering environment that adopts the parallel processing technology for every algorithm involved in the design-analysis cycle has been introduced very briefly. The integration of the programs developed in this research into such an environment has been highlighted. The use of some of the generated grids by other algorithms has also been illustrated. Two different simulations of typical fluid dynamic problems have been presented, and another simulation enhanced by grid adaptivity procedure has been shown.

Chapter 7

Conclusion and Recommendation for Further Research

Issues associated with using sequential grid generators for constructing large-scale unstructured grids have been addressed very early in this thesis (see section 1.2). It was stated: "relying on the traditional sequential grid generation methods only to generate large scale grids can not provide a practical solution .... alternative approaches had to be investigated, aiming to .... preclude the bottle neck of memory requirements." Subsequently, the ultimate objective of this research was identified as: *to develop a 'tool' by which large-scale unstructured grids for realistic engineering problems can be generated efficiently on any parallel computer platform.*

This thesis has addressed two different approaches for generating grids by using the parallel processing technology. First approach depends on parallelising an existing sequential algorithm directly, whilst the second approach is based upon a geometrical partitioning concept, where the computational domain is subdivided into a number of sub-domains which are then gridded independently in parallel.

Three different categories have been identified under the geometrical partitioning approach, see section 2.4.2 page 29. The strategy adopted in this thesis falls under the 'pre-gridding' category in which all internal boundaries are gridded prior to the construction of the closed boundary of individual sub-domains. It has been proved that algorithms in this category have a number of advantages to algorithms in the other two categories:

- Much larger grids can be generated, since algorithms under the other two categories would have to deal with the volume grid of the entire domain in one form or another during the generation procedure.

- There is more flexibility to enhance the quality of surface grids on the internal boundaries, and subsequently the quality of the overall volume grid.

- Cost of inter-processor communications in this category is almost negligible.

- There is a great deal of flexibility to integrate different procedures at certain steps of the overall algorithm

Two different methods (namely the indirect decomposition method and the direct decomposition method) have been investigated in this research. The key concept in the former method is to partition the computational domain by exploiting an 'initial grid', which is constructed using the original boundary points only. The grid partitioning procedure is carried out using a greedy-type algorithm, employing two different options as partitioning criterion (i.e. equal volume per sub-domain and based on an estimate of total number of elements expected per sub-domain. Nevertheless, this method in general has failed to demonstrate a steady and robust performance in gridding complex geometries, neither has it proved to be a scalable and efficient method for generating grids in parallel. The thesis has identified a number of problems with this method:

- The overall shape of sub-domain boundaries is usually very complicated, which makes maintaining the boundary integrity after Delaunay triangulation rather difficult and time consuming.

- The demand for generating an initial tetrahedral grid can become a real bottle neck in the process. Scalability and efficiency of the algorithm overall can get badly effected by such an inevitable sequential step.

- Achieving an acceptable level of workload balance among sub-domains at the domain decomposition step has proved to be very difficult if not impossible.

- The inter-domain communication cost is very likely to be high as a result of the extreme shape of the sub-domain boundary.

Therefore, a method that can address such issues was sought leading to the development of the direct decomposition method. In this method, the enclosed volume is subdivided *directly* by applying a set of parallel planar cuts distributed along one axis. Locations of the cutting planes are determined based on a criterion chosen by the user (i.e. equal spacing, equal number of triangles and predefined explicitly). Other ways for carrying out the decomposition

procedure using planar cuts in different forms such as a Cartesian network or by employing an octree decomposition procedure were also highlighted.

This thesis has demonstrated that the direct decomposition method, unlike the indirect decomposition method, will always produce sub-domains of more 'regular' shapes that have smooth and high quality surface grids. It has been illustrated that the direct decomposition method can be a reliable, scalable and efficient technique for generating tetrahedral grids in parallel. In general, a set of advantages associated with the direct decomposition method can be identified:

- It is a much more reliable method, mainly due to the nature of the sub-domain boundary and the high quality of its surface grids.
- It is a highly scalable method, since there is no sequential procedure such as generating the initial grid in the indirect decomposition method.
- The element quality on the internal boundary surfaces, and subsequently in the overall volume grid, has proved to be much better in comparison to the indirect decomposition method.
- The initial distribution of the workload is much more balanced, and there is a great potential for further improvements in the future.
- The inter-domain communication cost is low, and it can be reduced significantly if more sophisticated domain decomposition techniques is used.
- There is a great chance to minimize, or eliminate, the cost associated with the load redistribution step as a post-processing procedure.

The direct decomposition method, mainly when it is implemented with uni-directional cuts, may show some limitations in terms of the total number of sub-domains that can be created. Obviously, such an issue can be addressed by implementing a multi-directional cuts procedure or another more advanced domain decomposition algorithm. However, the enhancement introduced in this thesis by integrating a local re-partitioning procedure, which repartitions individual sub-domains independently, has proved to be very effective.

The implementation of the parallelisation strategy has been presented using one general algorithm that represents both of indirect and direct decomposition methods. A unique computational framework has been presented. This framework adopts the Message Passing Library (MPL) model and uses the Single Program Multiple Data (SPMD) structure with a Manager/ Workers mechanism. A special technique termed the Dynamic Parallel Processing (DPP) has been integrated into the template in order to enhance its flexibility, such

that processing an arbitrary number of tasks on an arbitrary number of processors became always possible. It has been demonstrated that this framework with the DPP technique can always provide an efficient and scalable algorithm for generating unstructured grids in parallel. Also, most interestingly, it was shown that generating massive grids can be accomplished by employing a single processor machine only.

A number of 'post-processing' algorithms have been presented. Whilst some of them were developed in respond to practical requirements, such as the construction of one global volume grid by merging all sub-domains; others were more essential, such as the construction of inter-domain communication tables. The load balancing issue has been discussed thoroughly in this thesis. The development of a new 'purpose-built' load redistribution algorithm was justified based on the fact that using off-the-shelf algorithm might become very problematic with such large-scale grids. Though the developed algorithm has proved to be very effective in achieving highly balanced element distribution, it is still considered as inefficient procedure. However, it has been shown that this algorithm has a great potential for further improvements and a well acceptable performance is achievable. An attempt to improve the element quality has been made by developing a point relaxation algorithm that operates on the internal boundary surface grids. The thesis has concluded not to recommend using this algorithm due to the very little improvements it provides.

To sum up, apparently, the framework developed in this research has proved to be a very effective tool for generating large-scale tetrahedral grids. Grids of realistic engineering problems and to the order of 115 million elements (generated using one processor on an SGI Challenge machine with 512 MBytes of shared memory) were presented.

Recommendation for Further Research

Research presented in this thesis has already been subject to further development, see the PhD thesis by Larwood in reference [154]. In fact, some of the following recommendations have already been investigated, however, the parallel grid generation field remains a young and active area of research and there is still much scope of further developments. "Because the development of efficient scalable parallel techniques takes much more than their sequential counterparts, it may take a while before parallel grid generation comes to state of maturity." [22].

- The issue of achieving highly balanced workload distribution at the domain decomposition step must be given a priority. Options such as using

algorithms based on Recursive Cartesian Bisection or Octree partitioning should be explored. Considering the effects of the grid point spacing parameters is crucial.

- More sophisticated approach to the domain decomposition issue can be investigated. In such an approach, the domain decomposition procedure would consider the inter-domain communication cost as another major criterion (i.e. in addition to the workload distribution).

- The task of surface grid generation on the internal boundary will always form a significant part of the overall algorithm, therefore, it must always be carried out in parallel. Sequential processing of such a task would cause a serious problem to the algorithm overall scalability.

- The Dynamic Parallel Processing technique has proved to be extremely effective, therefore, it is strongly recommend to be maintained in all future development.

- With the growing increase in the size of surface grids associated with large-scale volume grids, it is very likely that the parallelisation of the surface grid generation procedure (i.e. operating on the original boundary) will become essential.

- There is a great potential to modularise the general framework such that it can accept various types of grid generators with very minimum, if any at all, interaction with the original program

- With such large-scale grids, and in orderer to maintain the high scalability of the framework, it is strongly recommended to keep the 'post-processing' type of algorithms in general to the very minimum.

Appendix A

Unstructured Grid Generation Using Delaunay Triangulation

"It is a remarkable fact that seemingly simple concepts can often lead to whole new fields of research and find extensive applications in many diverse areas. This phenomenon is well illustrated by the Voronoi diagram (1908, [130]) and its dual the Delaunay triangulation (1934, [24]). Though formulated early in the twentieth century long before the rise of scientific computing, these fundamental geometric ideas have recently found a wealth of applications.....", [5]. In fact, this can be taken back even further to the nineteenth century. The basic idea was proposed by Dirichlet in (1850, [26]), and subsequently rediscovered by various workers in many fields and described under variety of names such as Voronoi diagram in the computational geometry area. In 1981 Bowyer [12] and Watson [135] have independently developed an algorithm for computing the Delaunay triangulation. Most of the Delaunay based grid generators that are available nowadays employ one of these algorithms. However, although the Delaunay triangulation and associated Voronoi diagram had been used as a natural setting for calculations involving irregular spaced points, the real implementation of it as an explicit method of generating grids for complex shapes was not known until a bit later in the 1980s [3, 136, 141]. Since its early days the method has proved to be very efficient and flexible, and it became very popular, (see [86] for extended bibliography). However, although a number of advanced algorithms for Delaunay grid generation have already been developed and matured enough by now, the method still attracts a considrable attention for further research [5, 11, 50, 143, 58].

Given a set of points in the plane, the Dirichlet tessellation is constructed which assigns to each point a territory that is the area of the plane closer to that point than to any other point in the set. This tessellation of a closed domain results in a set of non-overlapping convex polygons, called a Voronoi diagram. The boundaries of such polygons form the perpendicular bisectors

between each point and its immediate neighbours. If every two points with a common boundary are connected by a straight line then a triangulation, known as Delaunay triangulation, is obtained. This definition readily extends into three dimensions where tetrahedra are formed.

Obviously, the Delaunay-Voronoi dual construction is applicable on a *given* set of points only, thus, from the grid generation viewpoint there is still a demand for another technique that produces the grid points. The other issue needs to be addressed in conjunction with the Delaunay triangulation is how to ensure that the resulting triangulation is boundary confirming, which means that the initial boundary is preserved. Of course, it is beyond our intention to discuss in great detail all these different concepts. However, in order to provide the reader with some of the basic knowledge that directly related to the work of this research, two of the topics mentioned above (i.e. the Delaunay triangulation and point creation) are reviewed briefly while for the third issue (i.e. boundary confirming) the reader is advised to consult [3, 56, 143] [1].

A.1 The Delaunay Triangulation

Delaunay-Voronoi Geometrical Structure

If a set of points is denoted by $\{P_i\}$, then the Voronoi regions $\{V_i\}$ can be defined as

$$\{V_i\} = \{P : \|p - P_i\| < \|p - P_j\|, \forall j \neq i\};$$

i.e., the Voronoi regions $\{V_i\}$ is the set of all points P that are closer to P_i to any other point. The sum of all points p forms a Voronoi polygon, see (I) and (II) in Figure A.1.

The definition above is valid for n dimensional space. Apparently, in the two dimensional space a line segment of a Voronoi polygon must be equidistant from the two points that it separates and is thus a segment of the perpendicular bisector of the line joining these two points. If all point pairs which have a segment of polygon in common are joined by straight lines, the result is a triangulation of the convex hull of the data points. This triangulation is known as the Delaunay triangulation of the set of points $\{P_i\}$, see (III) in Figure A.1.

It is apparent that for each triangle there is an associated vertex of the Voronoi diagram which is at the circumcentre of the three points which form the triangle. In other words, each Delaunay triangle has a unique vertex of the Voronoi diagram and no other vertex lies within the circle centred at this vertex, see

[1] Discussion presented in this Appendix depends heavily on materials taken from references [12, 56, 136, 139, 140, 147], in which the reader can find more details.

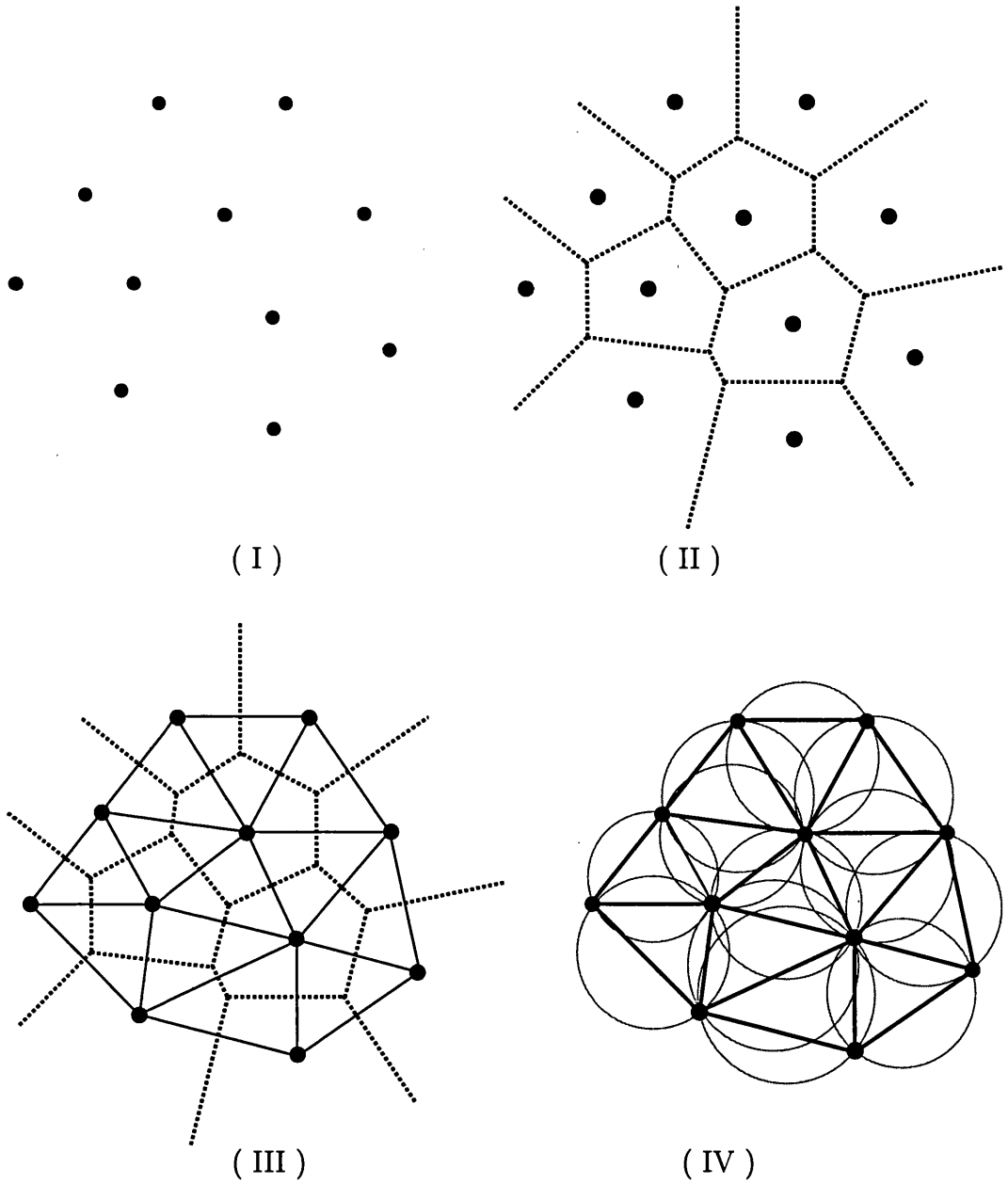


Figure A.1: Illustration of the Voronoi and Delaunay geometrical constructions. (I) set of points in 2D, (II) the Voronoi diagram, (III) Delaunay triangulation obtained by connecting every pair of points with a common segment, (IV) illustration of the in-circle criterion.

(IV) in Figure A.1. This is what is commonly known as the 'in-circle criterion'. In the three dimensional space, the territory of each data point is a convex polyhedron and the Voronoi diagram faces are equidistant between point pairs. If points with a common face are connected, then a set of tetrahedra is formed which covers the convex hull of points. Similarly in the two dimensions case, each vertex is at the circumcentre of a sphere defined by the four points that describe the tetrahedron, and an '*in-sphere* criterion' is applicable.

Construction of the Delaunay Triangulation

There are several algorithms used to construct the Delaunay triangulation. The algorithm developed by Bowyer, [12], is the most widely used in grid generation, it is favoured because of its flexibility to be applied in the two and three dimensions. Also, it is the approach adopted in the Delaunay grid generator used in this work [136, 139]. Thus, a brief description of its main steps is presented with some illustration figures that demonstrate the insertion of a new point into an existing Delaunay structure.

Triangulation Algorithm

In Bowyer's algorithm, each point is introduced into an existing Delaunay-satisfying structure, which is locally broken and then reconstructed to form a new Delaunay- satisfying construction. Hence, before any of the data points can be inserted a form of Delaunay-Voronoi structure must exist first, in fact, this is what **Step 1** in the following algorithm should provide.

Step 1
Define a set of points which form a convex hull within which all points will lie. It is appropriate to specify four (eight in 3D) points together with the associated Voronoi diagram structure.

Step 2
Introduce a new point anywhere within the convex hull, e.g. Q in (I) Figure A.2.

Step 3
Determine all vertices of the Voronoi diagram to be deleted. A point that lies within a circle centred at a vertex of the Voronoi diagram and passes through its three forming points results in the deletion of that vertex, e.g. vertices d_i in (II) Figure A.2. This follows from the in-circle criterion.

Step 4
Find the forming points of all the deleted Voronoi vertices, i.e. C_i in (II) Figure A.2. These are the contiguous points to the new point Q.

Step 5
Determine the neighbouring Voronoi vertices to the deleted vertices that have

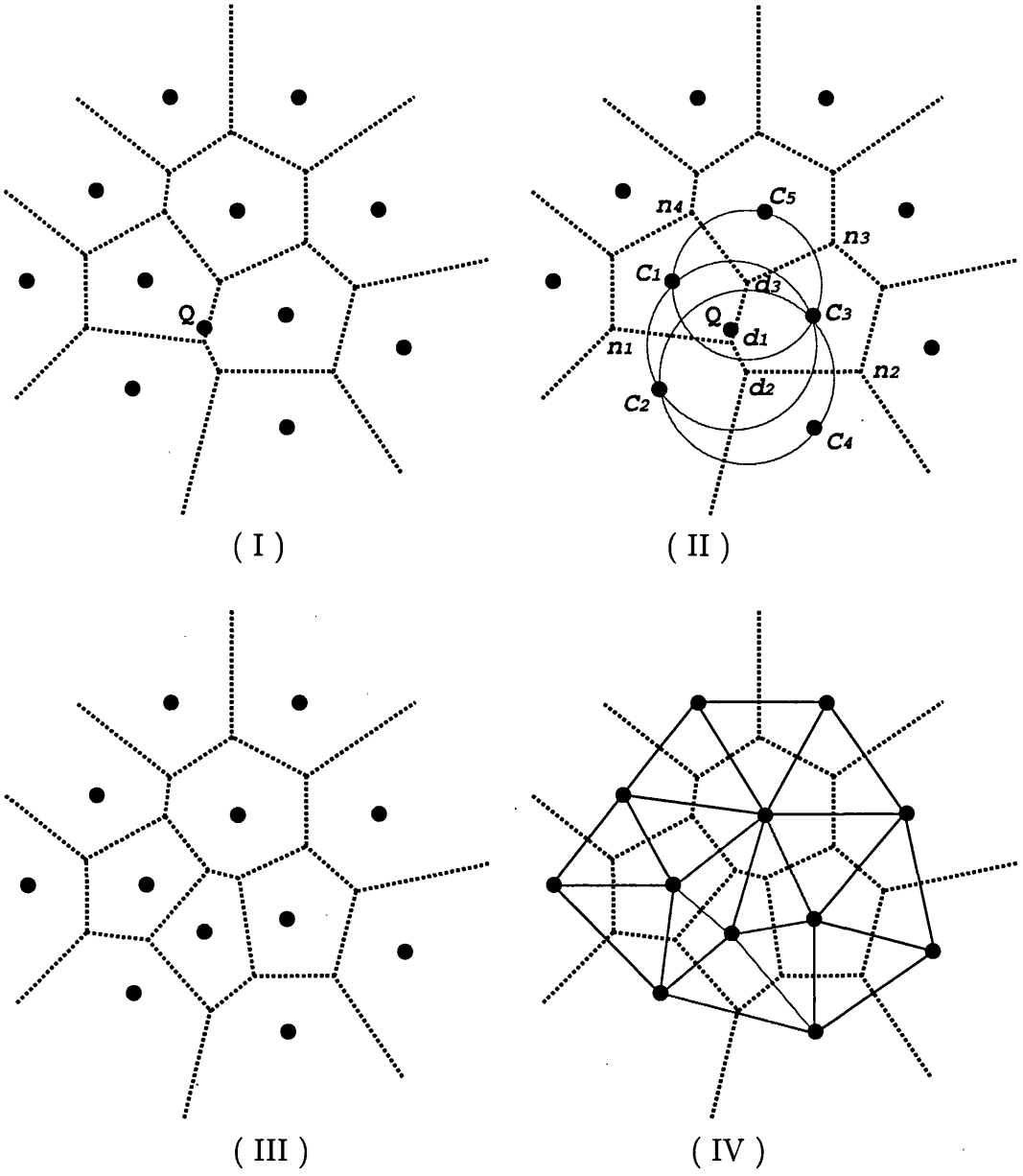


Figure A.2: Illustration of the Voronoi and Delaunay geometrical constructions while a new point (Q) is introduced to an existed diagram, (I). Vortices (di) are to be deleted and information related to their neighbouring vortices (n_i) and forming points (Ci), (II), are to be used in constructing the *new* Voronoi diagram, (III), and Delaunay triangulation (IV).

not themselves been deleted, i.e. n_i. These data provide the necessary information to enable valid combination of the contiguous points to be constructed.

Step 6
Determine the forming points of the new Voronoi vertices. The forming points of the new vertices must include the new point together with the two points that are contiguous to the new point and form an edge of a neighbour triangle, see (III) in Figure A.2.

Step 7
Determine the neighbouring Voronoi vertices to the new Voronoi vertices. Following Step 6, the forming points of all new vertices have been computed. For each new vertex, perform a search through the forming points of the neighbouring vertices, as found in Step 5, to identify the common pairs of forming points. When a common combination occurs, then the two associated vertices are neighbours of the Voronoi diagram.

Step 8
Reorder the Voronoi diagram data structure, overwriting the entries of the deleted vertices. See Figure A.3 and Table A.1.

Step 9
Repeat Steps 2-8 for the next point.

Figure A.4 shows a comparison between the triangulation before and after introducing a new point in (I), and an illustration of the in-circle criterion in the obtained triangulation in (II).

Data Structure

The structure of the Voronoi diagram and Delaunay triangulation can be described by constructing two lists for each Voronoi vertex. Each of the lists is of length three in two dimensions and four in three dimensions. The first list holds the points that define the forming points (i.e. Delaunay triangles in 2D and tetrahedra in 3D) of each Voronoi vertex, the second list holds the adjacent neighbouring vertices. Such tree data structure is essential for achieving an efficient implementation of the algorithm. Clearly, with the exception of **Step 3**, all the operations described are of a local nature and can be carried out in an operation count independent of the total number of points. A naive search for vertices of the Voronoi diagram to be deleted would amount to $O(N)$ operations per point and the to $O(N^2)$ operations for the total workload. An obvious improvement is to identify any one vertex that is to be deleted and then perform a tree search through the neighbouring Voronoi diagram data structure. However, Table A.1 demonstrates the data structure of Voronoi-Delaunay structure that is associated with the demonstrate example mentioned throughout the algorithm steps above. Lists presented in the Table can be examined against the diagrams in Figure A.3. Notice that vertices

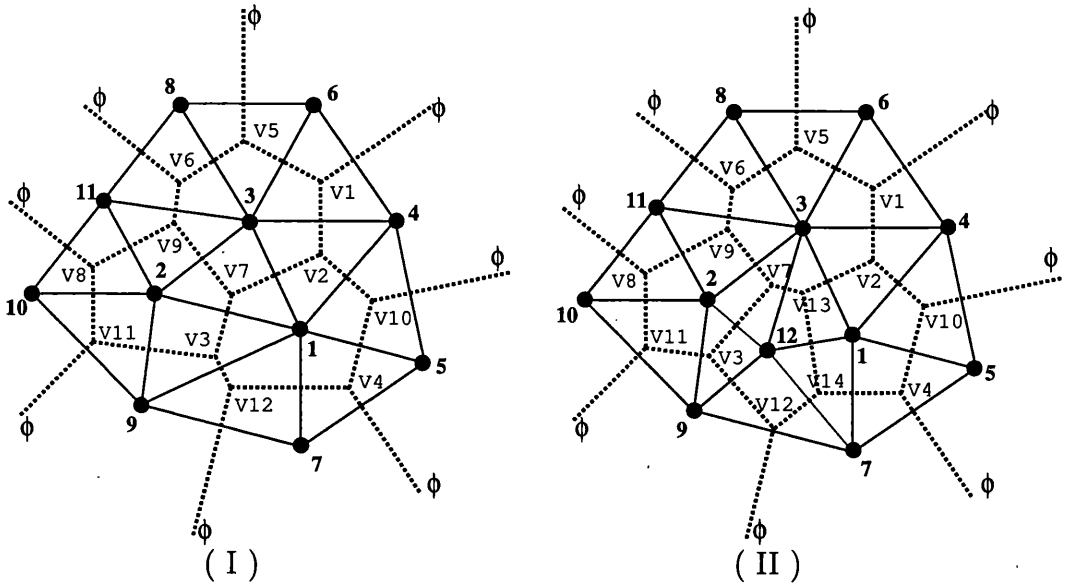


Figure A.3: Numbering of the Delaunay-Voronoi construction, before (I) and after (II) inserting a new point into an already existing structure. This figure is associated with the data structure presented in Table A.1.

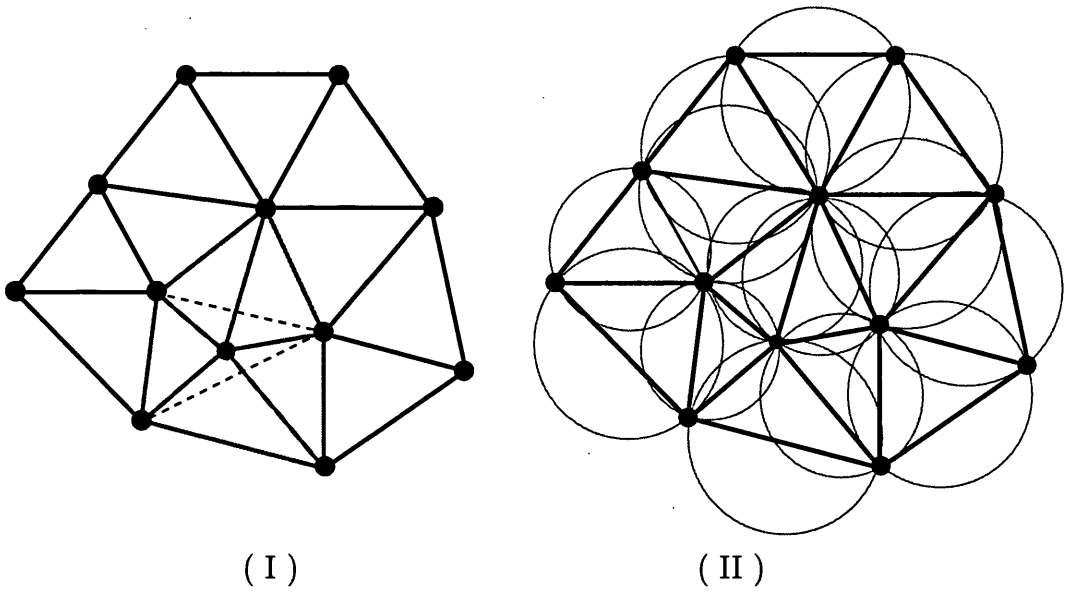


Figure A.4: (I) The *new* Delaunay triangulation (solid) obtained after introducing a point in comparison to the *old* triangulation, in (II) illustration of the in-circle criterion on the new triangulation.

which lie outside the convex hull, and therefore do not possess three forming points nor three neighbouring vertices, have been identified as *null* vertices.

	Before						After					
Vortex	Forming points			Neighbours			Forming points			Neighbours		
1	3	4	6	2	0	5	3	4	6	2	0	5
2	3	1	4	7	10	1	3	1	4	13	10	1
3	2	9	1	7	11	12	2	9	12	7	11	12
4	5	1	7	10	12	0	5	1	7	10	14	0
5	6	8	3	1	0	6	6	8	3	1	0	6
6	3	8	11	5	0	9	3	8	11	5	0	9
7	1	3	2	2	9	3	12	3	2	13	9	3
8	2	11	10	9	0	11	2	11	10	9	0	11
9	3	11	2	7	6	8	3	11	2	7	6	8
10	5	4	1	0	2	4	5	4	1	0	2	4
11	2	10	9	3	8	0	2	10	9	3	8	0
12	1	9	7	4	3	0	12	9	7	14	3	0
13	-	-	-	-	-	-	1	3	12	2	7	14
14	-	-	-	-	-	-	1	12	7	4	13	12

Table A.1: Data structure for the Voronoi diagram and Delaunay triangulation shown in Figure A.3, before and after introducing a new point to an existing structure. Notice the modifications in both lists, forming points and neighbouring vertices, e.g. vertices 2-4.

A.2 Automatic Point Creation

A grid must have a sufficient number of points so all important geometrical features of the domain are well presented, also the approximation in the numerical solution is satisfactory. The point spacing should vary smoothly such as the gradient of the solution can be captured too. On the other hand, computational efficiency must be taken into consideration so the grid must not be over dense.

Obviously, the triangulation algorithm presented above gives no indication as to how grid points can be generated. Hence, an external procedure that derives the points location must be introduced if a Delaunay based grid generator is to be built. Two ways which have been used include grid superposition methods and points generated from independent technique, e.g. structured grid methods. The basic idea in the former approach is to superimpose a regular

grid over the domain, whilst in the later is to generate a set of points independently and then connect them to form the grid. Although the superposition methods can be very fast and produce good quality grids in the interior of the domain, the quality near the boundary can be problematically poor. The second approach is very restrictive, in particular, for general complex geometries. However, in addition to the clear limitations above, the requirement for controlling the density of points within the domain in a more flexible manner, has led into the development of more advanced procedures.

Automatic point creation with 'controlled' spacing

A summary of a procedure that has proved to be flexible and efficient in creating points in association with the Delaunay triangulation technique is presented, for more details the reader can consult [56, 140, 147]. The proposed procedure allows for different methods (e.g. boundary point distribution, background grid and sources) to be used in controlling the points density and distribution in the grid. Although, that the presented algorithm considers the method adopted in this research only, the layout in general is applicable on all other methods and for both of triangular and hexahedral grids. However, it is appropriate to mention that the procedure overall requires minimal manual user input and provides good grid quality.

Layout of point creation procedure

Step 1

Compute the point distribution function dp_i for each boundary point $\mathbf{r}_i$ as:

$$dp_i = \frac{1}{M} \sum_{j=1}^{M} \|\mathbf{r}_j - \mathbf{r}_i\|$$

where $j = 1, M$ are the points surrounding point i; $M = 2$ in the triangular grids (2D) and $M > 2$ in the hexahedral grids (3D).

Step 2

Generate an *initial* grid that connects the boundary points by a set of triangles (tetrahedra) using the Delaunay triangulation procedure.

Step 3

Initialise the number of interior field points created, $N = 0$.

Step 4

For each triangle (tetrahedron) within the domain perform the following:

a) Define a prospective point, c, to be at the centroid as:

$$\mathbf{r}_c = \frac{1}{K} \sum_{k=1}^{K} \mathbf{r}_{nk}$$

where K is number of nodes per element, i.e. 3 in 2D and 4 in 3D.

b) Derive the point distribution function dp_c by interpolating the distribution function from the associated triangle (tetrahedron) nodes:

$$dp_c = \frac{1}{K} \sum_{k=1}^{K} dp_{nk}$$

c) Update the dp_c based on the effect of the point spacing sources available [2].

d) Compute the distances l_k from the prospective point c to each node n_k, where $k = 1, 2, 3$ in 2D and $k = 1, 2, 3, 4$ in 3D:

if $l_k < \alpha\, dp_c$ (for any k) **then**

reject the point and return to the beginning of Step 4

where α is a parameter that effects the grid point density globally.

else

accept the point for insertion by the Delaunay triangulation.

add the point into the new points list, $N = N + 1$; $\mathbf{r}_N = \mathbf{r}_c$

assign the value of dp_c found in step c above to the new point.

end if

Step 5
If $N = 0$ go to Step 7.

Step 6
Perform the Delaunay triangulation of the N derived points, go to Step 3.

Step 7
Apply a smoothing technique on the entire grid points.

[2]Further discussion of this step is presented immediately after the algorithm.

Point Spacing Sources

The concept of the use of 'sources' provides a mechanism by which an adequate point spacing in particular regions of the domain can be defined and controlled explicitly. An appropriate description of a source must include a clear definition of the region that is effected as well as the 'function' adopted to control the point distribution. Figure A.5 shows three different types of sources used in this research, presented in a simple 2D form. The 'point source' is the basic type in which the fundamental features for a source i are defined by:

- The location within the domain, $\mathbf{X}_i$
- Two circles centred at $\mathbf{X}_i$ and have the radii r_{i1} and r_{i2}, where $r_{i2} \geq r_{i1}$.
- A parameter δ_i which can be used to define the point distribution function at a point c as:

$$dp_c = \delta_i \qquad \text{if} \quad \|x_{ci}\| \leq r_{i1}$$

Or,

$$dp_c = \delta_i e^{\|x_{ci}\| \ln\left(\frac{2}{r_{i2}-r_{i1}}\right)} \qquad \text{if} \quad \|x_{ci}\| > r_{i1}$$

where $\|x_{ci}\|$ denotes the distance between point c and the centre of the point source i.

As **Step 4**–*c* in the algorithm suggests that the value of point spacing at point c is checked against available sources $i = 1, ...M$. The *minimum* value obtained from all sources, based on the point distribution function defined above, is considered. However, in order to ensure an adequate and consistent point distribution in the *entire* domain (i.e. including regions that are not influenced by sources), it is appropriate to check the value obtained from sources with the value obtained from interpolation in **Step 4**–*b*. Thus, eventually, the smallest value between b and c in **Step 4** is assigned to the prospective point c.

The extension to line and triangle sources is straightforward, see Figure A.5 where the regions considered as within r_1 are bounded by the dashed lines, also see Figure A.6 for an illustration of such sources effect. In practice, the features of each end point source can be defined independently. Figure A.7 shows a typical example of two line sources, where the effected area differs between the two end point sources in one of the line sources (on the right) and the point spacing parameter in the other line sources (on the left).

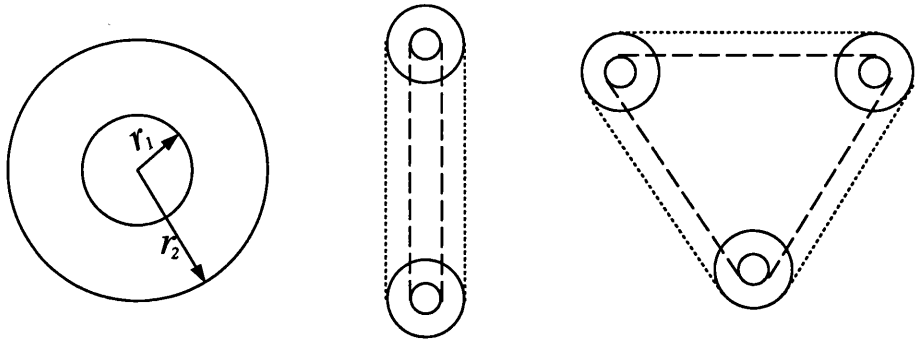


Figure A.5: Point (I), line (II) and triangle (III) sources. The inner circle, r_1, defines the region over which the point spacing is equal to a specified value. Whilst $r_2 - r_1$ defines the region over which the specified value decays to the value imposed globally. The inner circle effect region in the line and triangle sources is bounded by dashed lines.

All the concepts of point spacing sources presented above in the two dimentional form can naturally be extended into the three dimensions by simply considering the two circles r_1 and r_2 as two spheres. However, Figure A.8 shows a typical example of point, line and triangle sources in the 3D space distributed over an aircraft configuration in preparation to generate a tetrahedral grid for a CFD simulation.

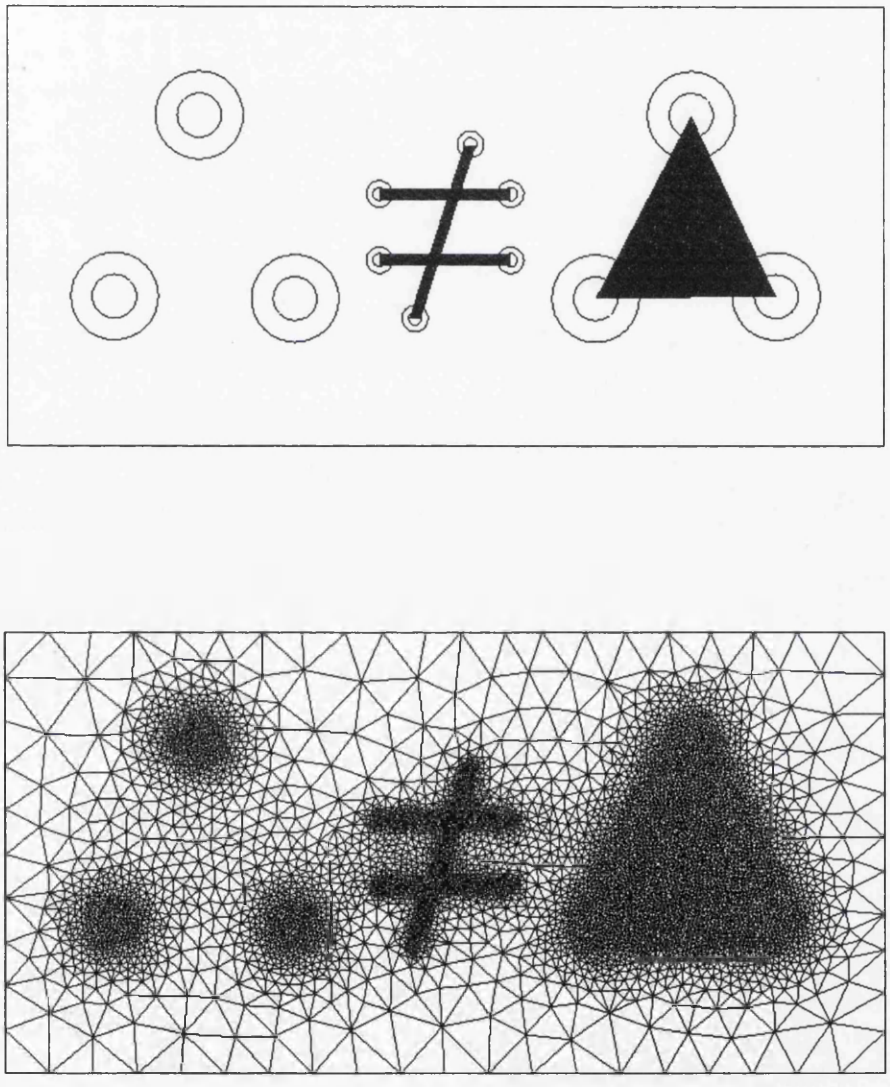
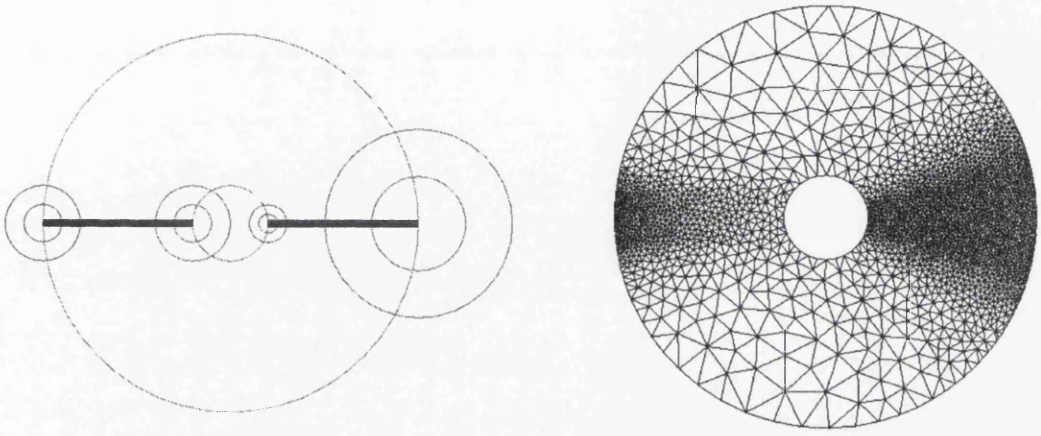


Figure A.6: Illustration of the effect of different type of grid point spacing sources.



(I) (II)

Figure A.7: Variations in the parameters of the line (triangle) sources can be used. (I) Two line sources with different areas of effect (right) and point spacing parameter (left). (II) The final grid.

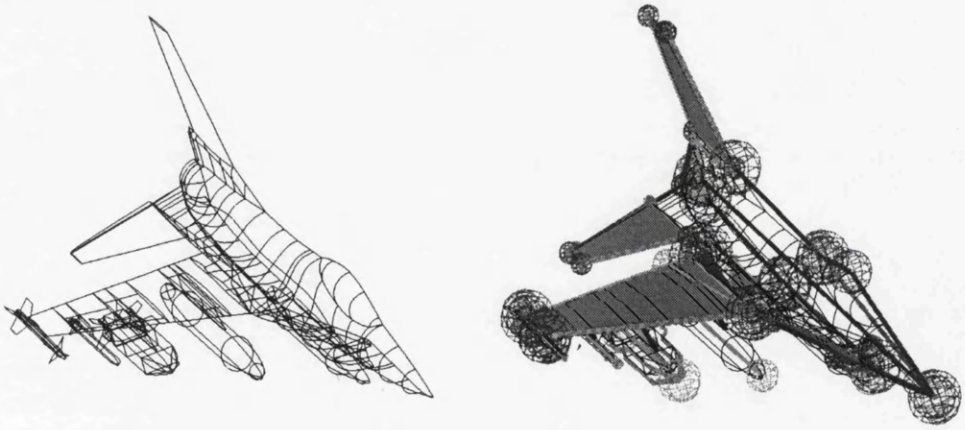


Figure A.8: An example of point spacing sources in the three dimensions distributed around a militarily aircraft configuration.

Appendix B

Message Passing Library

Many message passing libraries have been developed in the last decade, only some of them have become widely available and proved to be highly portable [68]. Parallel Virtual Machine (PVM) was a very popular library. An early version of PVM was written at the University of Tennessee in 1991 based on the concepts had been constructed in the Oak Ridge National Laboratory [120]. Later in 1993 Version 3.3 was released and immediately embraced by researchers in the parallel process community, and since then PVM has been undergoing further developments. In 1997 Version 3.4.3 embarked on new features like the interoperablity between Windows NT and Unix systems [47], and currently the research team in ORNL is involved in the HARNESS (Heterogeneous Adaptable Reconfigurable NEtworked SystemS) project[1] [7].

Since the early days of PVM the MPL model for parallel programming has emerged assertively, but the wide variation in the syntax and implementation among different libraries was growing to a problematic level. In Spring 1992, a general agreement among experts in the high performance computing community had been reached, "an attempt at standardisation might usefully be undertaken" [126]. After a series of meetings in 1993 for a broadly based committee of vendors, implementors, and users (the Message Passing Interface Forum) a draft document was produced. In May 1994 Version 1.0 of MPI (Message Passing Interface) standard was completed [51]. MPI is a library specification for message-passing, it was designed for high performance on both massively parallel machines and on workstation clusters.

Programmers always wonder whether to write their parallel applications using PVM or MPI. In fact, comparison between the two[2] has been the subject for several papers. It is beyond the intention of this thesis to get engaged on

[1]This project seeks to remove some of PVM limitations in order to create a Distributed Virtual Machine (DVM)!.

[2]Considering that PVM is a specification as presented in [45]

this debate, however, it is appropriate to mention that MPI has been chosen based on the performance and the portability criteria. Nevertheless, switching between MPI and PVM should be relatively straightforward for all the parallelisation work done in this research. Table B.1 compares a few of the most common message passing functions between two typical libraries of MPI and PVM. Readers interested in a *thorough* comparison between the two systems are advised to consult [46, 55], where the differences between features are described and under what circumstances one is favoured over the other.

PVM3.3	MPICH1.0.13	Main job of the subroutine
pvm_mytid	MPI_Init	Start a message passing session
pvm_exit	MPI_Finalize	Stop the message passing session
pvm_parent	MPI_Comm_rank	Determine the process identifier
pvm_psend	MPI_Send	Send a message
pvm_precv	MPI_Recv	Receive a message
pvm_pack	MPI_Pack	Incrementally add data to a buffer
pvm_unpack	MPI_Unpack	Receive data from a buffer

Table B.1: Basic message passing subroutines in PVM and MPI, C language binding.

Actually, few attempts were made to merge the two systems or some of their features under one combined new system. UNIFY, a message passing library developed in the Mississippi State University, provides a subset of MPI within the PVM environment, without sacrificing the PVM calls already available [127]. PVMPI, developed in University of Tennessee and ORNL, was an early attempt to integrate both Systems (PVM3.4 and MPI-1). The main goal in this integration was to interface the flexible process management and the Virtual Machine control from PVM, with the advanced point-to-point and collective communication options from MPI [29]. Nevertheless, recently some convergence between the functionality offered by the two systems has occurred. MPI-2[3], the latest release of MPI became available in 1998, offers a wide range of additional functions to the implementors. Features were in PVM and not in MPI-1, (like dynamic process creation and management, external interface and parallel I/O), now are available, also bindings for Fortran 90 and C++ are standardised.

Multiple implementations of MPI have been developed, some of them are highly specialised, known and used by a small number of people in the parallel

[3]Due to the nature of constant changes in Web resources, the intention throughout this thesis was not to use it for citing at all. Unfortunately, the author couldn't find any other source to cite the official document of MPI-2 standard specification, "*www.mpi-forum.org/index.html*".

processing field:
- MPI-FM (MPI-Fast Messages)[79].
- MPI-CCL (MPI-Collective Communication Library)[13].

Others are more popular and portable:
- CHIMP (Common High-level Interface to Message Passing)[1].
- LAM (Local Area Multicomputer) [14].

MPICH is the most popular public-domain implementation of MPI, developed jointly by Argonne National Laboratory and Mississippi State University [53]. "The 'CH' in MPICH stands for 'Chameleon', symbol of adaptability to one's environment and thus portability. Chameleons are fast, and from the beginning a secondary goal was to give up as little efficiency as possible for the portability" [54]. (MPICH Version 1.0.13/1996)[4] is the library used in all programs developed during the course of this research, any further refer to MPI in this Chapter conveys to it.

Although, MPI in its entirety is a complex system, it comprises more than 125 functions. Some of these functions may have numerous parameters or variants. Fortunately, a small number of them, less than 20, can provide an adequate data base for writing parallel programs for a wide range of applications. A number of these functions and their use in the developed programs will be illustrated shortly with example, but first a useful brief overview of the main features in MPI is to be addressed.

B.1 Main Features and Functions of MPI

MPI provides *point-to-point* communication operations, in which two already named processors can send and receive messages between themselves only. Also, it provides *collective* communication operations, in which one distinguished processor can broadcast messages to all other processors *one-to-all*, or gather messages from them *all-to-one*. Various arithmetic operations can be applied simultaneously while the data is collected on one processor using the later type of communication. However, in the programs developed for this study, the first type *point-to-point* is used in most of the communication operations (whenever the Manager communicates with individual Workers or vice versa), whilst the *one-to-all* type is used in a few simple operations only.

MPI has a mechanism called *Communicator* which allows the programmer to define subsets of processors to be in groups. MPI_COMM_WORLD is a predefined communicator, which includes *all* processors that evolve at the beginning of the execution time. It is the only type of Communicator employed in this research. *Topology* is another mechanism which allows associating different

[4]MPICH Version 1.2.0/December, 1999 implements most of the MPI-2 specifications

numbering schemes with the processors in Communicator(s). There are essentially two types of topologies, a Cartesian (grid) topology and a graph topology, they both have no relation with the actual physical structure of the processors. An adequate use of the Topology and the Communicator mechanisms can cause substantial improvements on the performance of some parallel programs, in particular when the communication among certain processors takes a form of repetitive pattern. However, none of the two mechanisms was exploited in this research. In fact, because of the relatively small number of communication operations, introducing such mechanisms into the developed parallel framework would have compounded its structure; in addition, achieving a noticeable improvement on the performance overall is very unlikely.

MPI has two types for sending and receiving data between two processors: *blocking* and *nonblocking*. In the former, the program won't continue running until the buffering operation (send (MPI_Send) or receive (MPI_Recv)) is complete, i.e. the memory equipped by the buffer is free and available to be utilised again. In the *nonblocking*, the program may continue as soon as the call for sending (MPI_Isend) or receiving (MPI_Irecv) is established. In other words, the calls to send or receive may return before the buffering operation is complete. Using *nonblocking* type for communicating data between processors improves the performance overall, but it may demand more memory since more than one copy of the buffer can be filled simultaneously. However, having the priority in the developed algorithms given to the optimal use of the available computing memory the *nonblocking* type was totally ignored.

Also, MPI has four different modes for communications, *standard, buffered, synchronous* and *ready*. In *standard* mode the operating system controls the buffering, while in *buffered* mode it is done explicitly by the user. In *synchronous* mode a send will not complete until a matching receive has occurred. In the *ready* mode the programmer can notify, and hand in the operation to, the underlying system as soon as the receive has been posted. Once again, discussing such specialised technical issues is beyond the main interest of this thesis, however, it may now make more sense if we report that *standard blocking* is the communication mode that adopted throughout the developed programs. More information about the communication modes can be found in [51, 131].

Basic functions of MPI in every parallel program

The number and type of MPI functions employed in parallel programs may vary dramatically, but in general there must be *minimum functions* in each. MPI_Init and MPI_Finalize are the start and end points for every parallel session respectively. No MPI function can be called before MPI_Init or after MPI_Finalize, and they both are called once and only once. The former, sets up the parallel environment such that other MPI functionalities can be used, while MPI_Finalize secures an appropriate exit by cleaning all 'unfinished business'.

It is interesting to mention, that using any other MPI function apart from these two functions is 'optional' but, obviously, using more of data communication functions is inevitable. However, an ordinary message passing parallel program would have at least one call to each one of the *basic* functions that are presented within the following general layout.

Layout of a typical 'MPI program'

```
#include "mpi.h"
          ...

int main(int argc, char **argv){
          ...
          ...
  /* No MPI function must be called before here */

    MPI_Init(&argc, &argv);
    MPI_Barrier(MPI_Comm MPI_COMM_WORLD);
    MPI_Comm_rank(MPI_Comm MPI_COMM_WORLD, int &my_id);
          ...
    MPI_Comm_size(MPI_Comm MPI_COMM_WORLD, int &num_procs);
          ...
          ...
      /* Start of the dominating parallel processing work  */

         Other MPI functions can be utilised in here such as:
           ... ... ...
             MPI_Pack(......);  MPI_Send(......);
             MPI_Recv(......);  MPI_Unpack(......);
           ... ... ...

      /* End of the dominating parallel processing work  */
          ...
          ...
    MPI_Finalize();

  /* No MPI function must be called after here */

}
```

Including the header file "mpi.h", which contains definitions of macros and function prototypes necessary for compiling, is a compulsory point. Argu-

ments for the MPI_Init function must be very familiar to a C programmer, and they can be ignored by any other reader. The MPI_Finalize is an integer type function with a *void* argument list, it passes back to the system an integer reporting on the exit mode (e.g. 0 the program has terminated normally). Recalling the definition of the 'MPI_COMM_WORLD' in here would make the understanding of functions MPI_Comm_rank, MPI_Comm_size and MPI_Barrier straightforward. The first function returns, in its second argument (my_id), a positive integer that is considered as the rank of the calling processor within the associated communicator MPI_COMM_WORLD. In the developed parallel framework, the MPI_Comm_rank is called at the very beginning of the run time and processor with rank (0) is assigned to be the Manager. Subsequently, the Manager calls the MPI_Comm_size function to find out about the total number of processors involved (i.e. num_procs), such that the environment for the parallel framework is setup. The MPI_Barrier function blocks the flow of the program until all the communicator processors reach the same point (synchronisation). Other functions mentioned in the layout above are classified as data communication functions, which are discussed in detail next section.

B.2 Data Communication in MPI

In order for a message to be successfully communicated, in addition to the *data* itself extra information must be available which form the *envelope.* MPI specifies the following items:

1. The rank of the Sender
2. The rank of the Receiver
3. A communicator
4. A tag

To examine these items we detail the syntax of two functions, MPI_Send and MPI_Recv. C language binding is used, however, the Fortran version is very similar with one integer as an additional item in the argument list. This 'extra' integer replaces the *return value* facility that is available naturally in the C language functions.

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buffer, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status )
```

The **buffer** points to a contiguous block of memory where the contents of the message are stored. The **count** identifies the maximum number of elements, which have the MPI *datatype* type, in the buffer. MPI has its own types of data which match the types known in standard programming languages (e.g. MPI_INT for integers) and exceeds by two extra. One of the extra data types is known as (MPI_PACKED) which is used in conjunction with 'packed data', this will be discussed shortly. The argument **dest** represents the destination (where the message is going to), and **source** represents the origin (where the message is coming from). The *rank* of the sending and receiving processors must be declared in both cases, but the **source** can be a predefined constant sometimes. The MPI_ANY_SOURCE is a common and very useful predefined constant which can be used when a processor is ready to receive from *any* sending processor. This feature has been exploited frequently by the Manager in the developed parallel framework, since the Manager receives data that can be sent *any* Worker.

Most likely the reader will be wondering, if there is a similar 'wild-card' form for the **dest** or not?. No, MPI does not allow this at all, instead it provides another function (MPI_Bcast) which enables one processor to *broadcast* a message to all processors in a communicator. In other words, no message can be sent out without knowing where it is supposed to be delivered. Furthermore, MPI insists on labelling every message with a distinguished positive integer, i.e. the **tag**. Although this may look a very strict policy, it should lead to unmistakable communication operations consider how it looks from the receiver view point.

First of all, it is important to report that MPI provides a predefined tag (MPI_ANY_TAG) which can be applied by the receiver only. Then there is a possibility to have a message received with unknown sender (MPI_ANY_SOURCE) and unknown label (MPI_ANY_TAG)!. Certainly, with such situation the 'strict policy' will become very loose. In fact, the answer to this is hidden within the last argument to be discussed here, which is the **status**. It is a data structure with three fields, two of them always holds information about the **source** and the **tag**. Thus, any message received with (MPI_ANY_SOURCE) the **status** can provide the rank of the processor sent it, and similarly with the (MPI_ANY_TAG) case the message tag is obtainable. Both cases of this feature are exploited extensively by the Manager, an example can be found in Section 4.2.1 where the Dynamic Parallel Processing is discussed.

Packing and unpacking scattered data.
As already stated, the communicated data must be always stored in contiguous memory locations. In other words, whenever a disjoint data arises then a separate communication operation has to be initiated. On the other hand, there is an overhead cost associated with every call to MPI_Send or MPI_Recv, regardless the length of the message. Hence, assembling the individual data items in a 'buffer' and subsequently initiating one communication operation instead of several should improve the program overall performance.

MPI offers various mechanisms to group a set of data in one message. The mechanism used in this research uses the functions MPI_Pack and MPI_Unpack in an explicit form. In order to put a set of sparse data into one block of contiguous memory, the program declares a memory address as a start of a buffer and then calls the MPI_Pack function a number of times in sequence. The MPI_Pack in every call *incrementally* adds the desired data to the associated buffer. In a similar manner, the MPI_Unpack can be employed to extract the data stored in the buffer. However, it might be appropriate to review the syntax of these two functions in a similar way as for the MPI_Send and MPI_Recv functions earlier. Items in the argument lists of the MPI_Pack and MPI_Unpack functions that have been discussed with previous functions will be ignored.

```
int MPI_Pack(void *pack_data,  int count,  MPI_Datatype datatype,
             void *buffer,  int buf_size,  int *position,
             MPI_Comm comm )


int MPI_Unpack(void *buffer,   int buf_size,   int *position,
               void *unpack_data, int count, MPI_Datatype datatype,
               MPI_Comm comm )
```

The **pack_data** and **unpack_data** are pointers to the data that is packed or unpacked in the functions MPI_Pack and MPI_Unpack respectively. The **buf_size** is an integer which represents the size (in bytes) of memory reserved for the **buffer** [5]. The **position** pointer is utilised by both functions to keep track of the *current* position within the buffer while the packing and unpacking operation is going on. It is both input and output argument at the same time.

[5]MPI provides a function (i.e. MPI_Pack_size) that can be employed to find the exact value of memory needed to store various data. Since this value may vary from one platform to another, such function becomes very essential to allocate the memory required by the buffer on the fly.

Before calling one of the functions, **position** points to the current location in the buffer, whilst on the return of the call it points to a new location that comes after (or before) a **count** number of elements in the buffer.

Neither of the two functions above performs any checking on memory overlapping nor data type mismatching. It is the programmer responsibility to make sure that a sufficient memory is allocated for the buffer usage, also to match the packing and unpacking operations. The 'matching' here does not mean the unpacking operation has to follow the same order used during the packing operation. The programmer has unlimited choice in extracting the data from the buffer. In fact, this property has been utilised extensively in the developed programs. In particular, when the Manager receives the internal boundary grids back from Workers. Some of the data in every internal boundary grid belongs to the original boundary grid, which is already available on the Manager and there is no need to *unpack* it. MPI allows mismatching data types to be assembled in one buffer, and consequently uses its own unique data type (MPI_PACKED) to define such combination. Thus, the 'packed data' can be sent (received) in one call of MPI_Send (MPI_Recv) with the **datatype** argument defined as MPI_PACKED.

The intention behind reviewing the syntax of the MPI_Send, MPI_Recv, MPI_Pack and MPI_Unpack is mainly to enrich the understanding of the parallelisation work presented in chapter 4 however, a comprehensive review can be found in [51, 53]. A reader who has no knowledge of MPI, and interested in a wider introduction to its use in computational mechanics particularly, can consult [69]. Where some of the functions discussed previously are reviewed and introduced within simple examples of numerical algorithms.

B.3 Performance Analysis Tools

Unlike the traditional sequential programs, the analysis of parallel programs performance is a complex task, and some special tools may have to be employed. UPSHOT is a very useful tool for understanding parallel programs behaviour, it offers a graphical display of parallel time-lines. Each line is associated with a processor, and coloured bars reflect on the state of the processor at any time can be utilised [72]. The exact time associated with each bar is accessible through the graphical interface, a *zooming* functionality is also available. UPSHOT works on 'logging files' generated by the MPE (Multi-Processing Environment) library [52].

Both MPE and UPSHOT are distributed with the MPICH library, but they are *not* supported by *all* MPI implementations. However, the integration of these tools within the developed parallel framework does not have any effect on the overall portability. The MPE functions inside the main program do not

interfere with the core work of the parallelisation, furthermore, they can be omitted in an automated way during the compilation procedure. In short, by exploiting the MPE and UPSHOT a detailed and comprehensive performance analysis of an MPI programs can be achieved.

The logging file can be seen as time-stamped event trace file. In such a file, information about the start and end time (i.e. events) of certain parts of the program (i.e. states) are collected in sequence. The MPE library provides a set of functions which can be, explicitly, utilised by the programmer to define the states and record (log) the events. However, implementing MPE library functions inside an MPI program can be summarised briefly as follow:

Every processor must start by calling the (MPE_Init_log) function, and finish by calling the (MPE_Finish_log) function which will force the data collected on it to merge with data from other processors into one logfile. No call to any MPE function can be made before (MPE_Init_log) or after (MPE_Finish_log). The logging can start, or resume, after calling (MPE_Start_log), and it can be stopped, or suspend, by calling (MPE_Stop_log). Between these two functions events can be recorded by calling the (MPE_Log_event) function, which marks either the start or the end of a state. The state should be already defined by the (MPE_Describe_state) function, which specifies the starting and ending event type; assigns a unique colour and name for the state.

A general layout of employing MPE functions in an MPI program can be demonstrated as:

```
#include "mpi.h"
#include "mpe.h"

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

   MPI_Init(&argc, &argv);
   MPI_Barrier(MPI_Comm MPI_COMM_WORLD);
   MPI_Comm_rank(MPI_Comm MPI_COMM_WORLD, int &my_id);

         ...

     /* No MPE function must be called before here */

   MPE_Init_log();

      /* Describe all states, including events (i.e.
         start and end), name and a unique colour */
```

```
   MPE_Describe_state(int start1, int end1, char *name1, char *color1);
   MPE_Describe_state(int start2, int end2, char *name2, char *color2);
   MPE_Describe_state(......);
         ...
         ...

   MPE_Start_log();        /* Start the logging session, to be called
                           by one processor only (e.g. the Manager)*/

           ... ... ...

         /* Then inside the main body of the program
            the MPE_Log_event function can be called
            whenever an event needs to be recorded. */

           ... ... ...

     MPE_Log_event(int event, int intdata, char *chardata);

           ... ... ...        /* The event could be the start
           ... ... ...           (e.g. start1) or the end (e.g.
                                 end1) of a state (e.g. name1) */
     MPE_Log_event(......);

           ... ... ...
                   /* Events can be reported by any processor. */
           ... ... ...

   MPE_Stop_log();        /* End the logging session, to be called
                          by one processor only (e.g. the Manager)*/
           ... ... ...
           ... ... ...

   MPE_Finish_log(*logfilename);
                       /* Merge the log data from all processors
                       and write out the logfile, by the Manager */
   MPI_Finalize();

}
```

Clearly, states (and events) should be identified (and reported) in a sensible

manner, such that information collected at the end of every run are just sufficient enough to demonstrate the program performance. Overloading a parallel program by reporting large number of events can eventually effect the overall performance, as well as make the UPSHOT's output difficult to read. For example, reporting *only* on the major functions (subroutines) in the developed framework would have required more than 40 states to be defined and hundreds of events to be logged. Whilst by identifying only a few states which are associated with well recognised procedures in the general algorithm can still be informative, see Table 6.11, Figures: 4.5 and 6.21.

MPE provides another set of functions that can be used to enhance program output with some simple graphics. These functions are not relevant to the logfile activities nor to the UPSHOT tools, hence, they are totally ignored in here. However, the syntax of these functions and simple examples are discussed briefly in [52].

Bibliography

[1] **R. Alasdair A. Bruce, James G. Mills and A. Gordon Smith** *CHIMP User Guide*, Technical Report EPCC-KTP-CHIMP-V2-USER, Edinburgh Parallel Computing Centre, University of Edinburgh, 1994.

[2] **S. Arcilla, J. Häuser, P.R. Eiseman and J.F Thompson,** *Proceeding of the 3rd International Conference on Grid Generation, in Computational Fluid Dynamics and Related Fields*, Barcelona, Spain, Pub. North–Holland, 1991.

[3] **T.J Baker,** Automatic Mesh Generation for Complex 3-Dimensional Regions Using a Constrained Delaunay Triangulation, *Engineering With Computers*, Vol. 5, 161–175, 1989.

[4] **T.J Baker,** Shape Reconstruction and Volume Meshing for Complex Solids, *International Journal for Numerical Methods in Engineering*, Vol. 32, No.4, 665–667, 1991.

[5] **T.J Baker,** Delaunay–Voronoi Methods, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 16-1–16-11.

[6] **S.T. Barnard and H.D. Simon,** A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems, *Concurrency: Practice & Experience*, Vol. 6, No.2, 101–117, 1994.

[7] **M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam,** HARNESS: A Next Generation Distributed Virtual Machine, *International Journal on Future Generation Computer Systems*, Elsevier Publ., Vol. 15, No. 5-6, 1999.

[8] **P. Bezier,** *The mathematical basis of the UNISURF CAD system,* Butterworths, London, 1986.

[9] **J. Bonet and J. Peraire,** An Alternating Digital Tree (ADT) algorithm for 3D geometric searching and intersection problems, *International Journal for Numerical Methods in Engineering*, Vol. 31, 1–17, 1991.

[10] **J. Boris,** A vectorised algorithm for determining the nearest neighbours, *Journal Of Computational Physics*, Vol. 66, 1–20, 1986.

[11] **H. Borouchaki and P.L. George,** Aspects of 2-d Delaunay mesh generation, *International Journal for Numerical Methods in Engineering*, Vol 40, No.11, 1957–1975, 1997.

[12] **A. Bowyer,** Computing Dirichlet tessellations, *Computers Journal*, Vol. 24, No.2, 162–166, 1981.

[13] **J. Bruck, D. Dolev, C.T. Ho, M.C. Rosu and R. Strong,** Efficient message passing interface (MPI) for parallel computing on clusters of workstations *Journal of Parallel and Distributed Computing*, Vol. 40, No.1, 19-34 1997.

[14] **G. Burns, R. Daoud and J. Vaigl,** LAM: An Open Cluster Environment for MPI, In:*Proc. Supercomputing Symposium 94*, editor J. Ross, University of Toronto, 379-386, 1994.

[15] **C.C. Chang, G. Czajkowski, and T.V. Eicken,** Design and Performance of Active Messages on the IBM SP2, *Cornell CS Technical Report 96-1572,* February 1996.

[16] **Min-Bin Chen, Tyng-Ruey Chuang and Jan-Jan Wu,** Experience in Parallelizing Mesh Generation Code with High Performance Fortran, In:*Proc. 9th SIAM Conf. on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 1999.

[17] **L.P. Chew, N. Chrisochoides and F. Sukup,** Parallel Constrained Delaunay Meshing, *Trends in unstructured Mesh Generation, ASME*, AMD-Vol 220, 89-96, July 1997.

[18] **N. Chrisochoides and F. Sukup,** Task Parallel Implementation of the Bowyer-Watosn Algorithm, In:*Proc. 5th Int. Conf. on Numerical Grid Generation in Computational Field Simulation*, Pub. NSF Engineering Research Centre for Computational Field Simulation, 773-782, 1996.

[19] **S. Coons,** *Surface patches and B-Spline curves,* In R. Barnhill and R. Riesenfeld editors, Computer Aided Geometric Design, 1–16, Academic Press, 1974.

[20] **H.L. de Cougny, M.S. Shephard and C. Oztrura,** Parallel Three–Dimensional Mesh Generation, *Computing Systems in Engineering*, Vol. 5 No.4–6, 311–323,1994.

[21] **H.L. de Cougny and M.S. Shephard,** Parallel volume meshing using face removals and hierarchical repartitioning, *Computer Methods in Applied Mechanics and Engineering*, Vol. 174, 275–298, 1999.

[22] **H.L. de Cougny and M.S. Shephard,** Parallel unstructured grid generation, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 24-1–24-18.

[23] **M. Cross, B.K. Soni, J.F. Thompson, J. Häuser and P.R. Eiseman,** *Proceeding of the 6th International Conference on Numerical Grid Generation in Computational Field Simulation*, London, UK, Pub. International Society of Grid Generation(ISGG), 1998, ISBN 0-9651627-2-9.

[24] **B. Delaunay,** Sur la Sphère Vide, *Bulletin of Academic Science,* URSS VII, Class. Science National, 793–800, 1934.

[25] **Y.M. Ding and J.P. Densham,** A Dynamic and Recursive Parallel Algorithm for Constructing Delaunay Triangulations, *Advances in GIS Research*, Vol. 1 & 2, Ch.71, 682-696, 1994.

[26] **G.L. Dirichlet,** Uber die Reduction der Positiven Quadratischen Formen mit deri Unbestimmten Ganzen Zahlen, *Z. Reine Angew. Mathematics*, Vol. 40, No.3, 209–227, 1850.

[27] **M. Eldredge, T.J.R. Hughes, R.M. Ferencs, S.M. Rifai, A. Raefsky and B. Herndon,** High–Performance Parallel Computing in Industry, *Parallel Computing*, Vol. 23, 1217–1233, 1997.

[28] **A.L. Evans and D.P. Miller**, NASA IGES and NASA-IGES NURBS-Only Standard, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 31-1–31-20.

[29] **G. Fagg and J. Dongarra,** PVMPI: An Integration of PVM and MPI Systems, Calculateurs Parallels, Vol. 8, No. 2, 151–166, 1996.

[30] **C. Farhat,** A simple and efficient automatic F.E.M. domain decomposer, *Computer and Structures*, 28(2):579–602, 1988.

[31] **C. Farhat and M. Lesoinne,** Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics, *International Journal for Numerical Methods in Engineering*, Vol. 36, 745–764, 1993.

[32] **C. Farhat, S. Lanter and H.D. Simon,** TOP/DOMDEC–A Software Tool for Mesh Partitioning and Parallel Processing, *Computing Systems in Engineering*, Vol. 6, No.1, 13–26, 1995.

[33] **G. Farin,** *Curves and surfaces for Computer-Aided Geometric Design a practical guide*, Academic Press, 1993, Third Edition, ISBN 0-12-249052-5.

[34] **D.F. Ferguson,** Spline Geometry: A Numerical Analysis View, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 27-1–27-24.

[35] **D.A. Field,** Laplacian Smoothing and Delaunay Triangulations, *Communications in Applied Numerical Methods*, Vol. 4, 709–712, 1988.

[36] **D.A. Field,** The Legacy of Automatic Mesh Generation From Solid Modelling, *Computer Aided Geometric Design*, Vol. 12, No.7, 651–673, 1995.

[37] **R. Fletcher,** *Practical Methods of Optimization,* John Wiley & Sons, Chichester 1987.

[38] **L. Formaggia,** Data Structures For Unstructured Mesh Generation, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 14-1–14-20.

[39] **G.E. Forsythe and W.R. Wasow,** *Finite Difference Methods for Partial Differential Equations*, New York, Wiley, 1960.

[40] **I. Foster,** *Designing and Building Parallel Programs*, Pub. Addison-Wesley, 1995, ISBN 0-201-57594-9.

[41] **W.H. Frey and D.A. Field,** Mesh Relaxation: A New Technique for Improving Triangulations, *International Journal for Numerical Methods in Engineering*, Vol. 31, 1121–1133, 1991.

[42] **P.J. Frey, H. Borouchaki and P.L. George,** 3D Delaunay Mesh Generation Coupled with an Advancing-Front Approach, *Computer Methods in Applied Mechanics and Engineering*, Vol. 157, No.1-2, 115–131, 1998.

[43] **A. Gaither, D. Marcum, D. Reese and N.P. Weatherill,** A Paradigm for Parallel Unstructured Grid Generation, In:*Proc. 5th Int. Conf. on Numerical Grid Generation in Computational Field Simulation*, Pub. NSF Engineering Research Centre for Computational Field Simulation,731–740, 1996.

[44] **J. Galtier and P.L. George,** Prepartitioning as a Way to Mesh Subdomains in Parallel, *5th International Meshing Roundtable*, Pittsburgh, PA, 107–121, 1996.

[45] **A. Geist, A. Beguelin, J. Dongara, W. Jiang, R. Manchek, and V. Sunderam,** *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[46] **A. Geist, J. Kohl and P. Papadopoulos** PVM and MPI: a Comparison of Features, *Calculateurs Parallels*, Vol. 8, No. 2, 137–150, 1996.

[47] **A. Geist,** Advanced Tutorial on PVM 3.4 New Features and Capabilities, Proceedings of EuroPVM-MPI'97, Cracow Poland, Springer Verlag, November 1997.

[48] **P. L. George,** *Automatic Mesh Generation, Application to Finite Element Methods*, John Wiley & Sons, Paris, 1991, ISBN 0-471-93097-0.

[49] **P. L. George, F. Hecht and E. Saltel,** Automatic Mesh Generator with Specified Boundary, *Computer Methods in Applied Mechanics and Engineering*, Vol. 92, 269–288, 1991.

[50] **P. L. George,** Improvements on Delaunay–Based Three–Dimensional Automatic Mesh Generator, *Finite Elements In Analysis and Design*, Vol. 25, 297–317, 1997.

[51] **W. Gropp, E. Lusk, and A. Skjellum,** *Using MPI: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, Cambridge, Massachusetts. London, England, 1994.

[52] **W. Gropp, E. Karrels and E. Lusk,** MPE graphics – Scalable X11 Graphics in MPI, *Proc. Scalable Parallel Libraries Conf.*, Mississippi State University (IEEE Computer Society Press, Silver Spring, MD, 1994) 49–54.

[53] **W. Gropp and E. Lusk,** User's Guide for MPICH, a Portable Implementation of MPI, Tech. Report,*Mathematics and Computer Science Division, Argonne National Laboratory*, ANL-96/6, 1996.

[54] **W. Gropp, W. Lusk, N. Doss and A. Skjellum,** A High-performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol. 22, 789–828, 1996.

[55] **W. Gropp and E. Lusk,** Why are PVM and MPI so Different?, In*Recent Advances in Parallel Virtual Machine and MEssage Passing*

Interface, Volume 1332 of Lecture Notes in Computer Science 4^{th} European PVM/MPI Users's Group Meeting, Cracow, Poland, 3–10, Springer Verlag, 1997.

[56] **O. Hassan, N. P. Weatherill, J. Peraire, J. Peiro, K. Morgan and E. J. Probert,** Generation and Adaption of Unstructured Meshes, Dept. Report,*University of Wales Swansea, Dept. of Civil Eng., UK*, CR/872/95, 1995.

[57] **O. Hassan, K. Morgan, E. J. Probert, and J. Peraire,** Unstructured Tetrahedral Mesh Generation for Three–Dimensional Viscous Flows, *International Journal for Numerical Methods in Engineering*, Vol. 39, 549–567, 1996.

[58] **O. Hassan and E.J. Probert,** Grid Control and Adaptation, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 35-1–35-29.

[59] **O. Hassan, E.J. Probert, K. Morgan and N. P. Weatherill,** Unsteady flow simulation using unstructured meshes, *Computer Methods in Applied Mechanics and Engineering*, Vol. 189, 1247–1275, 2000.

[60] **J. Häuser, and C. Taylor,** *Proceeding of the 1st International Conference on Grid Generation*, Landshut, Germany, Pub. Pineridge Press, Uk, 1986.

[61] **J. Häuser, P.R. Eiseman, Y. Xia and Z. Cheng,** Parallel Multiblock Structured Grids, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 12-1–12-26.

[62] **C. Hirsch,** *Numerical Computation of Internal and External Flows*, Vol. 1 Fundamentals of Numerical Discretization 1988, Vol. 2 Computational Methods for Inviscid and Viscous Flows 1990 John Wiley& Sons.

[63] **D.C. Hodgson and P.K. Jimack,** Efficient Parallel Generation of Partitioned Unstructured Meshes, *Advances in Engineering Software*, Vol. 27, 59–70, 1996.

[64] **J. Hoschek and D. Lasser,** translated by **L. L. Schumaker,** *Fundamentals of Computer Aided Geometric Design,* A K Peters, 1993, ISBN 1-56881-007-5.

[65] **S.H. Hsieh, G.H. Paulino and J.F. Abel,** Evaluation of Automatic Domain Partitioning Algorithms for Parallel Finite Element Analysis, *International Journal for Numerical Methods in Engineering*, Vol. 40, 1025–1051, 1997.

[66] **Y.F. Hu, R.J. Blake and D.R. Emerson,** An Optimal Dynamic Load Balancing Algorithm, *Concurrency: Practice and Experience*, Vol. 10, 467–483, 1998.

[67] **Y.F. Hu and R.J. Blake,** An Improved Diffusion Algorithm for Dynamic Load Balancing, *Parallel Computing*, Vol. 25, 417–444, 1999.

[68] **K. Hwang and Z. Xu,** *Scalable Parallel Computing: Technology, Architecture, Programming,* WCBMcGraw–Hill, 1998. ISBN 0-07-031798-4

[69] **P.K. Jimack and N. Touheed,** An Introduction to MPI for Computational Mechanics, In:*Parallel and Distributed Processing for Computational Mechanics, Systems and Tools*, editor, B.H.V. Topping, Pub. Saxe–Coburg, 207–223, 1999.

[70] **Z. Johan, K.K. Mathur, S.L. Johnsson and T.J. Hughes,** Scalability of Finite Element Applications on Distributed Memory Parallel Computers, *Computer Methods in Applied Mechanics and Engineering*, Vol. 119, 61–72, 1994.

[71] **J. Jones and N.P.Weatherill,** The Visualisation of Large Unstructured Grid Data Sets, In: *Proc. 6th Int. Conf. Numerical Grid Generation in Computational Field Simulations*, editors: M. Cross, P.R. Eiseman, J. Häuser, B. K. Soni and J.F. Thompson, published by International Society of Grid Generation (ISGG), 899–909, 1998.

[72] **E. Karrels and E. Lusk,** Performance Analysis of MPI Programs, In:*Environments and Tools for Parallel Scientific Computing*, Editors, J. Dongarra and B. Tourancheau, 195–200, SIAM, Philadelphia, PA, 1994.

[73] **G. Karypis and V. Kumar,** A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, Tech. Report *University of Minnesota, Department of Computer Science*, 95–035, 1995.

[74] **G. Karypis and V. Kumar,** Analysis of Multilevel Graph Partitioning, Tech. Report *University of Minnesota, Department of Computer Science*, 95–037, 1995.

[75] **G. Karypis and V. Kumar,** A Coarse–Grain Parallel Formulation of Multilevel *K-way* Graph Partitioning Algorithm, In:*Parallel Processing for Scientific Computing* SIAM, Philadelphia, M. Heath, editor, 1997.

[76] **Y. Kono, M. Seto, K. Nishimatsu, H. Fukumori and Y. Muraoka,** Parallel Mesh Generation of Finite Element Method - Parallel Constructing of Voronoi Diagram, *IPSJ, SIGNotes High Performance Computing*, No.060-008, 1995.

[77] **D. Knuth,** *The Art Of Computer Programming Sorting And Searching, Vol. 3,* Addison-Wesley, Reading, MA, 1973.

[78] **L. Lämmer and M. Burghardt,** Parallel Generation of Triangular and Quadrilateral Meshes, *Advances in Engineering Software*, Vol. 31, No.12, 929-936, 2000.

[79] **M. Lauria and A. Chien,** MPI–FM: High Performance MPI on Workstation Clusters, *Journal of Parallel and Distributed Computing*, Vol. 40, No.1, 4-18 1997.

[80] **S.Y. Lee, C.I. Park and C.M. Park,** An Improved Parallel Algorithm for Delaunay Triangulation on Distributed Memory Parallel Computers, *Advances in Parallel and Distributed Computing - Proceedings*, Ch.58, 131-138, 1997.

[81] **R. Löhner and P. Parikh,** Three–Dimensional Grid Generation by Advancing Front Method, *International Journal For Numerical Methods In Fluids*, Vol. 8, 1135–1149, 1988.

[82] **R. Löhner, J. Camberos and M. Merriam,** Parallel Unstructured Grid Generation, *Computer Methods in Applied Mechanics and Engineering*, Vol. 95, 343–357, 1992.

[83] **R. Löhner and R. Ramamurti,** A Load Balancing Algorithm for Unstructured Grids, *International Journal of Computational Fluid Dynamics*, Vol. 5, No.1-2, 39–58, 1995.

[84] **R. Löhner,** Extensions and Improvements of the Advancing Front Grid Generation Technique, *Communications in Numerical Methods in Engineering*, Vol. 12, No.10, 683–702, 1996,

[85] **R. Löhner,** Regridding Surface Triangulation, *Journal of Computational Physics*, Vol. 126, 1–10, 1996.

[86] **R. Löhner,** Automatic Unstructured Grid Generators, *Finite Elements in Analysis and Design*, Vol. 25, 111–134, 1997.

[87] **K–L. Ma and T.W. Crochett,** A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data, In:*Proceeding 1997 IEEE Symposium on Parallel Rendering,* editors, K–L. Ma, J. Painter and G. Stoll, 95–104, Pub. ACM SIGGRAPH, 1997.

[88] **K–L. Ma and T.W. Crochett,** Parallel Visualisation of Large–scale Aerodynamics Calculations: A Case Study on the Cray T3E, *ICASE Report*, No. 99-41, 1999.

[89] **D.J. Mavriplis and S. Pirzadeh,** Large–scale Parallel Unstructured Mesh Computations for 3D High-lift Analysis, *NASA/CR-1999-208999 ICASE Report No. 99-9*, Feb. 1999.

[90] **D.J. Mavriplis** Large-scale parallel viscous flow computations using an unstructured multigrid algorithm, *NASA/CR-1999-209724 ICASE Report No. 99-44*, Nov. 1999.

[91] **K. Morgan, N.P. Weatherill, O. Hassan, P.J. Brooks, M.T. Manzari and R. Said,** Aerospace Engineering Simulations on Parallel Computers, In:*Frontier of Computational Fluid Dynamics*, editors, D.A. Caughy and M.M. Hafez, John Wiley & Sons, 1–15, 1997.

[92] **K. Morgan, L.B. Bayne, O. Hassan, J.E. Probert and N.P. Weatherill,** The Simulation of 3D Unsteady Inviscid Compressible Flows With Moving Boundaries, In:*Computational Science for the 21st Century*, editor, M.O. Bristeau, John Wiley & Sons, Chichester, 347–356, 1997.

[93] **K. Morgan and J. Peraire,** Unstructured Grid Finite Element Methods for Fluid Mechanics, *Reports on Progress in Physics*, Vol.61, No.6, 569–638, 1998.

[94] **K. Morgan, P.J. Brooks, O. Hassan and N.P. Weatherill,** Parallel Processing for the Simulation of Problems Involving Scattering of Electromagnetics Waves, *Computer Methods in Applied Mechanics and Engineering*, Vol. 152, 157–174, 1998.

[95] **K. Morgan, N. P. Weatherill, O. Hassan, P. J. Brooks, R. Said and J. Jones,** A Parallel Framework for Multidisciplinary Aerospace Engineering Simulations Using Unstructured Meshes, *International Journal For Numerical Methods In Fluids*, Vol. 31, 159–173, 1999.

[96] **A. K. Noor,** New Computing and Future High–Performance Computing Environment and their Impact on Structural Analysis and Design, *Computers & Structures,* Vol. 64, No.1–4, 1–30, 1997.

[97] **T. Okusanya and J. Peraire,** Parallel Unstructured Grid Generation, In:*Proc. 5th Int. Conf. on Numerical Grid Generation in Computational Field Simulation*, Pub. NSF Engineering Research Centre for Computational Field Simulation, 719–729, 1996.

[98] **P. Papadopoulos, E. Venkatapathy, D. Prabhu, M.P. Loomis and D. Olynick** Current Grid Generation Strategies and Future Requirements in Hypersonic Vehicle Design, Analysis and Testing, *Applied Mathematical Modelling*, Vol. 23, No.9, 705–735, 1999.

[99] **J. Peiró, J. Peraire and K. Morgan,** FELISA System Reference Manual, Dept. Report, *University of Wales Swansea, Dept. of Civil Eng., UK*, CR/821/94, 1994.

[100] **J. Peiró,** Surface Grid Generation, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 19-1–19-20.

[101] **J. Peraire, M. Vahdati, K. Morgan and O.C. Zienkiewicz,** Adaptive Remeshing for Compressible Flow Computations, *Journal of Computational Physics*, Vol. 72, 449–466, 1987

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

[103] **J. Peraire, J. Peiró, and K. Morgan,** Advancing Front Grid Generation, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 17-1–17-22.

[104] **L. Piegl and W. Tiller,** *The NURBS Book*, Springer, 1995. ISBN 3-540-55069-0.

[105] **W. Preilowski, E. Dahlhaus and G. Wechsung,** *New Parallel Algorithms for Convex Hull and Triangulation in 3-Dimensional Space*, Lecture Notes in Computer Science, Vol.629, 442-450, 1992.

[106] **R. Said,** *Geometry Repair in Two and Three Dimensions,* MSc Thesis, Univeristy of Wales Swansea, C/M/302/95, 1995.

[107] **R. Said, N.P. Weatherill, K. Morgan and N.A. Verhoeven,** Distributed Parallel Delaunay Mesh Generation, *Computer Methods in Applied Mechanics and Engineering*, Vol. 177, 109–125, 1999.

[108] **R. Said, B. Larwood, N.P. Weatherill, O. Hassan and K. Morgan,** Parallel Delaunay Unstructured Grid Generation, In: *Proc. 7th International Conference on Numerical Grid Generation in Computational Field Simulation*, Whistler, Canada, B.K.Soni, J. Haeuser, J.F.Thompson, P.R. Eiseman, editors ISGG, NSF ERC for CFS at Mississippi State University, 2000.

[109] **M. Saxena and R. Percchio,** Parallel FEM algorithms based on recursive spatial decomposition. I. Automatic mesh generation , *Computers and Structures*, Vol. 45, No.5-6, 817-831, 1992.

[110] **W.J. Schroeeder and M.S. Shephard,** A Combined Octree-Delaunay Method for Fully Automatic 3-D Mesh Generation, *International Journal for Numerical Methods in Engineering*, Vol. 29, 37–55, 1990.

[111] **S. Sengupta, J.F. Thompson, P.R. Eiseman and J. Häuser,** *Proceeding of the 2nd International Conference on Grid Generation, in Computational Fluid Dynamics*, Miami, USA, Pub. Pineridge Press, Uk, 1988.

[112] **H. Simon,** Partitioning of Unstructured Problems for Parallel Processing, *Computing Systems in Engineering*, Vol. 2, No. 2-3, 135–148, 1991.

[113] **M.S. Shephard, M.K. Georges,** Reliability of Automatic 3D Mesh Generation, *Computer Methods in Applied Mechanics and Engineering*,Vol. 101, No.1-3, 443–462, 1992.

[114] **M.S. Shephard, H.L. de Cougny, R.M. O'Bara and M.W. Beall,** Automatic Grid Generation Using Spatially Based Trees, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 15-1–15-21.

[115] **A. Shostko and R. Löhner,** Three–Dimensional Parallel Unstructured Grid Generation, *International Journal for Numerical Methods in Engineering*, Vol. 38, 905–925, 1995.

[116] **D. Slama, J. Garbis and P. Russell**, *Enterprise Corba* , Prentice Hall, 1999. ISBN:0130839639.

[117] **B.K. Soni, J.F. Thompson, J. Häuser and P.R. Eiseman,** *Proceeding of the 5th International Conference on Numerical Grid Generation in Computational Field Simulation*, Mississippi, USA, Pub. NSF Engineering Research Centre for Computational Field Simulation, 1996, ISBN 0-9651627-0-2.

[118] **B.K.Soni, J. Haeuser, J.F.Thompson, P.R. Eiseman,** *Proceeding of the 7th International Conference on Numerical Grid Generation in Computational Field Simulation*, Whistler, Canada, September 25-28, 2000, Pub. ISGG, NSF ERC for CFS at Mississippi State University, 2000.

[119] **K.A.Sorensen, O. Hassan, K. Morgan and N. P. Weatherill,** An Agglomerated Multigrid Hybrid Mesh Method for Compressible Flow, In proc.*the First MIT Conference on Computational Fluid and Solid Mechanics*, 2001, Boston.

[120] **V. Sundreram,** PVM: A framework for Parallel Distributed Computing, *Computing Concurrency: Practice & Experience*, Vol. 2., No. 4, 1990.

[121] **W.C. Thacker,** A Brief Review of Techniques for Generating Irregular Computational Grids, *International Journal for Numerical Methods in Engineering*, Vol. 15, 1335–1341, 1980.

[122] **J.F. Thompson, Z.U.A. Warsi and C.W. Mastin,** *Numerical Grid Generation, Foundations and Applications*, North Holland pub. 1985. ISBN

[123] **J.F. Thompson,** A General Three Dimensional Elliptic Grid Geeration System on a Composite Block-Structur, *Computer Methods in Applied Mechanics and Engineering*, Vol. 64, 1987.

[124] **J.F. Thompson, B.K. Soni and N.P. Weatherill,** Editors, *Handbook of Grid Generation*, CRC Press, 1999. ISBN 0-8493-2687-7.

[125] **N. Touheed, P. Selwood, P.K. Jimak and M. Berzins,** *A Comparison of Some Dynamic Load-Balancing Algorithms for a Parallel Adaptive Flow Solver*, Submitted to Parallel Computing, 1999.

[126] **A. Trew and G. Wilson,** Editors, *Past, Present, Parallel,* A Survey of Available Parallel Computing Systems, Springer-Verlag, 1991. ISBN 0-387-19664-1.

[127] **P.L. Vaughan, A. Skjellum, D.S. Reese and F.C. Cheng,** Migration from PVM to MPI, Part I: The Unify System, *Proc. 5th Symp.. on the Frontiers of Massively Parallel Computation*, McLean, VA, IEEE Computer Society Technical Committee on Computer Architecture (IEEE Computer Society Press, Silver Spring, MD, 1995) 488–495.

[128] **N. A. Verhoeven, N. P. Weatherill, and K. Morgan,** Dynamic Load Balancing in a 2D Parallel Mesh Generator, In A. Ecer, J. Periaux, N. Satofuko, and S. Taylor, editors, *Parallel Computational Fluid Mechanics: Implementation and Results Using Parallel Computers*, 641–648. Elsevier Science B.V., 1995.

[129] **N.A. Verhoeven, R. Said, N.P. Weatherill, and K. Morgan,** Delaunay Mesh Generation On Distributed Computer Platforms, *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools.* B.H.V. Topping editor, 77–90, Saxe-Coburg Publications, 1997, ISBN 1-874672-03-2.

[130] **G. Voronoi,** Nouvelles Applications des Paramètres Continues *à* la Thèorie des Formes Quadratiques Dieuxieme Memoire: Researches sur les Parallelloedres Primitif, *Journal Reine Angew. Mathematics*, Vol. 134, 198–205, 1908.

[131] **D.W. Walker and J.J. Dongara,** MPI - A Standard Message-Passing Interface, *Supercomputer*, Vol. 12, No.1, 56–68, 1996.

[132] **C. Walshaw, M. Cross and M. Everett,** Parallel Dynamic Graph Partitioning for Unstructured Meshes, Tech. Report *University of Greenwich, Centre for Numerical Modelling and Process Analysis*, 97/IM/20, 1997.

[133] **C. Walshaw, M. Cross and M. Everett,** Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes, *Journal of Parallel and Distributed Computing*, Vol. 47, No.2, 102–108, 1997.

[134] **C. Walshaw and M. Cross,** Parallel Optimisation Algorithms for Multilevel Mesh Partitioning, Tech. Report *University of Greenwich, Centre for Numerical Modelling and Process Analysis*, 99/IM/44, 1999.

[135] **D. F. Watson,** Computing the N–Dimensional Delaunay Tessellations with Application to Voronoi Polytopes, *Computers Journal*, Vol. 24, No.2, 167–172, 1981.

[136] **N.P. Weatherill,** A Method For Generating Irregular Computational Grids In Multiply Connected Planar Domains, *International Journal for Numerical Methods in Fluids*, Vol. 8, 181–197, 1988.

[137] **N.P. Weatherill,** Delaunay Triangulation in Computational Fluid Dynamics, *Computers & Mathematics with Applications*, Vol. 24, No. 5–6, 129–150, 1992.

[138] **N.P. Weatherill, P.R. Eiseman, J. Häuser and J.F Thompson,** *Proceeding of the 4th International Conference on Grid Generation, in Computational Fluid Dynamics and Related Fields*, Swansea, UK, Pub. Pineridge Press, Uk, 1994.

[139] **N.P. Weatherill, O. Hassan,** Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints, *International Journal for Numerical Methods in Engineering*, Vol. 37, 2005–2039, 1994.

[140] **N.P. Weatherill, O. Hassan, M.J. Marchant and D.L. Marcum,** Grid Adaptation Using a Distribution of Sources Applied to Inviscid Compressible Flow Simulation, *International Journal For Numerical Methods In Fluids*, Vol. 19, 739–764, 1994.

[141] **N.P. Weatherill,** The Delaunay Triangulation – From The Early Work in Princeton, *Frontiers of Computational Fluid Dynamics*, D.A. Caughy and M.M. Hafez, editors, 83–100, John Wiley & Sons, 1994.

[142] **N.P. Weatherill, O. Hassan and D. L. Marcum,** A Compressible Flow–field Solutions with Unstructured Grids Generated by Delaunay Triangulation, *AIAA Journal*, Vol. 33, No. 7, 1995.

[143] **N.P. Weatherill,** The Reconstruction of Boundary Contours and Surfaces in Arbitrary Unstructured triangular and Tetrahedral Grids, *Engineering Computations*, Vol. 13, No. 8, 64–79, 1996.

[144] **N.P. Weatherill, M. J. Marchant, E. Turner-Smith, Y. Zheng, and M. Sotirakis,** The Design of a Graphical User Environment for Multi–Disciplinary Computational Engineering, In *Proceedings of the ECCOMAS'96 Conference*, Paris, France, September 1996.

[145] **N.P. Weatherill, R. Said and K. Morgan,** The Construction of Large Unstructured Grids by Parallel Delaunay Grid Generation, Invited paper in Proc. 6th Int. Conf. Numerical Grid Generation in Computational Field Simulations, editors: M. Cross, P.R. Eiseman, J. Häuser, B. K. Soni and J.F. Thompson, published by International Society of Grid Generation (ISGG), 53–73, 1998.

[146] **N.P. Weatherill, K. Morgan, O. Hassan, P.J. Brooks, R. Said, J. Jones and M.T. Manzari,** A Parallel Framework For Computational Fluid Dynamics and Computational Electromagnetics, In:*Proceeding of the International Workshop on New Models and Numerical Codes for Shock Wave Processes in Condensed Media*, (in press), 1999.

[147] **N.P. Weatherill,** Unstructured Grids: Procedures and Applications, In:*Handbook of Grid Generation*, J.F. Thompson, B.K. Soni and N.P. Weatherill, editors, (CRC Press, 1999), 26-1–26-36.

[148] **N.P. Weatherill, K. Morgan, O. Hassan and R. Said,** The Construction of Large Unstructured Grids by Parallel Delaunay Grid Generation, *von Karman Institute for Fluid Dynamics, Lecture Series 2000-05,* Brussels, March, 2000.

[149] **N.P. Weatherill, O. Hassan, K Morgan; J.W. Jones and B Larwood,** Towards Fully Parallel Aerospace Simulations on Unstructured Meshes, *Engineering Computations*, Vol.18, No.3-4, 347–376, 2001.

[150] **R.D. Williams,** Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations, *Concurrency: Practice and Experience*, Vol. 3, 157–181, 1991.

[151] **M.A. Yerry and M.S. Shephard,** Automatic Mesh Generation for 3-Dimensional Solids, *Computers & Structures*, Vol.20, No.1-3, 31–39, 1985.

[152] **O.C. Zienkiewicz and D.V. Phillips,** An Automatic Mesh Generation Scheme For Plane and Curved Surfaces by Isoparametric Coordinates, *International Journal for Numerical Methods in Engineering*, Vol. 3, 519–528, 1971.

[153] **O.C. Zienkiewicz and R.L.Taylor,** *The Finite Element Method*, 4th edition, Vol. 1 Basic Formulation and Linear Problems, Vol. 2 Solid and Fluid Mechanics Dynamics and Non–Linearity, McGraw Hill, 1994.

[154] **B.G. Larwood,** *Mesh generation for large-scale and complex computational simulation* PhD thesis, University of Wales Swansea, 2003.