12 Parallel Multiblock Structured Grids

Jochem Häuser
Peter Eiseman
Yang Xia
Zheming Cheng

12.1 Overview

In this overview the lesson learned from constructing 3D multiblock grids for complex geometries are presented, along with a description of their interaction with fluid dynamics codes used in parallel computing. A brief discussion of the remaining challenging problems is given, followed by an outlook of what can be achieved within the next two or three years in the field of parallel computing in aerospace combined with advanced grid generation.

The overall objective of this chapter is to provide parallelization concepts independent of the underlying hardware — regardless whether parallel or sequential — that are applicable to the most complex topologies and flow physics. At the same time, the solver must remain efficient and effective. An additional requirement is that once a grid is generated, the flow solver should run immediately without any further human interaction.

The field of CFD (computational fluid dynamics) is rapidly changing and is becoming increasingly sophisticated: grids define highly complex geometries, and flows are solved involving very different length and time scales. The solution of the Navier–Stokes equations must be performed on parallel systems, both for reasons of overall computing power and cost effectiveness.

Complex geometries can either by gridded by completely unstructured grids or by structured multiblock grids. In the past, unstructured grid methods almost exclusively used tetrahedral elements. As has been

0-8493-2687-7/99/$0.00+$.50

shown recently this approach has severe disadvantages with regard to program complexity, computing time, and solution accuracy as compared to hexahedral finite volume grids [Venkatakrishnan, 1994]. Multiblock grids that are unstructured on the block level but structured within a block provide the geometrical flexibility and retain the computational efficiency of finite difference methods. Consequently, this technique has been implemented in the majority of the flow solvers.

In order to have the flow solution independent of the block topology, grids must be slope continuous. This causes a certain memory overhead: if n is the number of internal points in each direction for a given block, this overhead is the factor $(n + 2)^3/n^3$, where an overlap of two rows or columns has been assumed. The overhead is mainly caused by geometrical complexity, i.e., generating a block topology that aligns the flow with the grid as much as possible requires a much more sophisticated topology.

Since grid topology is determined by both the geometry and the flow physics, blocks are disparate in size, and hence load balancing is achieved by mapping a group of blocks to a single processor. The message passing algorithm — we will specify our parallelization strategy in more detail in Section 12.8 — must be able to efficiently handle the communication between blocks that reside on the same processor, meaning that only a copy operation is needed. For message passing, only standard library packages are used, namely Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). Communication is restricted to a handful of functions that are encapsulated, thus providing full portability. A serial machine is treated as a one-processor parallel machine without message passing. Available parallelism (the maximum number of processors that can be used for a given problem) is determined by the number of points in the grid: a tool is available to split large blocks, if necessary. Grids generated employ NASA's standard Plot3D format.

In particular, a novel numerical solution strategy has been developed to solve the 3D N–S equations for arbitrary complex multiblock grids in conjunction with complex physics on parallel or sequential computer systems.

In general, numerical methods are of second order in space. A set of two ghost cells in each direction exists for each block, and parallelization is simply introduced as a new type of boundary condition. Message passing is used for updating ghost cells, so that each block is completely independent of its neighbors. Since blocks are of different size, several blocks are mapped onto a single processor to achieve almost always a perfect static load balancing. This implementation enables the code to run on any kind of distributed memory system, workstation cluster, massively parallel system as well as on shared memory systems and in sequential mode.

A comprehensive discussion of the prevailing concepts and experiences with respect to load balancing, scaling, and communication is presented in this article. Extensive computations employing multiblock grids have been performed to investigate the convergence behavior of the Newton-CG (conjugate gradient) scheme on parallel systems, and to measure the communication bandwidth on workstation clusters and on large parallel systems.

There can be no doubt that the future of scientific and technical computing is parallel. The challenging tasks to be tackled in the near future are those of **numerical scaling** and of **dynamic load balancing.** Numerical scaling means that the computational work increases with $O(N)$ or at most $O(N\log N)$ where N denotes problem size. This is normally not the case. A simple example, the inversion of a matrix with N elements, needs an order of $O(N^3)$ floating point operations. For instance, increasing the problem size from 100,000 points to 10 million points would increase the corresponding computing time a million times. Obviously, no parallel architecture could keep pace with this computational demand.

Therefore, one of the most challenging tasks is the development of algorithms that scale numerically. The so-called **tangled web** approach (see Section 12.10), based on the idea of adaptive coupling between grid points during the course of a computation, is an important technique that might have the potential to achieve this objective. It should be clear that in order to obtain numerical scaling, this tangled web approach will not result in an algorithm that is scalable and parallel. This is due to the high load imbalance that may be caused during the computation, based on the dynamic behavior of the algorithm. Thus, this approach is inherently dynamic and therefore needs dynamic load balancing. Only if these two requirements are satisfied can **very large scale applications** (VLSA) be computed in CFD.

From the lessons learned so far it can be confidently predicted that the techniques are available both for numerical scaling and dynamic load balancing for these VLSA. It remains to implement the basic ideas in such a way that routine computations for the complex physics and the complex geometries that characterize today's aerospace design can be performed at the increased level of accuracy demanded by the CFD applications of the future.

12.2 Multiblock Grid Generation and Parallelization

Structured grids use curvilinear coordinates to produce a body-fitted mesh. This has the advantage that boundaries can be exactly described and hence boundary conditions can be accurately modeled. In early grid generation it was attempted to always map the physical solution domain to a single rectangle or a single box in the computational domain. For multiply connected solution domains, branch cuts had to be introduced, a procedure well known in complex function theory and analytic mapping of 2D domains, e.g., for the Joukowski airfoil. However, it became soon obvious that certain grid line configurations could not be obtained. If one considers, for example, the 2D flow past an infinitely long cylinder with a small enough Reynold number, it would be advantageous if the grid line distribution would be similar to the streamline pattern. A 2D grid around a circle which is mapped on a single rectangle necessarily has O-type topology, unless additional slits (double-valued line or surface) or slabs (blocks that are cut out of the solution domain) are introduced. However in this case, the main advantage of the structured approach, namely that one has to deal logically only with a rectangle or a box, that is, the code needs only two or three "*for*" loops (C language), no longer holds. The conclusion is that this structuredness is too rigid, and some degree of unstructuredness has to be introduced. From differential geometry the concept of an atlas consisting of a number of charts is known. A set of charts covers the atlas where charts may be overlapping. Each chart is mapped onto a single rectangle. In addition, now the connectivity of the charts has to be determined. This directly leads to the multiblock concept, which provides the necessary geometrical flexibility and the computational efficiency of the finite volume or finite difference techniques used in most CFD codes.

For a vehicle like the Space Shuttle a variety of grids can be constructed. One can start with a simple monoblock topology that wraps around the vehicle in an O-type fashion. This always leads to a singular line, which normally occurs in the nose region. This line needs special treatment in the flow solution. It has been observed that the convergence rate is reduced; however, special numerical schemes have been devised to alleviate this problem. Furthermore, a singularity invariably leads to a clustering of grid points in an area where they are not needed. Hence, computing time may be increased substantially. In addition, with a monoblock mesh, gridline topology is fixed and additional requirements with regard to grid uniformity and orthogonality cannot be matched. However, with a multiblock mesh, a grid has to be smooth across block boundaries.

Since multiblock grids are unstructured at the block level, information about block connectivity is needed along with six faces of each block. For reasons of topological flexibility it is mandatory that each block has its own local coordinate system. Hence blocks are rotated with respect to each other. Slope continuity of grid lines across neighboring block boundaries, for instance as shown in Figures 12.1 and 12.2, is achieved by overlapping edges (2D) or faces (3D). For grid generation an overlap of exactly one row or one column is necessary (see Figure 12.8). A flow solver that retains second-order accuracy, however, needs an overlap of two rows or columns.

The solution domain is subdivided into a set of blocks or segments (in the following the words block and segment are used interchangeably). The multiblock concept, used as a domain decomposition approach, allows the direct parallelization of both the grid generation and the flow codes on massively parallel systems. Employment of the overlap feature directly leads to the message passing concept, i.e., the exchange of faces between neighboring blocks.

Each (curvilinear) block in the physical plane is mapped onto a Cartesian block in the computational plane (see Figure 12.3). The actual solution domain on which the governing physical equations are solved is therefore a set of connected, regular blocks in the computational plane. However, this does not mean

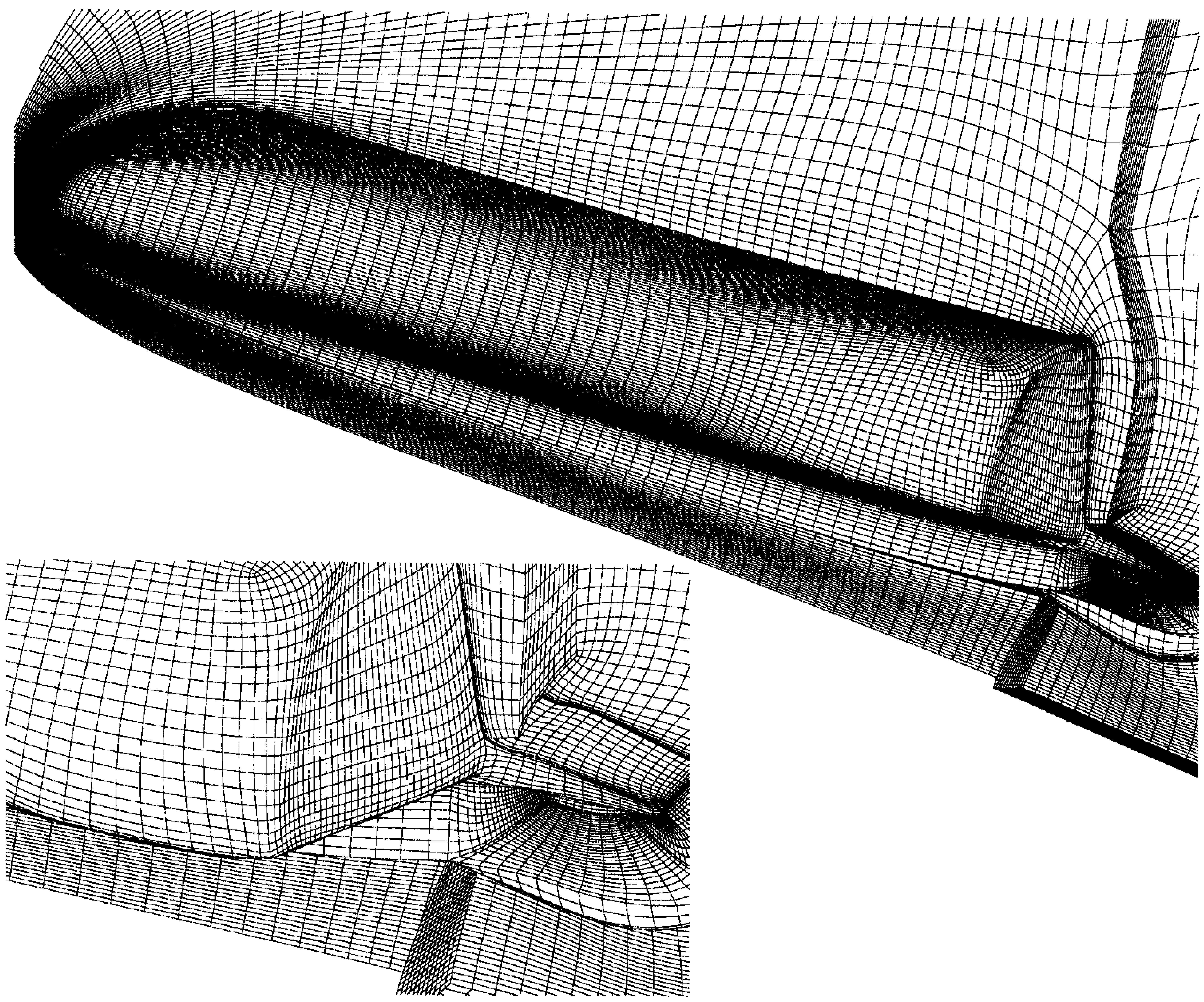

FIGURE 12.1 Halis Space Shuttle grid with local three dimensional clustering around the body flap. This clustering leads to high resolution at the body flap, but prevents the extension of this resolution into the farfield, thus substantially reducing the number of grid points. The grid is generated fully automatically, once the basic wireframe topology has been given. This procedure leads to a more complex topology and to blocks that are of different size. Vector computers that need long vector lengths will perform poorly on this topology. On the other hand, parallel machines in combination with the parallelization strategy described in this article, will give a high parallel efficiency.

that the solution domain in the computational plane has a regular structure, rather it may look fragmented. Therefore, an important point is that the parallelization approach must not rely on a nearest neighbor relation of the blocks. Hence, communication among blocks follows a random pattern. A parallel architecture based on nearest neighbor communication, e.g., for lattice gauge problems, will not perform well for complex aerodynamic applications, simply because of communication overhead, caused by random communication. However, as we will see in Section 12.9, communication time is not a problem for implicit VFD codes, but load balancing is a crucial issue.

The grid point distribution within each block is generated by the solution of a set of three Poisson equations, one for each coordinate direction, in combination with transfinite interpolation and grid quality optimization (cf. Chapter 4). In this context, a grid point is denoted as a boundary point if it lies on one of the six faces of a block in the computational plane. However, one has to discern between physical boundary points on fixed surfaces and matching boundary points on overlap surfaces of neighboring blocks. The positions of the latter ones are not known *a priori* but are determined in the solution process. Parallelization, therefore, in a multiblock environment simply means the introduction of a new boundary condition for block interfaces. Even on a serial machine, block updating has to be performed. The only difference is that a copy from and to memory can be used, while on a parallel system block updating is performed via message passing (PVM or MPI). The logic is entirely the same, except for the additional packing and sending of messages.

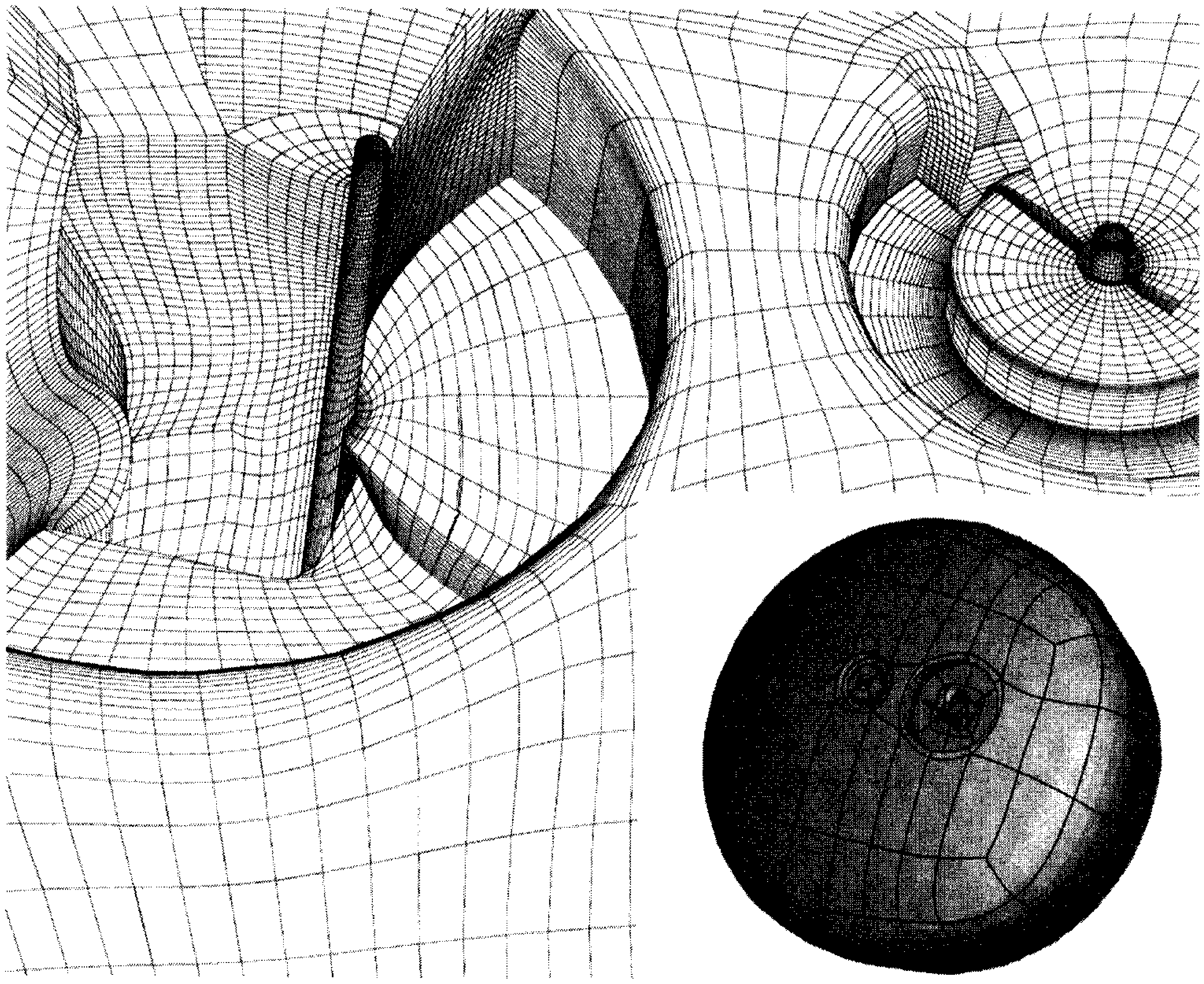
FIGURE 12.2 Cassini–Huygens Space Probe grid. The probe will enter Titan's atmosphere in 2004 to measure its composition. On the windward side several instruments are shown, leading to microaerodynamics phenomena [Bruce, 1995]. The grid comprises a complex topology, consisting of 462 blocks that are of different size.

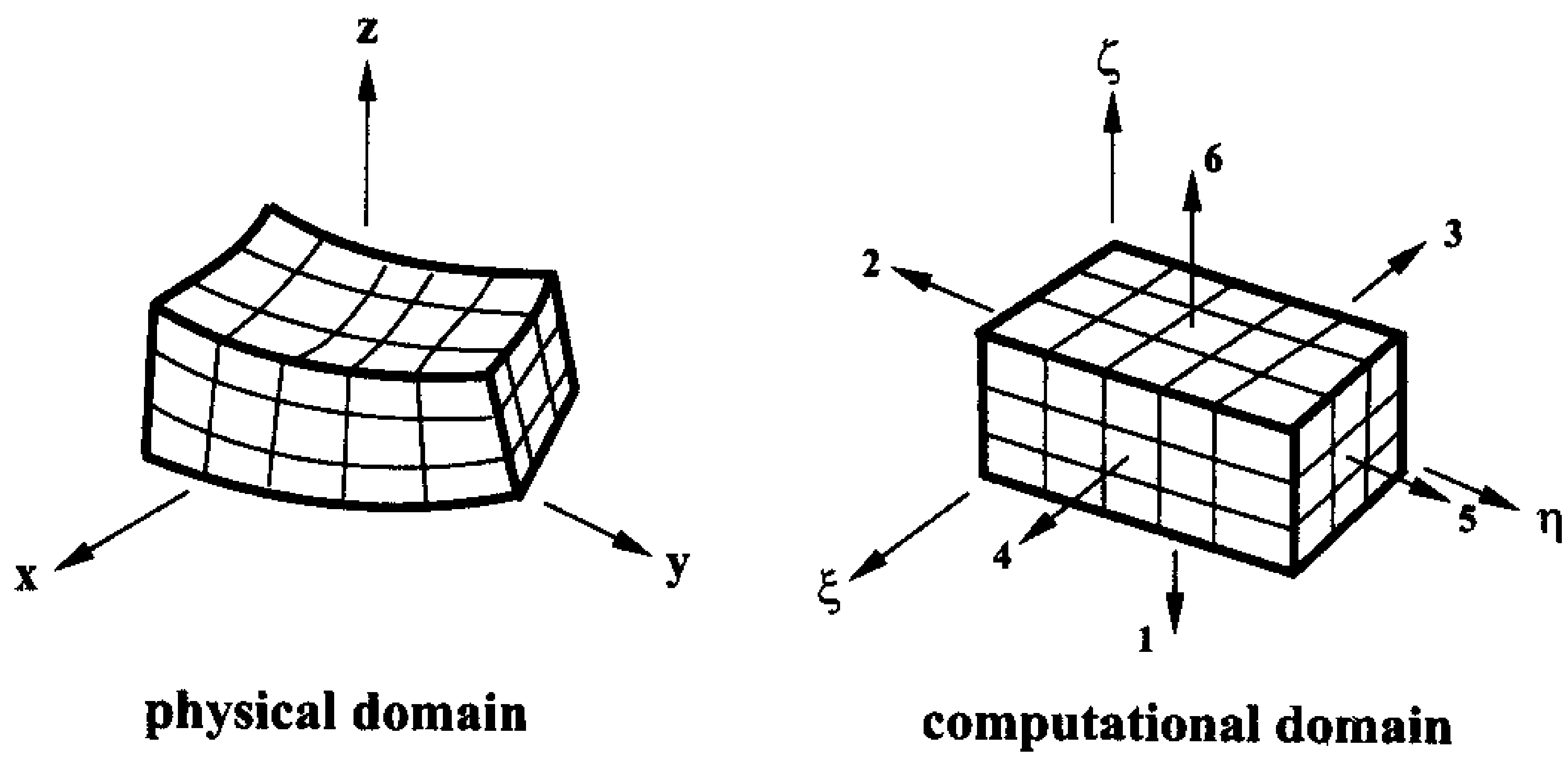

FIGURE 12.3 Mapping of a block from solution domain to computational domain. Arrows indicate orientation of faces, which are numbered in the following way: 1 bottom, 2 left, 3 back, 4 front, 5 right, 6 top. The rule is that plane $\zeta = 1$ corresponds to 1, plane $\eta = 1$ to 2, and plane $\xi = 1$ to 3.

12.3 Computational Aspects of Multiblock Grids

As has been discussed previously, boundary-fitted grids have to have coordinate lines, i.e., they cannot be completely unstructured. In CFD in general, and in high speed flows in particular, many situations are encountered for which the flow in the vicinity of the body is aligned with the surface, i.e., there is a prevailing flow direction. This is especially true in the case of hypersonic flow because of the high kinetic energy. The use of a structured grid, allows the alignment of the grid, resulting in locally 1D flow. Hence,

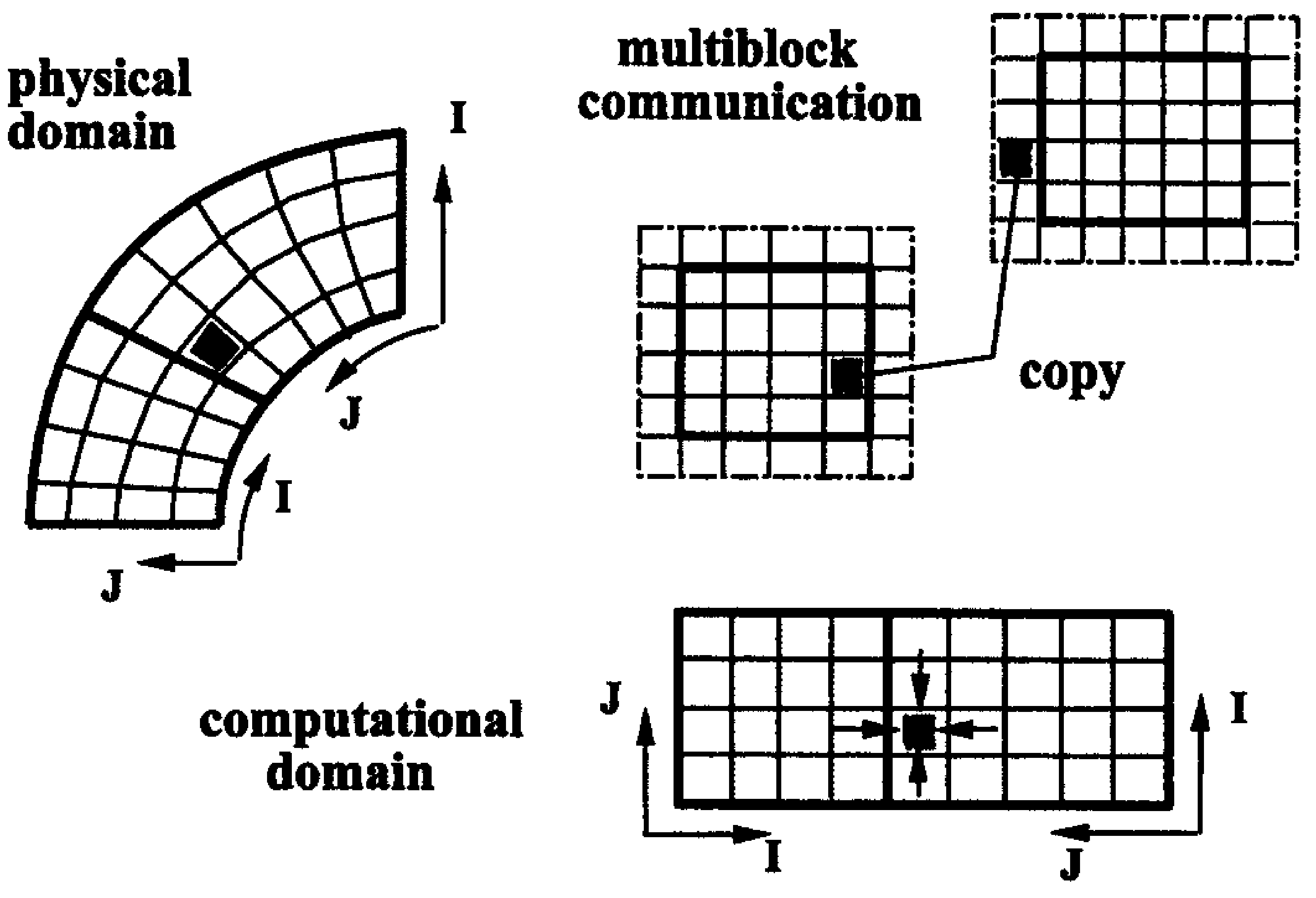

FIGURE 12.4 Multiblock grids are constructed using an overlap of one row or column. The information from an internal cell of the neighboring block is transferred via message passing (or memory copying) in the overlap cell of the current block.

numerical diffusion can be reduced, i.e., better accuracy can be achieved. In the present approach, a solution domain may be covered by hundreds or thousands of blocks. Second, structured grids can be made orthogonal at boundaries and almost orthogonal within the solution domain, facilitating the implementation of boundary conditions and also increasing numerical accuracy. This will be of advantage when turbulence models are employed using an almost orthogonal mesh. In the solution of the Navier–Stokes equations, the boundary layer must be resolved. This demands that the grid is closely wrapped around the body to describe the physics of the boundary layer (some 32 layers are used in general for structured grids). Here some type of structured grid is indispensable. For instance, the resolution of the boundary layer leads to large anisotropies in the length scales of the directions along and off the body. Since the time-step size in an explicit scheme is governed by the smallest length scale or, in the case of reacting flow, by the magnitude of the chemical production terms, extremely small time steps will be necessary. This behavior is not demanded by accuracy considerations but to retain the stability of the scheme. Thus, implicit schemes will be of advantage. In order to invert the implicit operator, a structured grid produces a regular matrix, and thus makes it easier to use a sophisticated implicit scheme.

12.4 Description of the Standard Cube

A formal description of block connectivity is needed to perform the block updating, i.e., to do the message passing. To this end, grid information is subdivided into topology and geometry data. The following format is used for both the grid generator and the flow solver, using the same topology description. All computations are done for a standard cube in the computational plane as shown in Figure 12.5. The coordinate directions in the computational plane are denoted by ξ, η, and ζ, and block dimensions are given by I, J, and K, respectively.

In the computational plane, each cube has its own right-handed coordinate system (ξ, η, ζ), where the ξ direction goes from back to front, the η direction from left to right, and the ζ direction from bottom to top, see Figure 12.5. The coordinate values are given by proper grid point indices i, j, k in the ξ, η, ζ directions, respectively. That means that values range from 1 to I in the ξ direction, from 1 to J in the η direction, and from 1 to K in the ζ direction. Each grid point represents an integer coordinate value in the computational plane.

A simple notation of planes within a block can be defined by specifying the normal vector along with the proper coordinate value in the specified direction. For example, face 2 can be uniquely defined by

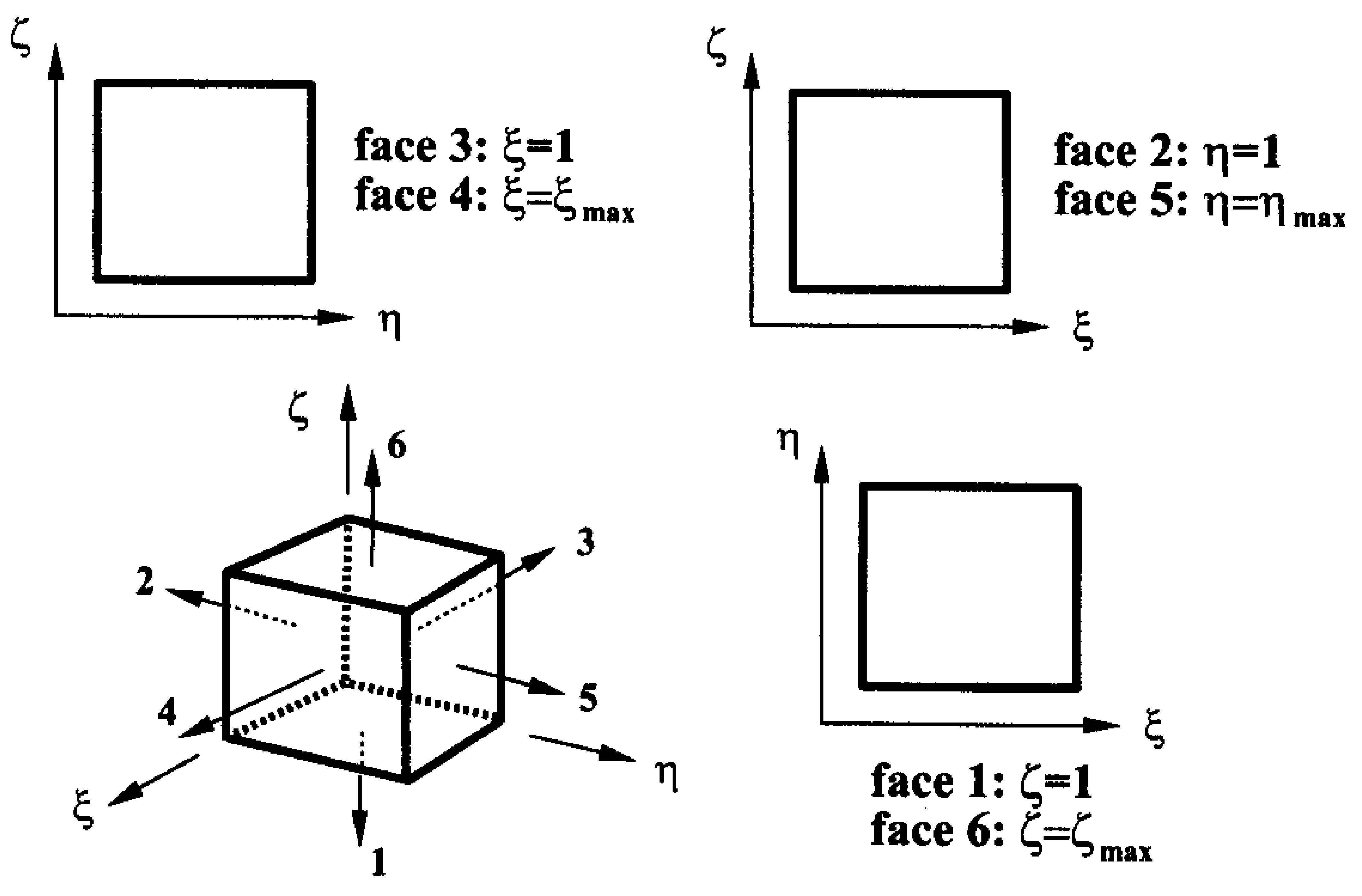

FIGURE 12.5

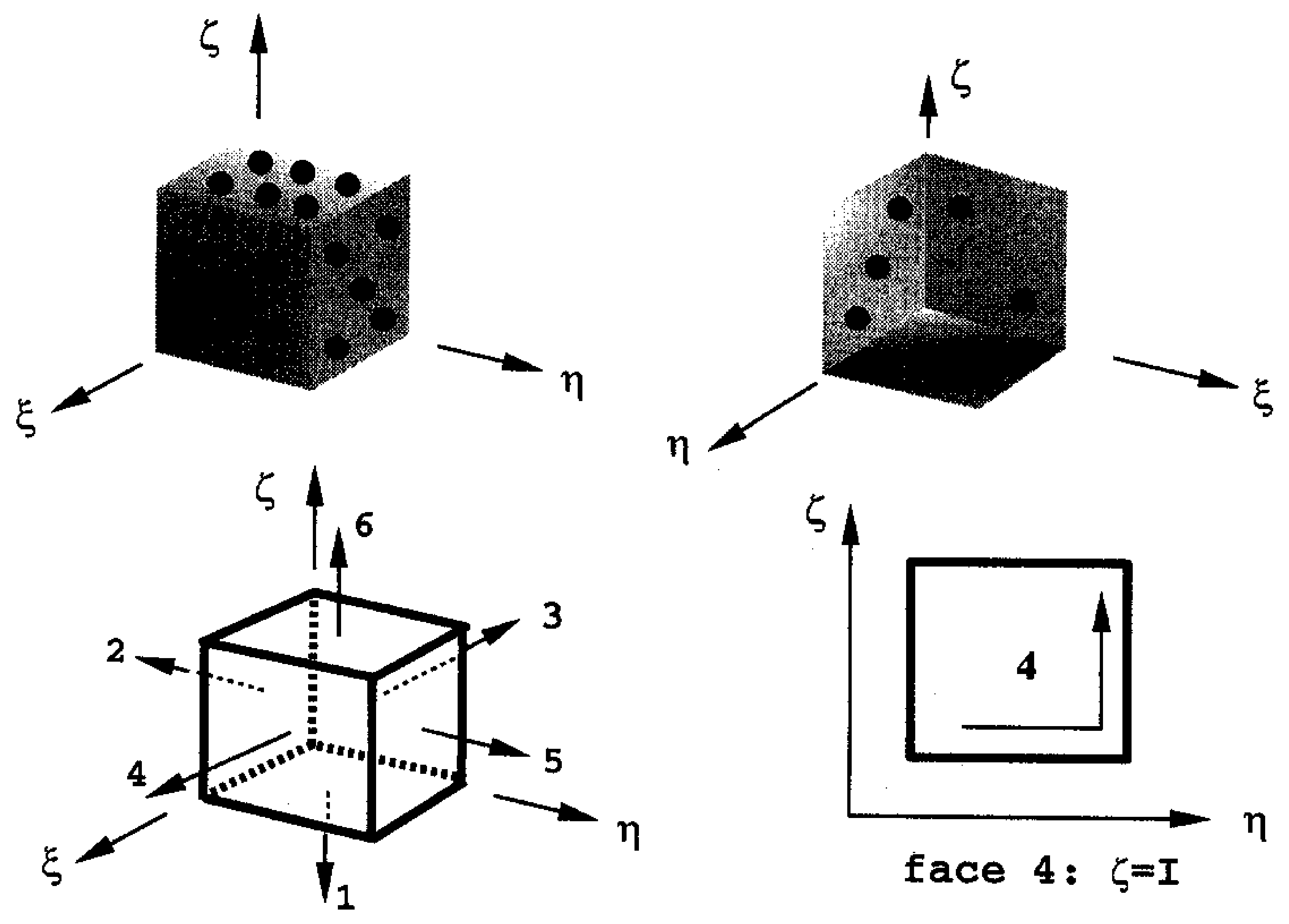

FIGURE 12.6 Orientation of faces. Coordinates ξ, η, ζ are numbered 1, 2, 3 where coordinates with lower numbers are stored first.

describing it as a $J(\eta)$ plane with a j value 1, i.e., by the pair $(J, 1)$ where the first value is the direction of the normal vector and the second value is the plane index. Thus, face 4 is defined by the pair (I, J). This notation is also required in the visualization module.

Grid points are stored in such a way that the I direction is treated first, followed by the J and K directions, respectively. This implies that K planes are stored in sequence. In the following the matching of blocks is outlined. First, it is shown how the orientation of the face of a block is determined. Second, rules are given how to describe the matching of faces between neighboring blocks. This means the determination of the proper orientation values between the two neighboring faces.

To determine the orientation of a face, arrows are drawn in the direction of increasing coordinate values. The rule is that the lower-valued coordinate varies first, and thereby the orientation is uniquely determined. The orientation of faces between neighboring blocks is determined as follows, see Figure 12.7. Suppose blocks 1 and 2 are oriented as shown. Each block has its own coordinate system (right-handed). For example, orientation of block 2 is obtained by a rotation of π of block 1 about the ζ-axis — rotations

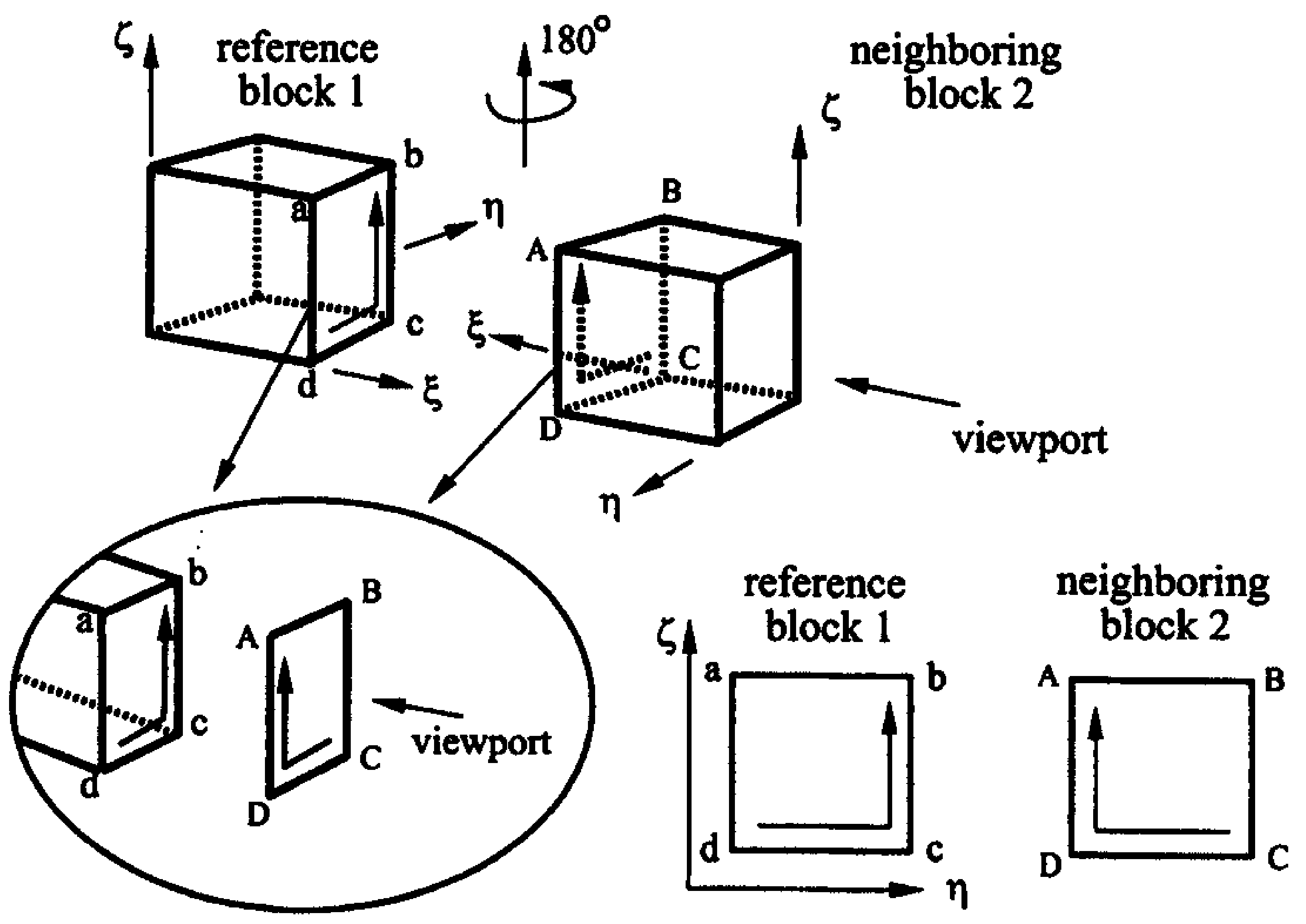

FIGURE 12.7 Determination of orientation of faces between neighboring blocks as seen from block 1 (reference block). The face of the reference block is always oriented as shown and then the corresponding orientation of the neighboring face is determined (see Figure 12.9).

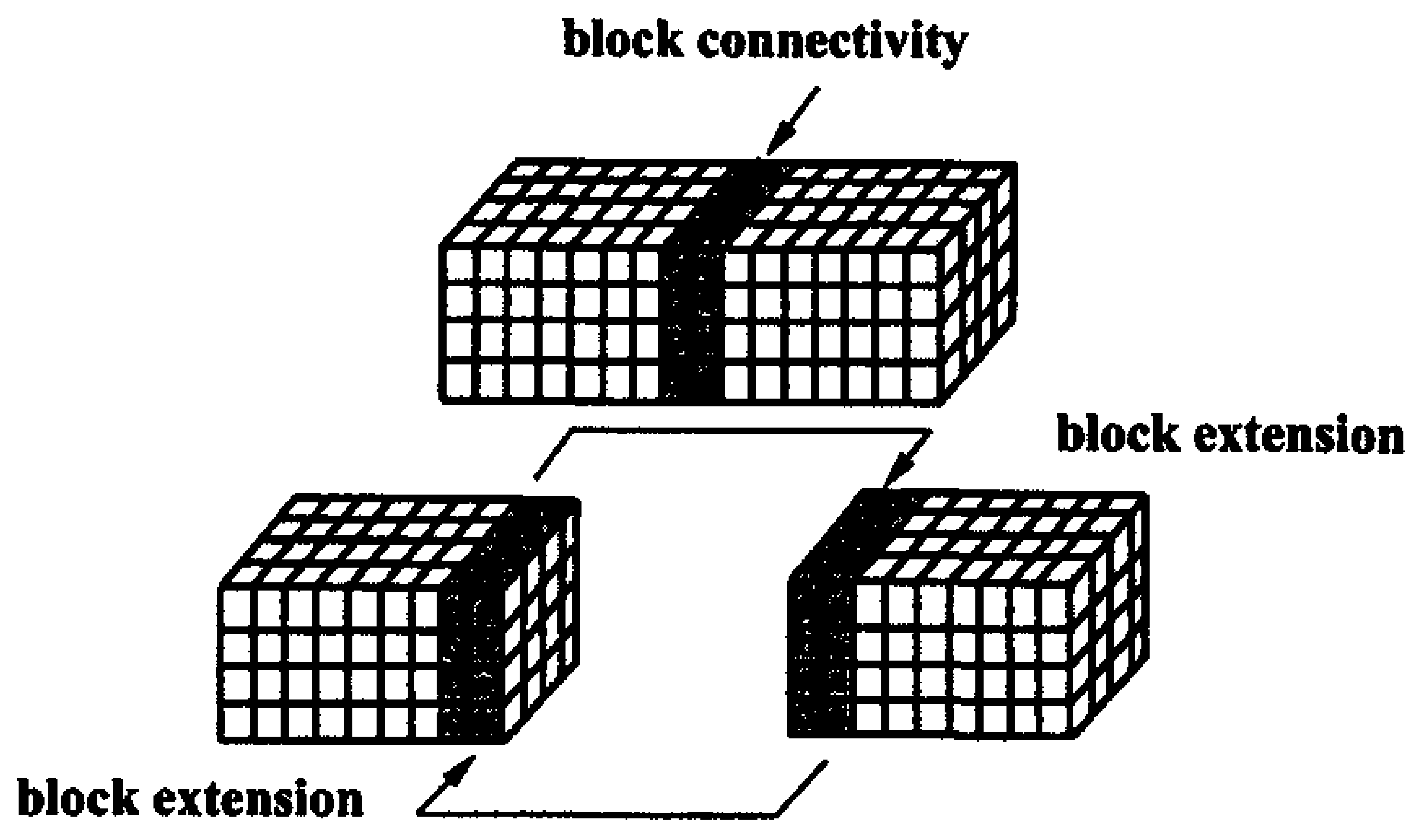

FIGURE 12.8 The figure shows the overlap of two neighboring blocks. For the flow solver, an overlap of two rows or columns is needed. The algorithm is not straightforward, because of the handling of diagonal points.

are positive in a counterclockwise sense. Thus face 4 of block 1 (used as the reference block) and face 4 of block 2 are matching with the orientations as shown, determined from the rules shown in Figure 12.9. All cases in group 1 can be obtained by rotating a face about an angle of 0, $1/2\pi$, π, or $3/2\pi$. This is also valid for elements in group 2. The code automatically recognizes when the orientation between two faces needs to be mirrored. Thus cases 1 and 7 in Figure 12.9 are obtained by rotating case 1 by $\pi/2$. Here, the rotations are denoted by integers 0, 1, 2, and 3, respectively.

12.5 Topology File Format for Multiblock Grids

To illustrate the topology description for the multiblock concept, a simple six-block grid for a diamond shape is shown in Figure 12.10. Only control file information for this grid is presented. The meaning of the control information is explained below. Since the example is 2D, the first line of this file starts with **\cntrl2d**. In 3D, this line has the form **\cntrl3d**. The corresponding coordinate values have been omitted, since they only describe the outer boundaries in pointwise form.

After the control line, object specific information is expected. The object specification is valid until the next control line is encountered or if the end of the current input file is reached. Control lines that cannot be identified are converted to the internal object type error.

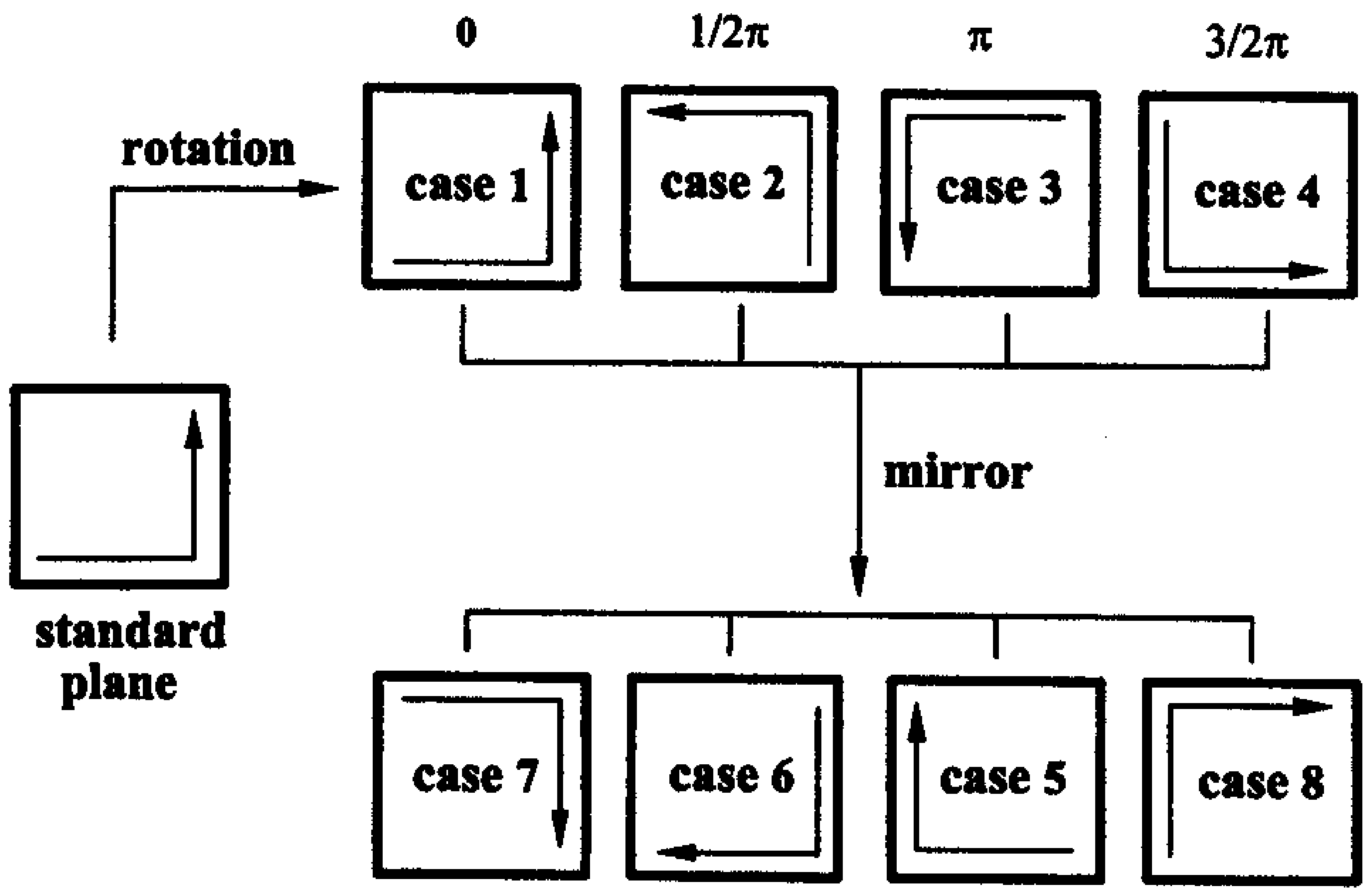

FIGURE 12.9 The 8 possible orientations of neighboring faces are shown. Cases 1 to 4 are obtained by successive rotations e.g., 0, $\frac{1}{2}\pi$, π, and $\frac{3}{2}\pi$. The same situation holds for cases 5 to 8 upon being mirrored.

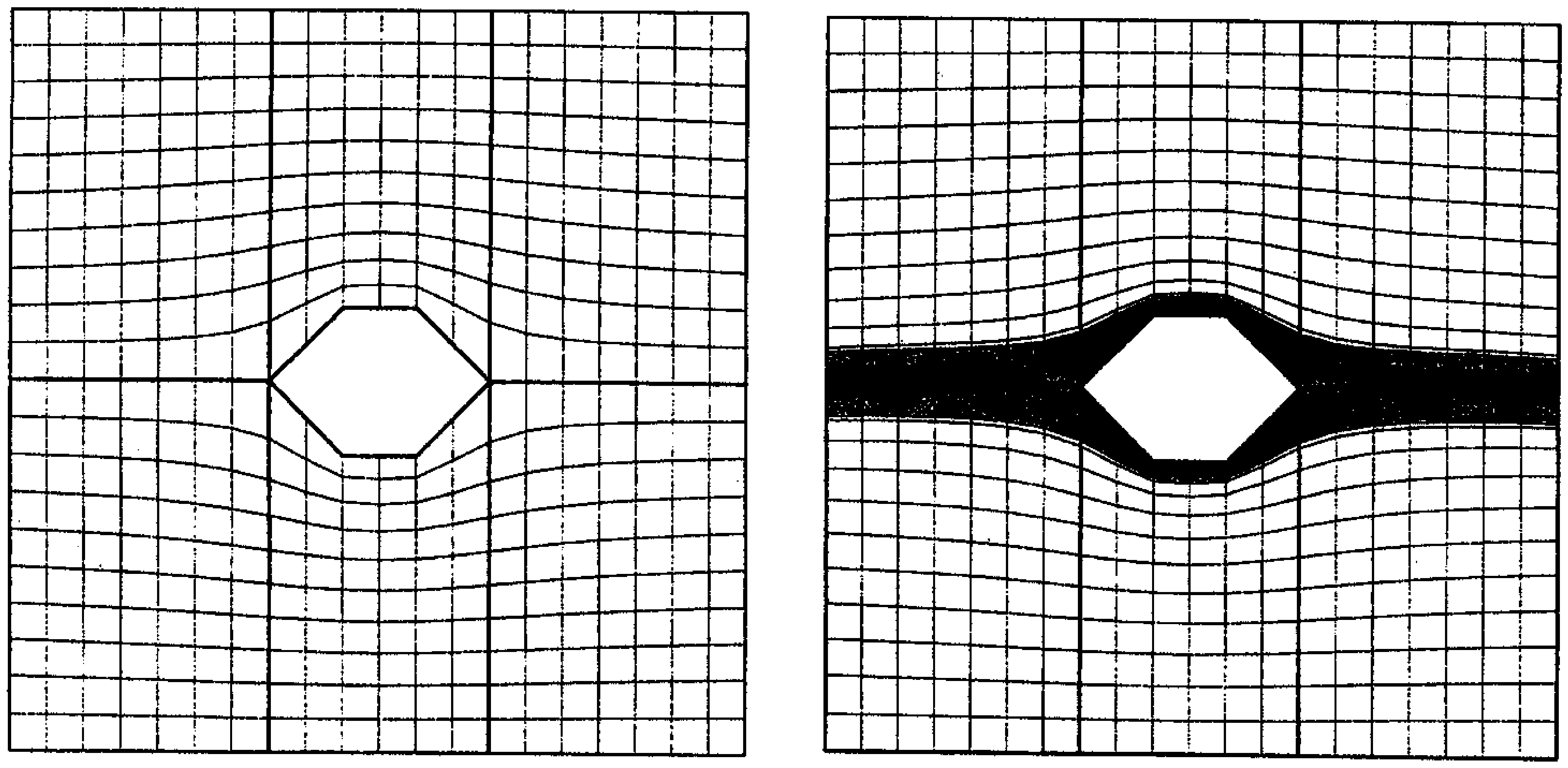

FIGURE 12.10 A six-block grid for a diamond-shaped body. This type of grid line configuration cannot be obtained by a monoblock grid. Grid lines can be clustered to match the flow physics, e.g., resolving a boundary layer. The topology information of this grid is shown in Table 12.1.

The control information in 2D has the form

```
\cntrl12d
nos
I J
s1 st nb ns
s2 st nb ns
s3 st nb ns
s4 st nb ns
```

where *nos* is the block number, and *I*, *J* are the number of grid points in the respective directions. The next four lines describe the four edges (or sides) of a block. *s*1 to *s*4 denote the side number where 1 is east, 2 north, 3 west, and 4 south. *st* is the side-type. 0 means fixed side, 1 is a fixed side used to compute the initial algebraic grid. A side of type 2 is a matching side (overlap). In this case, the corresponding values for *nb* and *ns* have to be given where *nb* is the number of the neighboring block and *ns* the number of the matching side of this block. If *st* is 0 or 1, these values should be set to zero. The edge control information can be in any order. The only restriction is that the same order must be used when boundary data are read. A similar format is used for the control information in 3D:

```
\cntrl13d
nos
I J K
s1 st nb ns nr
s2 st nb ns nr
s3 st nb ns nr
s4 st nb ns nr
s5 st nb ns nr
s6 st nb ns nr
```

Again, *nos* denotes the block number, and *I*, *J*, and *K* are the dimensions in *x*, *y*, and *z*-direction, respectively. Each block has six faces, so for each face there is one line with face-specific information. *s*1 to *s*6 are the face numbers as used for the standard block, see Figure 12.5, *st* is the face type, where a 1 denotes a face used for initialization (interpolated initial grid). In addition, to specify the neighboring block *nb* and the neighboring face *ns*, the rotation value *nr* is necessary.

- *s*1..*s*6: [1,6] $\rightarrow$ face number
- *st:* [0,3] $\rightarrow$ face type
- *nb:* [1,*N*] (*N* is total number of blocks) $\rightarrow$ neighboring block number
- *ns:* [1,6] $\rightarrow$ neighboring face of block *nb*
- *nr:* [0,3] $\rightarrow$ rotation needed to orient current face to neighboring face

Once the coordinates of a grid have been computed, the topology file as described above is constructed automatically from the grid points. While in the six-block example the command file could be set up by the user, the grid for the Cassini–Huygens Space Probe (see Figure 12.2), with its detailed microaerodynamics description, required a fully automatic algorithm. It would be too cumbersome for the user to find out the orientation of the blocks. Moreover, the generic aircraft (see Figure 12.15 later in this chapter), comprises 2200 blocks. All these tools are provided to the engineer in the context of the PAW (Parallel Aerodynamics Workbench) environment that serves as a basis from the conversion of CAD data to the realtime visualization of computed flow data by automating as much as possible the intermediate stages or grid generation and parallel flow computation.

12.6 Local Grid Clustering Using Clamp Technique

In the following we briefly describe a technology to obtain a high local resolution without extending this resolution into the far field where it is not needed. We thus substantially reduce the total number of grid points. This local clustering, however, changes block topology and leads to blocks of widely different size [Häuser et al., 1996]. Thus, it has a direct impact on the parallelization strategy, because size and number of blocks cannot be controlled. Therefore the parallelization strategy has to be adjusted to cope with this kind of sophisticated topology.

It is well known that along fixed walls a large number of grid lines is required in order to capture the boundary layer. In the remaining solution domain this requirement would cause a waste of grid points and reduce the convergence speed of flow solver. It is therefore mandatory to localize the grid line distribution. To this end, the so-called clamp clip technique was developed. Its principle is to build a closed block system connected to the physical boundary. The number of grid lines can be controlled within the block. Using clamp clips, grid lines are closed in clamp blocks. The local grid refinement can be achieved without influencing the far field grid. This is demonstrated in the lower part of Figure 12.11 and in Figure 12.1.

TABLE 12.1 Control Information for the Six-Block Diamond Grid

```
\cntrl2d
 1                    4
 8  11                7  11
 1   2   3   3        1   2   6   3
 2   3   2   4        2   0   0   0
 3   1   0   0        3   2   2   1
 4   1   0   0        4   1   0   0
 2                    5
 8  11                8  11
 1   3   4   3        1   0   0   0
 2   0   0   0        2   2   6   4
 3   1   0   0        3   3   3   1
 4   2   1   2        4   1   0   0
 3                    6
 7  11                8  11
 1   2   5   3        1   0   0   0
 2   0   0   0        2   0   0   0
 3   3   1   1        3   3   4   1
 4   1   0   0        4   3   5   2
                       \file diamond.lin
```

Note: This command file is also used by the parallel flow solver. File **diamond.lin** contains the actual coordinates values.

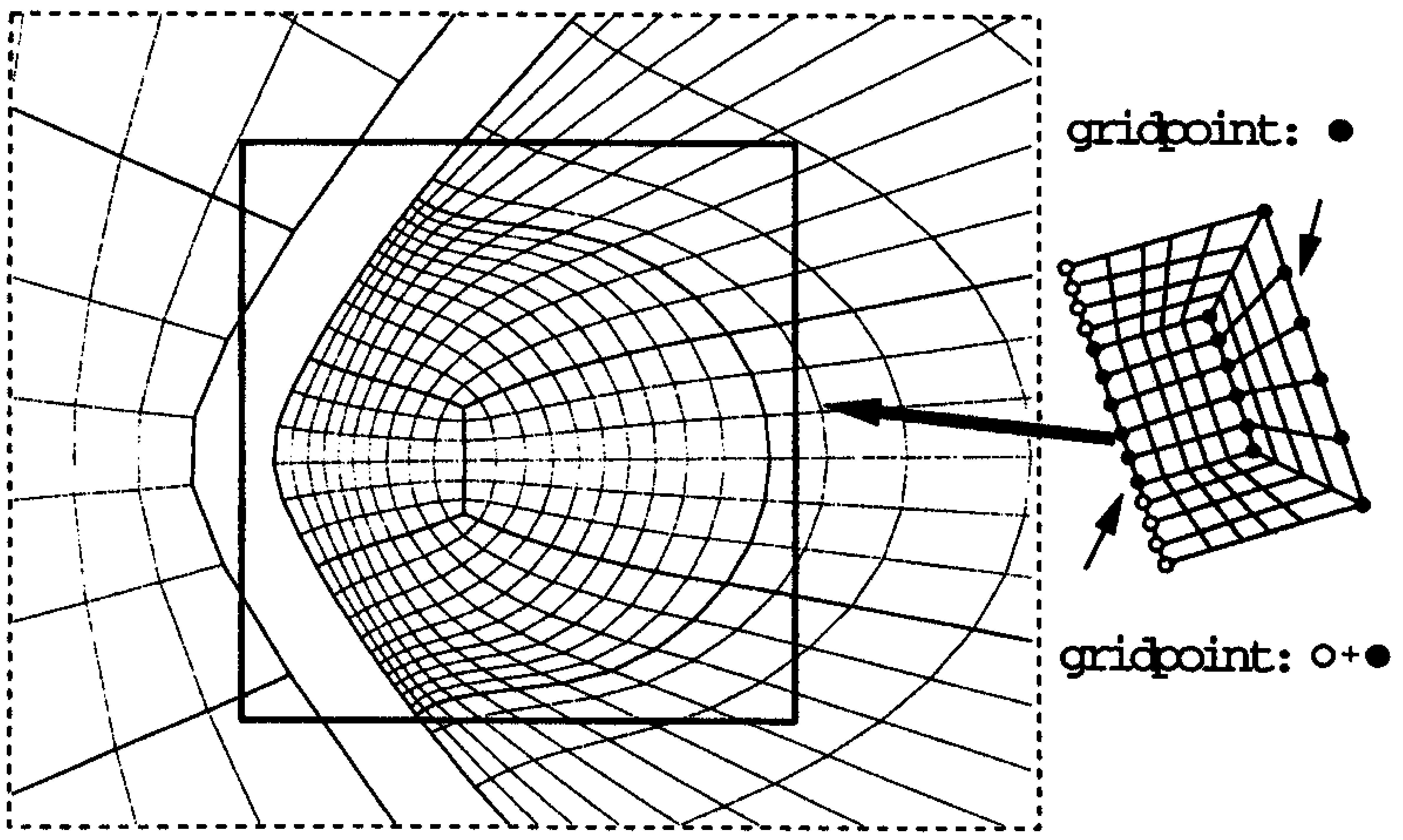

FIGURE 12.11 Clamp technique to localize grid line distribution. This figure shows the principle of a clamp. The real power of this technique is demonstrated in the Space Shuttle grid (see Figure 12.1).

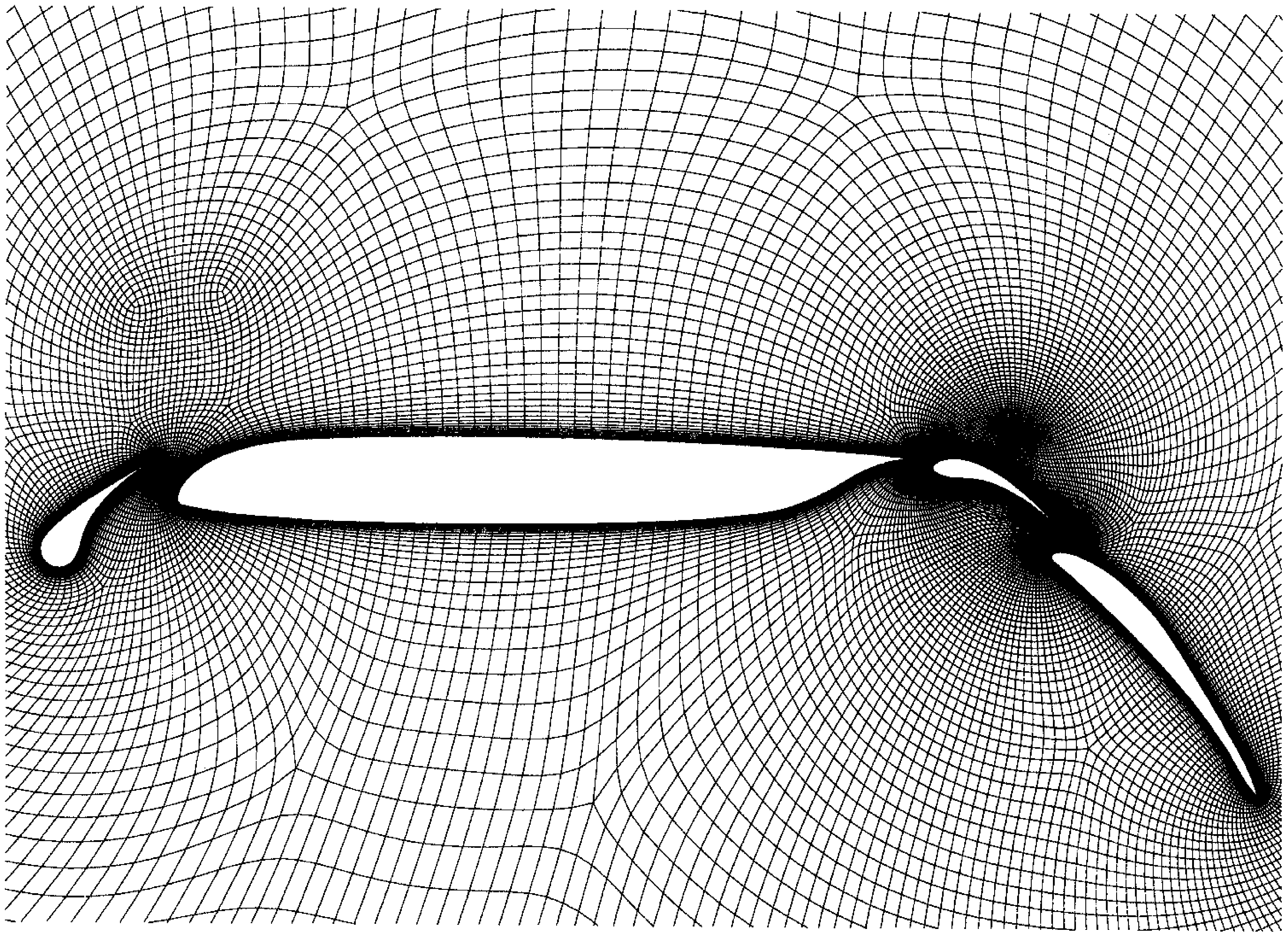

FIGURE 12.12 Navier–Stokes grid for a four-element airfoil, comprising 79 blocks. The first layer of grid points off the airfoil contour is spaced on the order of 10^{-6} based on chord length. Only the Euler grid is generated by the grid generator, the enrichment in the boundary layer is generated within a few seconds by the clustering module.

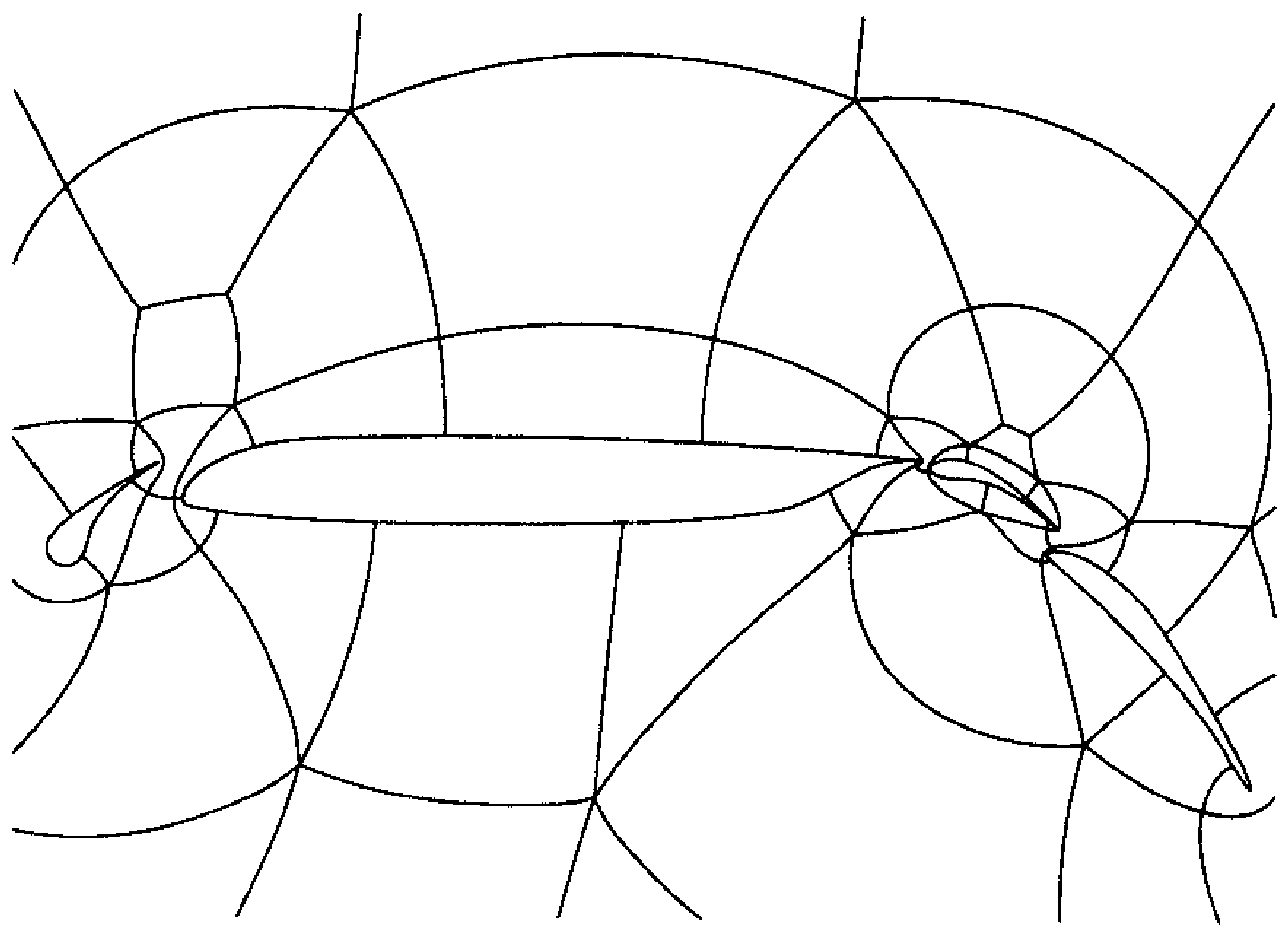

FIGURE 12.13 The figure shows the block structure of the four-element airfoil.

12.7 A Grid Generation Meta Language

12.7.1 Topology Input Language

With the parallel computers of today substantially more complex fluid flow problems can be tackled. The simulation of complete aircraft configurations, complex turbine geometries, or flows including combustion is now being computed in industry. Consequently, geometries of high complexity are now of interest as well as very large meshes, for instance, computations of up to 30 million grid points have

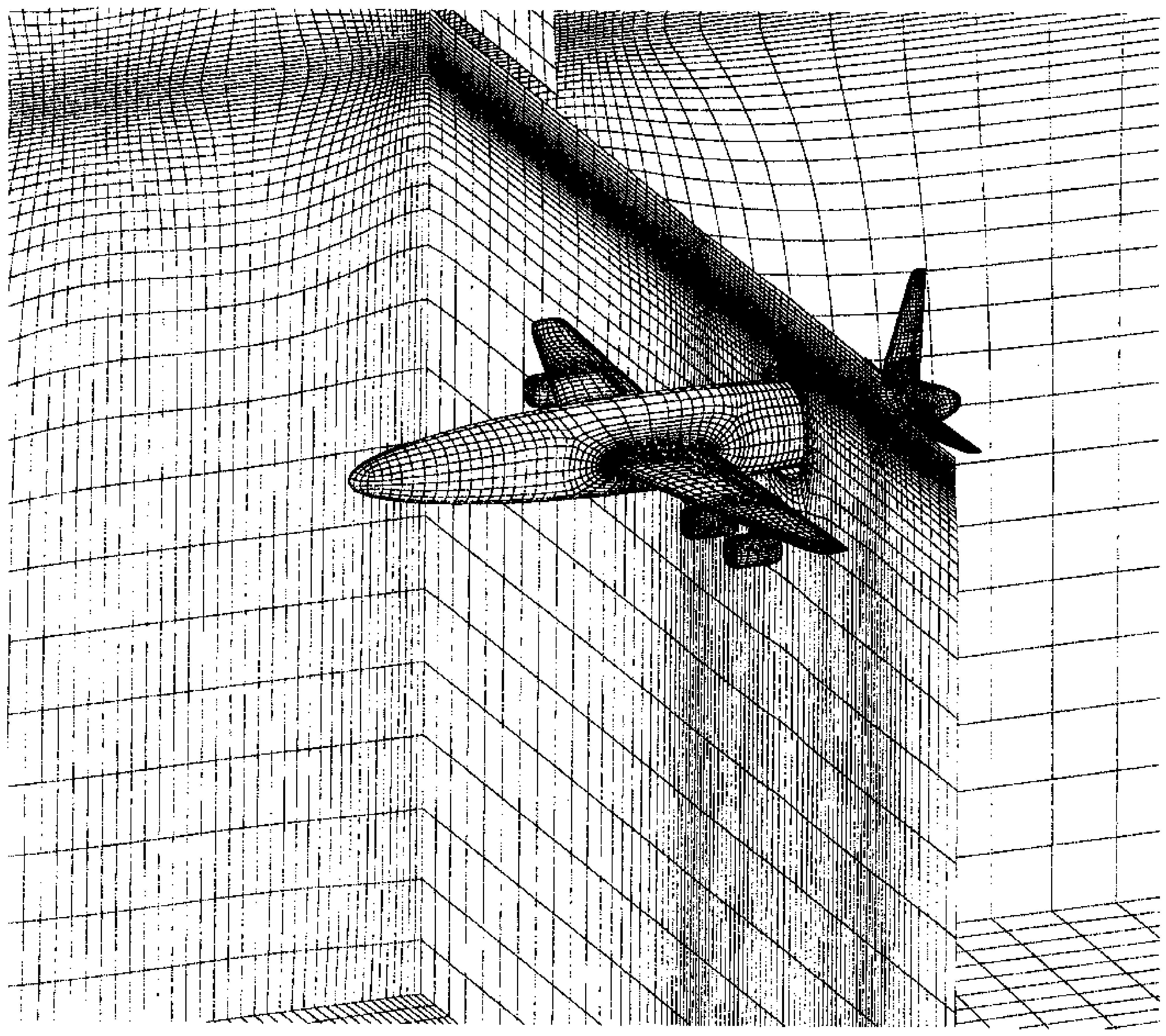

FIGURE 12.14 This picture shows the grid of the generic aircraft including flaps in a wind tunnel; however, the topology is exactly the same as for a real aircraft.

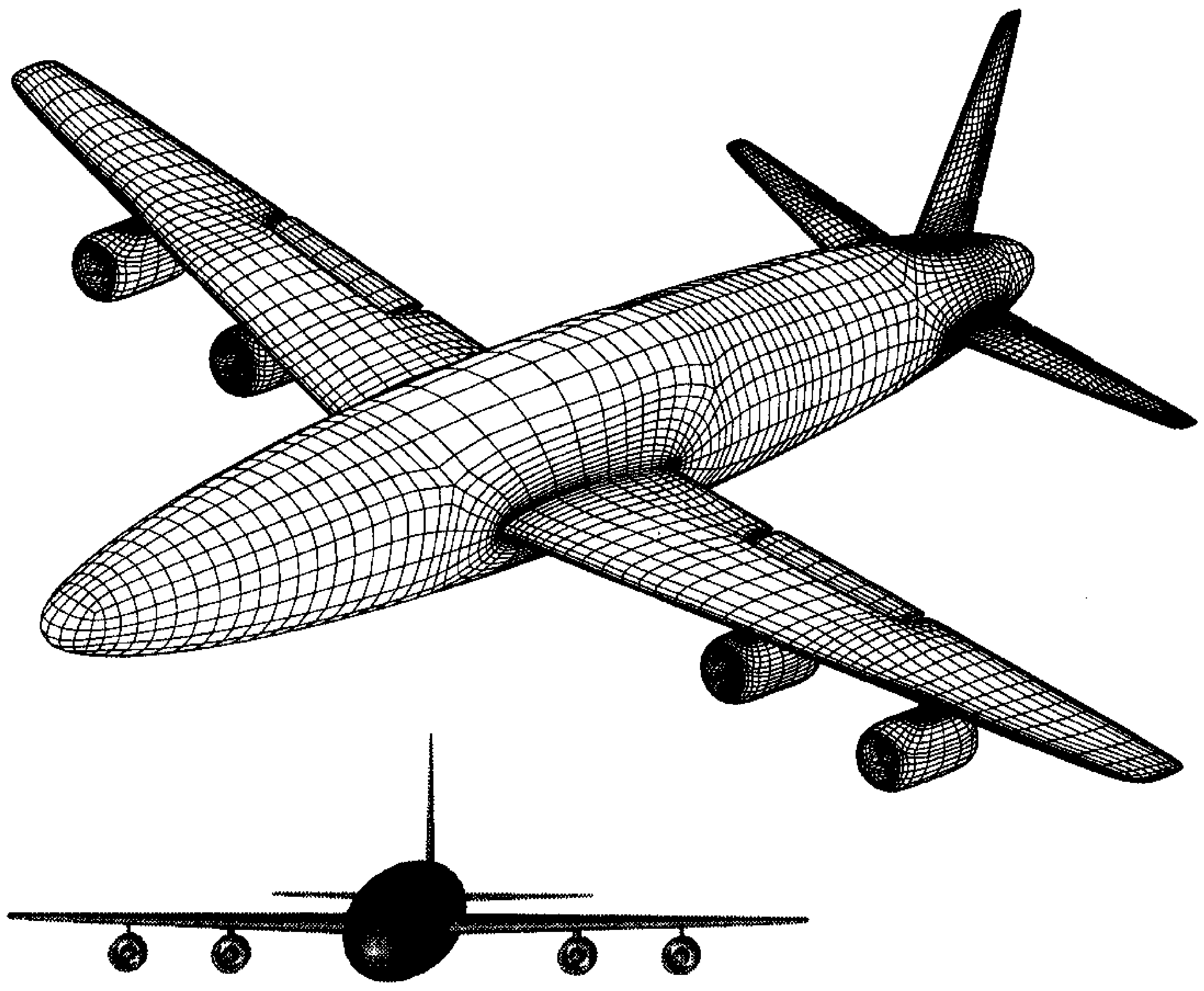

FIGURE 12.15 By modifying only a few lines of the **TIL** code, a four-engine generic aircraft is generated. The original two-engine grid was used as a starting grid.

been performed. Clearly, both grid generation codes and flow solvers have to be capable of handling this new class of application.

Conventional grid generation techniques derived from CAD systems that interactively work on the CAD data to generate the surface grid and then the volume grid are not useful for these large and complex meshes. The user has to perform tens of thousands of mouse clicks with no or little reusability of his/her input. Moreover, a separation of topology and geometry is not possible. An aircraft, for example, has a certain topology, but different geometry data may be used to describe different aircraft. Topology definition consumes a certain amount of work, since it strongly influences the resulting grid line configuration. Once the topology has been described, it can be reused for a whole class of applications. One step further would be the definition of objects that can be translated, rotated, and multiplied. These features could be used to build an application-specific data base that can be employed by the design engineer to quickly generate the grid needed.

In the following a methodology that comes close to this ideal situation is briefly described. A complete description can be found in [Eiseman, et al., 1996]. To this end, a completely different grid generation approach will be presented. A compiler-type grid generation language has been built, based on the ANSI-C syntax that allows the construction of objects. The user provides a (small) input file that describes the so-called **TIL** (Topology Input Language) code to build the wireframe model, see below, and specifies filenames used for geometry description of the configuration to be gridded. There is also the possibility to interactively construct this topology file, using the so-called AZ-Manager [Eiseman et al., 1996] package that works as a "topology generation engine."

For the description of the surface of a vehicle, a variety of surface definitions can be used. The surface can be described as a set of patches (quadrilaterals) or can be given in triangular form. These surface definitions are the interface to the grid generator. In general, a preprocessor is used that accepts surface definitions following the *NASA IGES CFD* [NASA, 1994] standard and converts all surfaces into triangular surfaces. That is, internally only triangular surfaces are used. In addition, the code allows the definition of analytic surfaces that are built in or can be described by the user
in a C function type syntax. The user does not have to input any surface grids, that is, surface and volume grids are generated fully automatically. This approach has the major advantage that it is reusable, portable, modular, and based on the object-oriented approach. Highly complex grids can be built in a step-by-step fashion from the bottom up, generating a hierarchy of increasingly complex grid objects.

For example, the grid around an engine could be an object (also referred to as component). Since an aircraft or spacecraft generally has more than one engine located at different positions beneath its wing, the basic engine object would have to be duplicated and positioned accordingly. In addition, the language is hierarchical, allowing the construction of objects composed of other objects where, in turn, these objects may be composed of more basic objects, etc. In this way, a library can be built for different technical areas, e.g., a turbomachinery library, an aircraft library, or a library for automotive vehicles. The TIL has been devised with these features in mind. It denotes a major deviation from the current interactive blocking approach and offers substantial advantages in handling both the complexity of the grids that can be generated and the human effort needed to obtain a high quality grid. No claims are made that TIL is the only (or the best) implementation of the concepts discussed, but it is believed that it is a major step toward a new level of performance in grid generation, in particular when used for parallel computing.

The versatility and relative ease of use — the effort is comparable with mastering LaTex, but the user need not write TIL code, because a TIL program can be generated by the interactive tool AZ-Manager, a procedure similar to the generation of applets using a Java applet builder — will be demonstrated by presenting TIL code for the six-block Cassini–Huygens probe. All examples presented in this chapter demonstrate both the versatility of the approach and the high quality of the grids generated.

In the following we present the TIL code to generate a 3D grid for the Cassini-Huygens space probe. Cassini-Huygens is a joint NASA-ESA project launched in 1997. After a flight time of seven years, the planet Saturn will be reached, and the Huygens probe will separate from
the Cassini orbiter and fly on to Titan, Saturn's largest moon. Titan is the only moon in the solar system possessing an atmosphere (mainly nitrogen). During the two-hour descent, measurements of the composition of the atmosphere will be performed by several sensors located at the windward side of the space probe. In order to ensure that laser sensors will function properly, no dust particles must be convected to any of the lens surfaces. Therefore, extensive numerical simulations have been performed investigating this problem.

TABLE 12.2 TIL Code for Six-Block Huygens Space Probe

```
SET GRIDDEN 47

COMPONENT Huygens()

BEGIN

s 1 -tube "HuygensProfil.lin";

s 2 -ellip(0.00015625 0.00015625 0.00015625) -o;

INPUT 1 (650 0 0 0 650 0 0 0 650)*face(sIN (1),cIN (-4),(-4),cOUT (1..4));

INPUT 2 (0 0 -4525)*(4525 0 0 0 4525 0 0 0 4525)*face(sIN (2),cIN (1:1..4),(-4),cOUT

(1..4));

INPUT 3 (0 0 325)*(650 0 0 0 650 0 0 0 650)*face(sIN (1),cIN (1:1..4),(-4),cOUT (1..4));

INPUT 4 (0 0 4525)*(4525 0 0 0 4525 0 0 0 4525)*face(sIN (2),cIN (3:1..4),(2:1..4),cOUT

(1..4));

END

COMPONENT face(sIN s, cIN c[1..4], C[1..4])

BEGIN

c 1 -1 - 1 + 0 -s s -L c:1 C:1;

c 2  1 - 1 + 0 -s s -L c:2 C:2 1;

c 3  1 + 1 + 0 -s s -L c:3 C:3 2;

c 4 -1 + 1 + 0 -s s -L c:4 C:4 1 3;

END
```

Note: The topology of this grid is explained in Figure 12.16.

In order to compute the microaerodynamics caused by the sensors, the proper grid has to be generated. A sequence of grids of increasing geometrical complexity has been generated. The simplest version, comprising six blocks, does not contain the small sensors that are on the windward side of the probe. With increasing complexity the number of blocks increases as well. The final grid, modeling the sensors, comprises 462 blocks. However, it is important to note that each of the more complex grids was generated by modifying the TIL code of its predecessor.

The general approach for constructing the Cassini-Huygens grids of increasing complexity is to first produce an initial mesh for the plain space probe without any instruments. Thus the first topology is a grid that corresponds to a *box in a box*, shown in Figure 12.16. The refinement of the grid is achieved by adding other elements designed as different objects. This topology describes the spherical far field and the body. The final grid is depicted in Figure 12.2 and Figure 12.17. This grid has a box-in-box structure: the outer box illustrates the far field and the interior one is the Huygens body. It should be noted that AZ-Manager was employed to automatically produce the TIL code from graphical user input [Ref.: "AZ-Manager"].

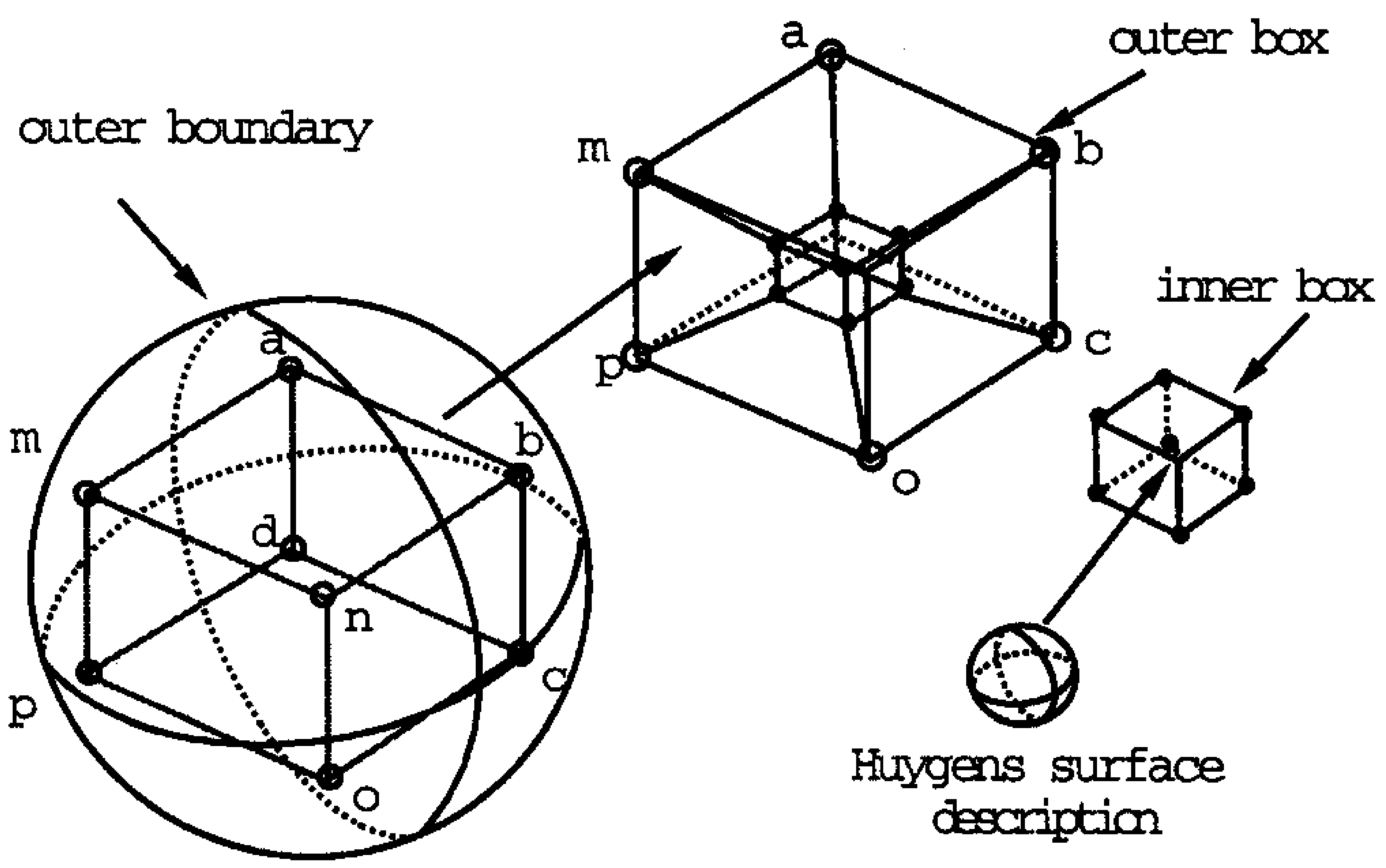

FIGURE 12.16 Topological design for the Huygens space probe grid. In this design all sensors are ignored. The topology is that of a 4D hypercube. The wireframe model consists of 16 vertices (corners). Vertices are placed interactively close to the surface (automatic projection onto the surface is performed) to which they are assigned. The grid comprises 6 blocks.

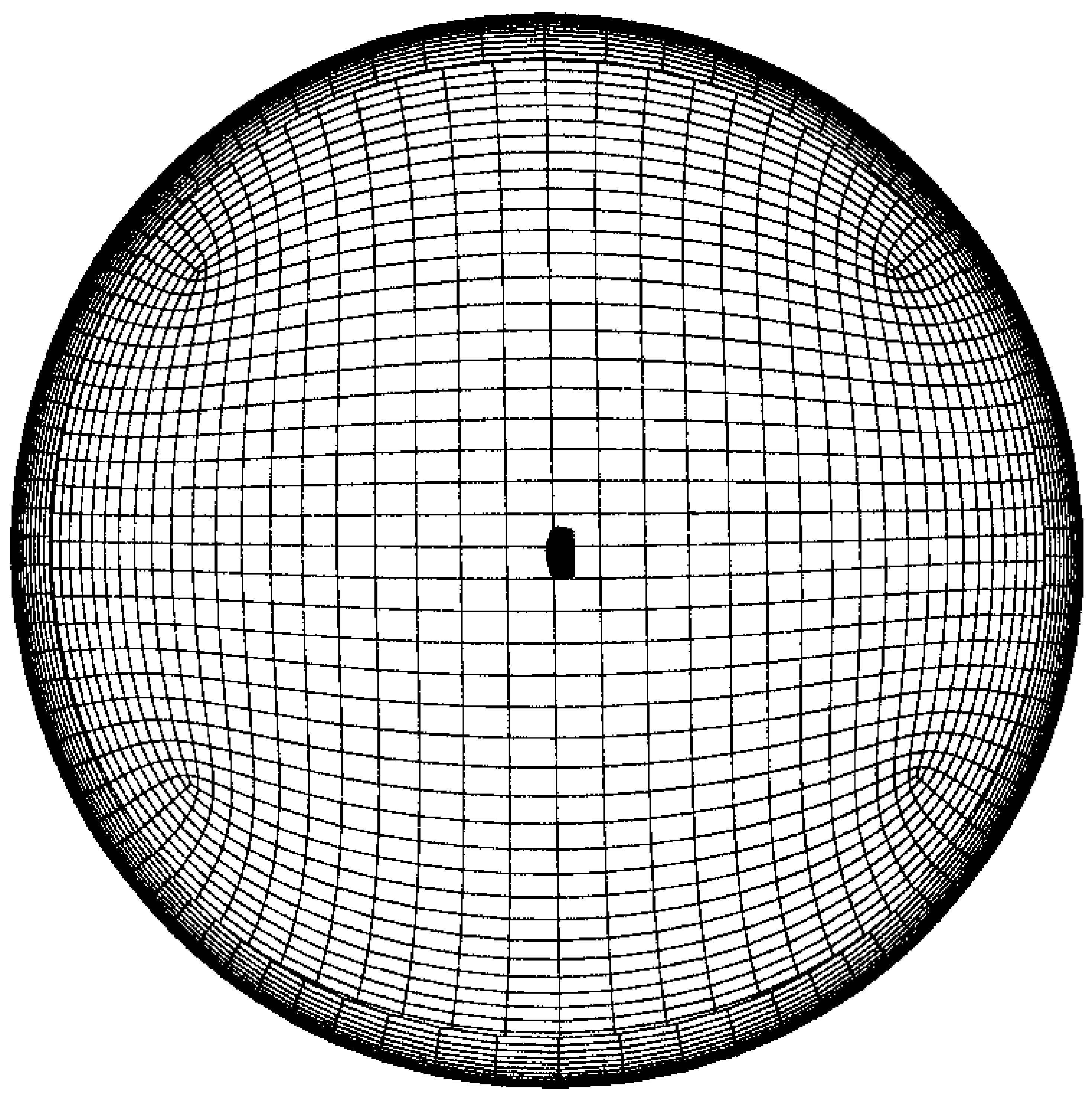

FIGURE 12.17 The 462-block grid for the Cassini–Huygens Space Probe launched in 1997 to fly to Saturn's moon Titan and to measure the composition of Titan's atmosphere after a flight time of seven years. This grid is bounded by a large spherical far field, in which the Huygens space probe is embedded. The ratio of the far field radius and the Huygens radius is about 20.

12.8 Parallelization Strategy for Complex Geometry

There are basically three ways of parallelizing a code. First, a simple and straightforward approach is to **parallelize the do-loops** in the code. Many so-called automatic parallelizers analyze do-loops and suggest a parallelization strategy based on this analysis. This concept, however, is not scalable to hundreds or thousands of processors, and results in a very limited speedup. A second approach therefore is to **parallelize the numerical solution process** for these equations. For example, if a matrix-vector multiplication occurs, this multiplication could be distributed on the various processors and performed in parallel. Again, scalability to a large number of processors cannot be obtained. Moreover, this technique would work only for large regular matrices. If a discretized problem were represented by a large number of smaller matrices (often the case in practice, e.g., multiblock grids), parallelization would be impossible.

The third approach adopts a simple idea and is denoted as **domain decomposition,** sometimes also refereed to as **grid partitioning.**

The solution domain is subdivided into a set of subdomains that exchange information to update each other during the solution process. The numerical solution takes place within each domain and is thus independent of the other domains. The solution space can be the actual space–time continuum or it can be an abstract space. For the computer simulation, this space is discretized and thus is described by a set of points. Domain decomposition is the most general and versatile approach. It also leads to the best parallel efficiency, since the number of points per subdomain (or block) can be freely varied as well as the number of subdomains per processor. A large number of codes in science and engineering use finite elements, finite differences, or finite volumes on either unstructured or structured grids. The process of parallelizing this kind of problem is to domain decompose the physical solution domain. Software [Williams et al., 1996] is available to efficiently perform this process both for unstructured and structured grids. Applying this strategy results in a fully portable code, and allows the user to switch over to new parallel hardware as soon as it becomes available.

There is, however, an important aspect in parallelization, namely the geometrical complexity of the solution domain. In the following, a brief discussion on geometrical complexity and how it affects parallelization is given. If the solution domain comprises a large rectangle or box, domain decomposition is relatively straightforward. For instance, the rectangle can be decomposed into a set of parallel stripes, and a box can be partitioned into a set of planes. This leads to a one-dimensional communication scheme where messages are sent to left and right neighbors only.

However, more realistic simulations in science and engineering demand a completely different behavior. For example, the calculation past an entire aircraft configuration leads to a partitioning of the solution domain that results in a large number of subdomains of widely different size, i.e., the number of grid points of the various blocks differ considerably. As a consequence, it is unrealistic to assume that a solution domain can be partitioned into a number of equally sized subdomains. In addition, it is also unrealistic to assume a nearest-neighbor communication. On the contrary, the set of subdomains is unordered (unstructured) on the subdomain level, leading to random communication among subdomains. In other words, the communication distance cannot be limited to nearest neighbors, but any distance on the processor topology is possible (processor topology describes how the processors are connected, for instance in a 2D mesh, in a torus or in a hypercube etc.). Hence, the efficiency of the parallel algorithm must not depend on nearest-neighbor communication. Therefore, the parallelization of solution domains of complex geometry requires a more complex communication pattern to ensure a load-balanced application. It also demands more sophisticated message passing among neighboring blocks, which may reside on the same, on a neighboring, or on a distant processor. The basic parallelization concept for this kind of problem is the introduction of a new type of boundary condition, namely the interdomain boundary condition that is updated in the solution process by neighboring subdomains via message passing. Parallelization then is simply achieved by the **introduction of a new type of boundary condition.** Thus, parallelization of a large class of complex problems has been logically reduced to the well-known problem of specifying boundary conditions.

12.8.1 Message Passing for Multiblock Grids

The following message-passing strategy has been found useful in the implementation of a parallel multiblock code. The parallelization of I/O can be very different with respect to the programming models (SPMD, Single Program Multiple Data, host-node — not recommended) and I/O modes (host-only I/O, node local I/O, fast parallel I/O hardware, etc.) supported by the parallel machines. The differences can also be hidden in the interface library.

Portability: Encapsulation of message passing routines helps to reduce the effort of porting a parallel application to different message passing environments.

Source code: Encapsulation allows the use of one source code both for sequential and parallel machines.

Maintenance and further development: Encapsulation keeps message-passing routines local. Thus, software maintenance and further development will be facilitated.

Common message-passing subset: Portability can be highly increased by restricting oneself to use only operations included in the common subset for implementing the interface routines.

- Since each processor of the parallel machine takes one or more blocks, there may not be enough blocks to run the problem on parallel machines. There are tools to automatically split the blocks to allow the utilization of more processors.
- In general, blocks are of very different sizes, so that the blocks must be distributed to the processors to produce a good load balance. There are tools to solve this bin-packing problem by a simple algorithm that takes virtually no time.

An extremely simple message passing model is implemented, consisting of only *send* and *receive*. The simplicity of this model implies easy portability.

For an elementary Laplace solver on a square grid, each grid point takes the average of its four neighbors, requiring 5 flops, and communicates 1 floating-point number for each gridpoint on the boundary. For a more sophisticated elliptic solver, needing 75 flops per internal grid point, grid coordinates have to be exchanged across boundaries. Our flow solver, *ParNSS* [Williams et al., 1996], in contrast, does a great deal of calculation per grid point, while the amount of communication is still rather small.

Thus we may expect any implicit flow solver to be highly efficient in terms of communication. When the complexity of the physics increases, with turbulence models and chemistry, we expect the efficiency to get even better. This is why a flow solver is a viable parallel program even when running on a workstation cluster with slow communication (Ethernet).

12.8.2 Parallel Machines and Computational Fluid Dynamics

For the kinds of applications that we are considering, we have identified four major issues concerning parallelism, whether on workstation clusters or parallel machines.

Load balancing. As discussed above, the number of blocks in the grid must be equal to or larger than the number of processors. We wish to distribute the blocks to processors to produce an almost equal number of grid points per processor; this is equivalent to minimizing the maximum number of grid points per processor. We have used the following simple algorithm to achieve this.

The largest unassigned block is assigned to the processor with the smallest total number of grid points already assigned to it, then the next largest block, and so on until all blocks have been assigned.

Given the distribution of blocks to processors, there is a maximum achievable parallel efficiency, since the calculation proceeds at the pace of the slowest processor, i.e., the one with the maximum number of grid points to process. This peak efficiency is the ratio of the average to the maximum of the number of grid points per processor, which directly proceeds from the standard definition of parallel efficiency.

Convergence. For convergence acceleration a block-implicit solution scheme is used, so that with a monoblock grid, the solution process is completely implicit, and when blocks are small, distant points become decoupled. Increasing the number of processors means that the number of blocks must increase, which in turn may affect the convergence properties of the solver. It should be noted that any physical fluid has a finite information propagation speed, so that a fully implicit scheme is neither necessary nor desirable.

Performance. It is important to establish the maximum achievable performance of the code on the current generation of supercomputers. Results from the Intel Paragon machine and SGI Power Challenge are presented.

Scalability. Parallel processing is only useful for large problems. For a flow solver, we wish to determine how many processors may be effectively utilized for a given problem size, since we may not always run extremely large problems.

TABLE 12.3 Intel Paragon Computations for a 192-Block Halis Grid

Node	Wall Time	Corrected Time (per step)	Iterations	Speedup	Maximum Efficiency
128	606	14.46	35	3.647	0.835
64	631	27.59	21	1.932	0.918
32	648	53.28	12	1.000	0.986

Note: Distributing 192 blocks of different size onto 128 processors leads to a certain load imbalance, hence speedup is somewhat reduced.

12.9 Parallel Efficiency for Multiblock Codes

It is often stated the scientific programs have some percentage of serial computational work, s, that limit the speedup, S, of parallel machines to an asymptotic value of $1/s$, according to Amdahl's law where $s + p = 1$ (normalized) and n is the number of processors:

$$S = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} \tag{12.1}$$

This law is based on the question, given the computation time on the serial computer, how long does it take on the parallel system? However, the question can also be posed in another way: Let s', p' be the serial and parallel time spent on the parallel system, then $s' + p'n$ is the time spent on a uniprocessor system. This gives an alternative to Amdahl's law and results in the speedup which is more relevant in practice:

$$S = \frac{s'+p'n}{s'+p'} = n-(n-1)s' \tag{12.2}$$

It should be noted that domain decomposition does not demand the parallelization of the solution algorithm but is based on the partitioning of the solution domain; i.e., the same algorithm on different data is executed. In that respect, the serial s or s' can be set to 0 for domain decomposition and both formulas give the same result. The important factor is the ratio r_{CT} (see below), which is a measure for the communication overhead. In general, if the solution algorithm is parallelized, Amdahl's law gives a severe limitation of the speedup, since for $n \rightarrow \infty$, S equals $1/s$. If, for example, s is 2% and n is 1000, the highest possible speedup from Amdahl's law is 50. However, this law does not account for the fact that s and p are functions of n. As described below, the number of processors, the processor speed, and the memory are not independent variables which simply means, if we connect more and faster processors, a larger memory is needed, leading to a larger problem size and thus reducing the serial part. Therefore speedup increases. If s' equals 2% and $n = 1024$, the scaled sized law will give a speedup of 980, which actually has been achieved in practice. However, one has to keep in mind that s and s' are different variables. If s' denoted the serial part on a parallel processor in floating point operations, it is not correct to set $s = s'n$, since the solution algorithms on the uniprocessor and parallel system are different in general.

For practical applications the type of parallel systems should be selected by the problem that has to be solved. For example, for routine applications to compute the flow around a spacecraft on 10^7 grid points, needing around 10^{14} floating point operations, computation time should be some 15 minutes. Systems of 1000 processors can be handled, so each processor has to perform about 10^{11} computations, and therefore a power (sustained!) of 100 MFlops per processor is needed. Assuming that 200 words, 8 bytes/word, are needed per grid point, the total memory amounts to 16 GB: that means 16 MB of private memory for each processor, resulting in 22 grid points in each coordinate direction. The total amount of processing time per block consists of computation and communication time:

TABLE 12.4 Convergence Behavior for 2D NACA 0012 Airfoil Speedup as Function of Number of Blocks

Block	Grid Points Per Block	Iteration	Computing Time	Speedup
2	2400	253	52519	1.00
32	1560	305	33930	1.55
120	435	317	22577	2.326
256	213	333	19274	2.725
480	119	349	17752	2.958
1024	61	380	18012	2.916

Note: This table clearly demonstrates that a fully coupled implicit solution scheme is not optimal.

$$t_p = N^3 * 10000 * t_c + 6N^2 * 10 * 8 * t_T \tag{12.3}$$

where we assumed that 10,000 floating point operations per grid point are needed, and 10 variables of 8 byte length per boundary point have to be communicated. Variables t_c, t_T are the time per floating point operation and the transfer time per byte, respectively. For a crude estimate, we omit the set-up time for a message. Using a bus speed of 100 MB/s, we find for the ratio of computation time and communication time.

$$r_{CT} := \frac{N^3 * 10000 * 100}{6N^2 * 10 * 8 * 100} \approx 20N \tag{12.4}$$

That is, for $N = 22$, communication time per block is less than 0.25% of the computation time. In that respect, implicit schemes should be favored, because the amount of computation per time step is much larger than for an explicit one.

In order to achieve the high computational power per node a MIMD (multiple instruction multiple data) architecture should be chosen; that means that the system has a parallel architecture. It should be noted that the condition $r_{CT} >> 1$ is not sufficient. If the computation speed of the single processor is small, e.g., 0.1 MFlops, this will lead to a large speedup, which would be misleading because the high value for r_{CT} only results from low processor performance.

12.10 Parallel Solution Strategy: A Tangled Web

12.10.1 Parallel Numerical Strategy

In this section a brief overview of the parallel strategy for the solution of large systems of linear equations as may be obtained from the discretization of the Navier-Stokes equations or elliptic grid generation equations is presented. The "tangled web" of geometry, grid and flow solver is discussed. The solution strategy is multifaceted, with

- **space strategy:** halving the grid spacing, termed grid sequencing,
- **linear solver strategy:** domain decomposition conjugate gradient–GMRES,
- **time-stepping strategy:** explicit/implicit Newton schemes for the N–S equations.

First we distinguish space discretization from time discretization. In case of the elliptic equations, we only have to solve the linear system once. The Navier–Stokes equations require the solution of a system of linear equations at each time step. If we are interested in a steady-state solution, a Newton–Raphson scheme is used. In addition, there is a sequence of grids, each with 8 times as many points as the last, and we loop through these from coarsest to finest, interpolating the final solution on one grid as the initial solution on the next finer grid. At the same time coarsening is used to compress the eigenvalue spectrum.

On each grid, the spatial discretization produces a set of ordinary differential equations: $d\mathbf{U}/dt = f(\mathbf{U})$, and we assume the existence of a steady-state $\mathbf{U}^*$ such that $f(\mathbf{U}^*) = 0$. We approach $\mathbf{U}^*$ by a sequence of explicit or implicit steps, repeatedly transforming an initial state $\mathbf{U}^0$ to a final state $\mathbf{U}$.

12.10.2 Time Stepping Procedure

For the Navier–Stokes equations, the following time stepping approach is used.
The explicit step is the two-stage Runge–Kutta:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + f\left(\mathbf{U}^n + f'\left(\mathbf{U}^n\right)\Delta t/2\right)\Delta t \tag{12.5}$$

The implicit time step is a backward Euler:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + f\left(\mathbf{U}^{n+1}\right)\Delta t \tag{12.6}$$

Third, we have the final step, getting to the steady-state directly via Newton, which can also be thought of as an implicit step with infinite Δt:

$$\text{solve} \quad \mathbf{f}(\mathbf{U}) = 0 \tag{12.7}$$

There is also a weaker version of the implicit step, which we might call the linearized implicit step, that is actually just the first Newton iteration of the fully nonlinear implicit step:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \left[1 - df/d\mathbf{U}\Delta t\right]^{-1} f\left(\mathbf{U}^n\right)\Delta t \tag{12.8}$$

The most time-consuming part in the solution process is the inversion of the matrix of the linear system of equations. Especially for fluid flow problems, we believe conjugate gradient (CG) techniques to be more robust than multigrid techniques, and therefore the resulting linear system is solved by the CG–GMRES method.

12.10.3 Parallel Solution Strategy

We use the inexpensive explicit step, as long as there is sufficient change in the solution, $\Delta\mathbf{U}$. When $\|\Delta\mathbf{U}\|/\|\mathbf{U}\|$ is too small, we begin to use the implicit step. Also, with each block the implicit solution scheme, the so-called dynamic GMRES, might exhibit a different behavior, that is a *Krylov* basis of different size may be used, eventually requiring dynamic load balancing.

12.10.4 Solving Systems of Linear Equations: The CG Technique

The conjugated gradient (CG) technique is a powerful method for systems of linear equations and therefore is used in many solvers. Its derivatives for nonpositive and nonsymmetric matrices (as obtained from discretizing the governing equations on irregular domains), for instance, GMRES (see Section 12.10.5), has a direct impact on the parallel efficiency of a computation. Its Krylov space dimension and hence the numerical load per grid point, varies during the computation depending on the physics. For instance, very high grid aspect ratios, a shock moving through the solution domain, or the development of a shear layer may have dramatic effects on the computational load within a block. Therefore, this kind of algorithm requires dynamic load balancing to ensure a perfect load balanced application.

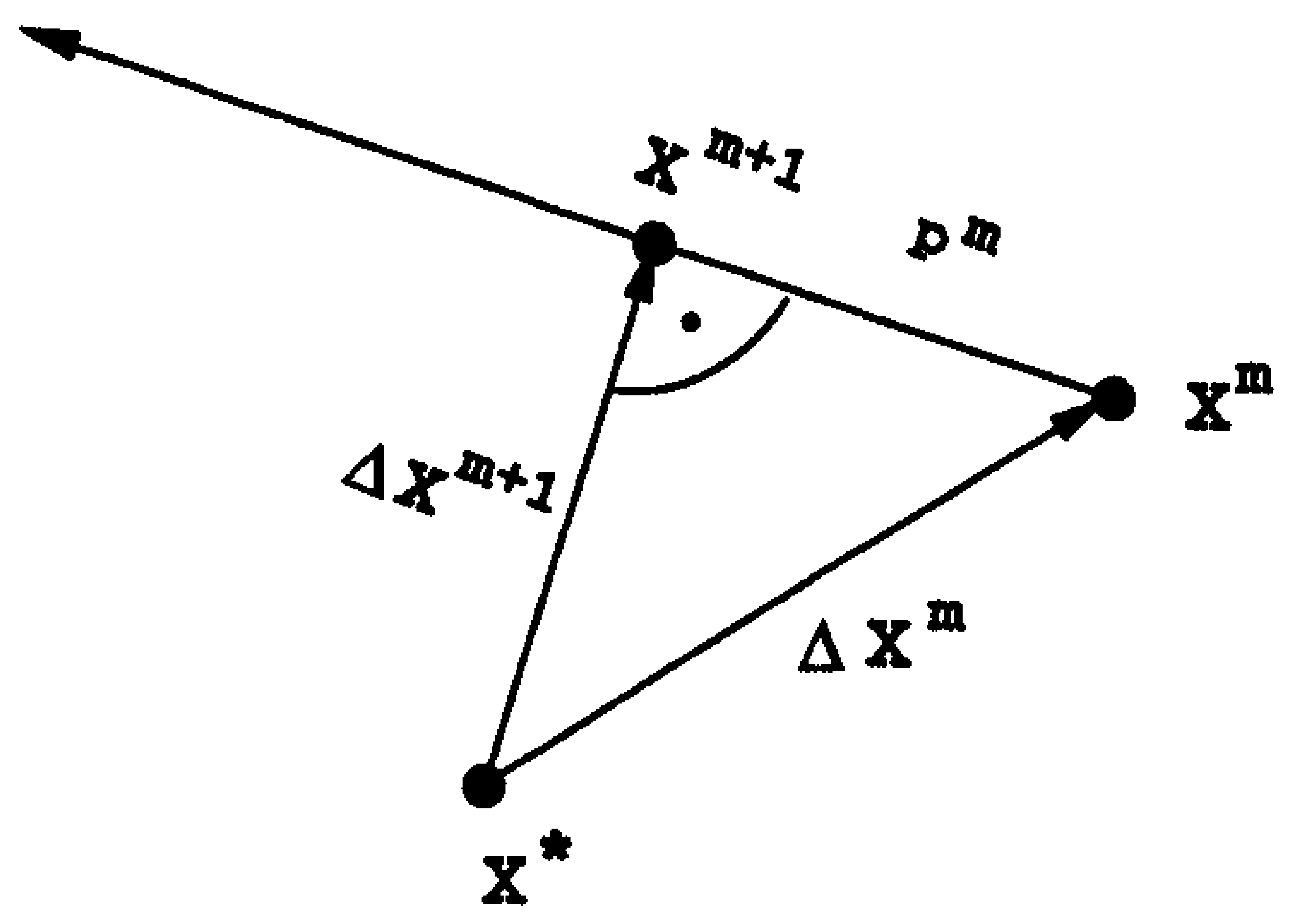

FIGURE 12.18 Geometrical interpretation of CG method. Let $\mathbf{x}^*$ denote the exact (unknown) solution, $\mathbf{x}^m$ an approximate solution, and $\Delta\mathbf{x}^m$ the distance from the exact solution. Given any search direction $\mathbf{p}^m$, except for $\mathbf{p}^m$ orthogonal to $\Delta\mathbf{x}^m$, it is straightforward to see that the minimal distance from $\mathbf{p}^m$ to $\mathbf{x}^*$ is found by constructing $\Delta\mathbf{x}^{m+1}$ perpendicular to $\mathbf{p}^m$.

In the following we give a brief description of the conjugate gradient method, explaining the geometric ideas on which the method is based. We assume that there is a system of linear equations derived from the grid generation equations or an implicit step of the N–S equations, together with an initial solution vector. This initial vector may be obtained by an explicit scheme, or simply may be the flow field from the previous step. It should be noted that the solution of this linear system is mathematically equivalent to minimizing a quadratic function. The linear system is written as

$$M \ \Delta U = R \Leftrightarrow \mathbf{Ax} = \mathbf{b} \tag{12.9}$$

using the initial solution vector x^0. The corresponding quadratic function is

$$f(x) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b} \tag{12.10}$$

where gradient $\nabla f = \mathbf{Ax} - \mathbf{b}$. For the solution of the Navier–Stokes equations, $\mathbf{x}^0$ is obtained from the most recent time steps, that is $\mathbf{x}^0 := \mathbf{U}^n - \mathbf{U}^{n-1}$ where index n denotes the number of explicit steps that have been performed. In the conjugate gradient method, a succession of one-dimensional search directions $\mathbf{p}^m$ is employed, i.e., the search is done along a straight line in the solution space — how these directions are constructed is of no concern at the moment — and a parameter α_m is computed such that function $f(\mathbf{x}^m - \alpha_m\mathbf{p}^m)$ along the $\mathbf{p}^m$ direction is minimized. Setting $\mathbf{x}^{m+1}$ equal to $\mathbf{x}^m - \alpha_m\mathbf{p}^m$, the new search direction is then to be found. In two dimensions, the contours $f =$ const. form a set of concentric ellipses, see Figure 12.19, whose common center is the minimum of f. The conjugate gradient method has the major advantage that only short recurrences are needed, that is, the new solution vector depends only on the previous one and the search direction. In other words, storage requirements are low. The number of iterations needed to achieve a prescribed accuracy is proportional to the square root of the condition number of the matrix, which is defined as the ratio of the largest to the smallest eigenvalue. Note that for second-order elliptic problems, the condition number increases by a factor of four when the grid-spacing is halved.

It is clear from Figure 12.18 that the norm of the error vector $\mathbf{x}^{m+1}$ is smallest being orthogonal to the search direction $\mathbf{p}^m$.

$$\left(\mathbf{x}^m - \mathbf{x}^n\right)\cdot\mathbf{p}^m = 0 \tag{12.11}$$

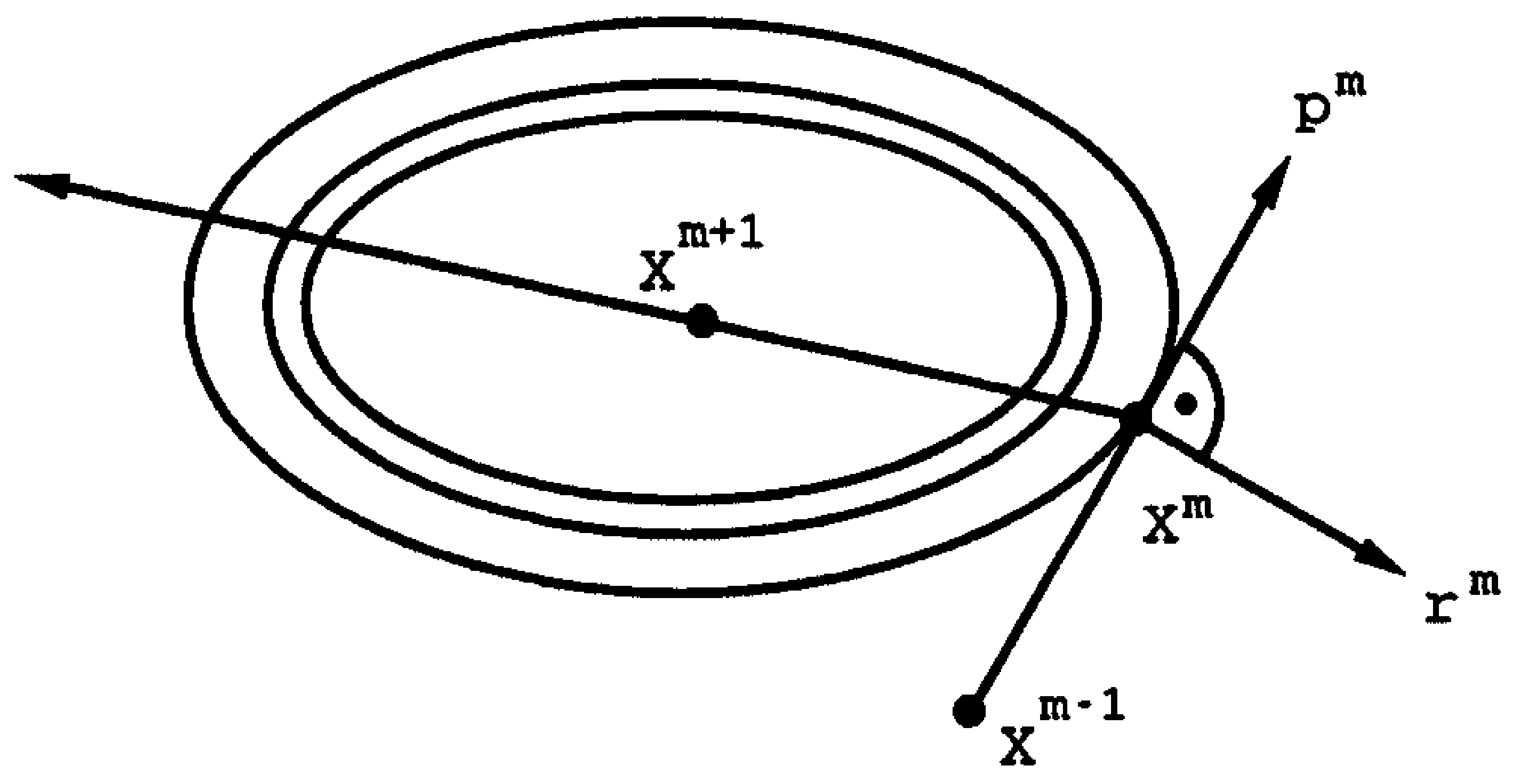

FIGURE 12.19 Geometrical interpretation of conjugate gradient method: since $\mathbf{r}^m$ is perpendicular to $\mathbf{p}^m$, a plane is spanned by these two vectors. The residual $\mathbf{r}^m$ is the gradient of the quadratic form $f(\mathbf{x})$ and thus perpendicular to the tangent of $f(\mathbf{x})$ = const. = $f(\mathbf{x}^m)$ at $\mathbf{x}^m$, because $\mathbf{x}^m$ is a minimal point. The next search direction pm+1 must therefore go through the midpoint of the ellipse, which is the projection of $f(\mathbf{x})$ onto this plane. The midpoint is the optimal point, i.e. gives the lowest residual in this plane. It is straightforward to show that $\mathbf{p}^{m+1}$ must satisfy $(\mathbf{p}^{m+1}, A\mathbf{p}^m) = 0$, simply because we are dealing with an ellipse. Moreover, $\mathbf{p}^{m+1}$ must be a linear combination of $\mathbf{r}^m$ and $\mathbf{p}^m$, and thus can be expressed as $\mathbf{p}^{m+1} = \mathbf{r}^m + \beta_k \mathbf{p}^m$.

From this **first orthogonality condition,** α_m can be directly computed. Figure 12.18 shows a right-angled triangle, and it directly follows (*Euclidean norm*) that the sequence of error vectors is **strictly monotonic decreasing.** In other words, *if the linear system derived from the Navier–Stokes equations,* $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, *has a unique solution, convergence is guaranteed, if N linear independent search vectors* $\mathbf{p}^m$ *are used.* This, however, is not of practical relevance, because in the solution of the Navier–Stokes equations there may be millions of variables, and only a few hundred or thousand iterations are acceptable to reach the steady state.

Since the exact change in the solution is not known, in practical computations the residual is used that is defined as

$$\mathbf{r}^m := \mathbf{b} - \mathbf{A}\mathbf{x}^m \tag{12.12}$$

Minimizing the quadratic function $f(\mathbf{x}^m - \alpha_m \mathbf{p}^m)$ along search direction $\mathbf{p}^m$ and using the expression for the residual directly gives

$$\alpha_m = \frac{(\mathbf{r}^m, \mathbf{p}^m)_A}{(\mathbf{p}^m, \mathbf{p}^m)_A} \tag{12.13}$$

In addition, it is required that $f(\mathbf{x}^m - \alpha_m \mathbf{p}^m)$ also be the minimum over the vector space spanned by all previous search directions $\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{m-1}$, because we must not destroy the minimal property when a new search direction is added. Hence the search directions are chosen to be A orthogonal, denoted as the **second orthogonality condition** defining the scalar product $(\mathbf{p}^k, \mathbf{p}^m)_A := (\mathbf{p}^k, A\mathbf{p}^m) = 0$ for $k \neq m$.

In determining the direction vectors, $\mathbf{p}^m$, a natural condition is that if a minimum in direction $\mathbf{p}^m$ is computed, the minimization already performed in the previous search directions, $\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{m-1}$ must not be affected. This is clearly the case if $\mathbf{p}^m$ is orthogonal to all previous basis vectors, because then $\mathbf{p}^m$ has no components in these directions and thus the minimum of f with respect over the subspace of $\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{m-1}$ is not changed by adding $\mathbf{p}^m$.

The original conjugate gradient method, however, has a requirement that matrix A by symmetric and positive definite (i.e., the quadratic form $\mathbf{x}^T \mathbf{A}\,\mathbf{x} > 0$). Clearly, matrix A of Eq. 12.9 does not possess these features. Therefore, an extension of the conjugate gradient method, termed *Dynamic GMRES* is employed that is described next.

12.10.5 Basic Description of GMRES

We have seen that the Navier–Stokes equations can be reduced to a system of linear equations, Eq. 12.9. Since a problem may comprise several million variables, an efficient method is needed to invert the matrix on the LHS. The system resulting from the Navier–Stokes equations is linear but neither positive definite nor symmetric, the term $(\mathbf{p}^m, A\mathbf{p}^m)$ is not guaranteed to be positive, and the search vectors are not mutually orthogonal. Therefore the conjugate gradient technique cannot be used directly. The extension of the conjugate gradient technique is termed the *generalized minimum residual* (GMRES) method [Saad, 1996]. It should be remembered that $\mathbf{p}^{m+1} = \mathbf{r}^m + \alpha^m \mathbf{p}^m$ and that the α^m are determined such that the second orthogonality condition holds, but this is no longer possible for the nonsymmetric case. However, this feature is mandatory to generate a basis of the solution space. Hence, this basis must be explicitly constructed. GMRES minimizes the norm of the residual in a subspace spanned by the set of vectors $\mathbf{r}^0, A\mathbf{r}^0, A^2\mathbf{r}^0, \ldots, A^{m-1}\mathbf{r}^0$, where vector $\mathbf{r}^0$ is the initial residual, and the mth approximation to the solution is chosen from this space. The above-mentioned subspace, a Krylov space, is made orthogonal by the well-known Gram–Schmidt procedure, known as an Arnoldi process when applied to a Krylov subspace. When a new vector is added to the space (multiplying by A), it is projected onto all other basis vectors and made orthogonal with the others. Normalizing it and storing its norm in entry $h_{m,m-1}$, a matrix H_m is formed with nonzero entries on and above the main diagonal as well as in the subdiagonal. Inserting the equation for x_m into the residual equation, and after performing some modifications, a linear system of equations for the unknown coefficients γ^l_m involving matrix H_m is obtained. H_m is called an upper Hessenberg matrix. To annihilate the subdiagonal elements, a 2D rotation (Givens rotation) is performed for each column of H_m until $h_{m,m-1} = 0$. A Givens rotation is a simple 2×2 rotation matrix. An upper triangular matrix R_m remains, which can be solved by back substitution.

It is important to note that the successful solution of the parallel flow equations can only be performed by a Triad numerical solution procedure. Numerical Triad is the concept of using **grid generation, domain decomposition,** and the **numerical solution scheme** itself. Each of the three Triad elements has its own unique contribution in the numerical solution process. However, in the past, these topics were considered mainly separately and their close interrelationship has not been fully recognized. In fact, it is not clear which of the three topics will have the major contribution to the accurate and efficient solution of the flow equations. While it is generally accepted that grid quality has an influence on the overall accuracy of the solution, the solution dynamic adaptation process leads to an intimate coupling of numerical scheme and adaptation process, i.e., the solution scheme is modified by this coupling as well as the grid generation process. When domain decomposition is used, it may produce a large number of independent blocks (or subdomains). Within each subdomain a block-implicit solution technique is used, leading to a decoupling of grid points. Each domain can be considered to be completely independent of its neighboring domains, **parallelism** simply being achieved by introducing **a new boundary condition,** denoted as inter-block or inter-domain boundary condition. Updating these boundary points is done by message passing. It should be noted that exactly the same approach is used when the code is run in serial mode, except that no messages have to be sent to other processors. Instead, the updating is performed by simply linking the receive buffer of a block to its corresponding neighboring send buffer. Hence, **parallelizing a multiblock code demands neither rewriting the code nor changing its structure.**

A major question arises in how the **decomposition process affects the convergence rate** of the implicit scheme. First, it should be noted that the N–S equations are not elliptic, unless the time derivative is omitted and inertia terms are neglected (Stokes equations). This only occurs in the boundary layer when a steady state has been reached or has almost been reached. However, in this case the Newton method will converge quadratically, since the initial solution is close to the
final solution. The update process via boundaries therefore should be sufficient. In all other cases, the N–S equations can be considered hyperbolic. Hence, a full coupling of all points in the solution domain would be unphysical, because of the finite propagation speed, and is therefore not desired and not needed. To retain second-order accuracy across block (domain) boundaries, an overlap of two points in each coordinate direction has to be implemented. This guarantees the numerical solution is independent of

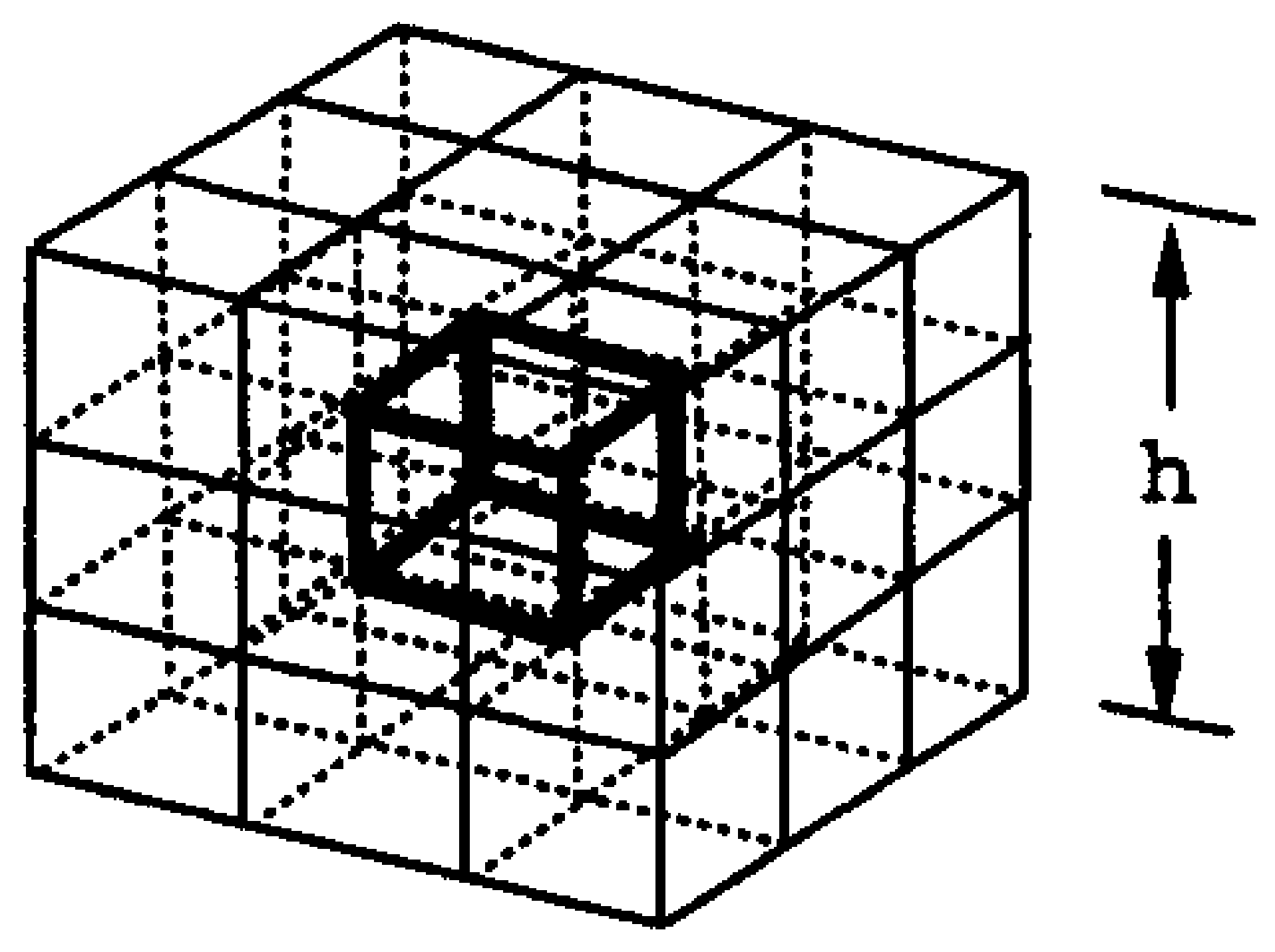

FIGURE 12.20 Flow variables are needed along the diagonals to compute mixed second derivatives for viscous terms. A total of 26 messages would be needed to update values along diagonals. This would lead to an unacceptable large number of messages. Instead, only block faces are updated (maximal six messages), and values along diagonals are approximated by a finite difference stencil.

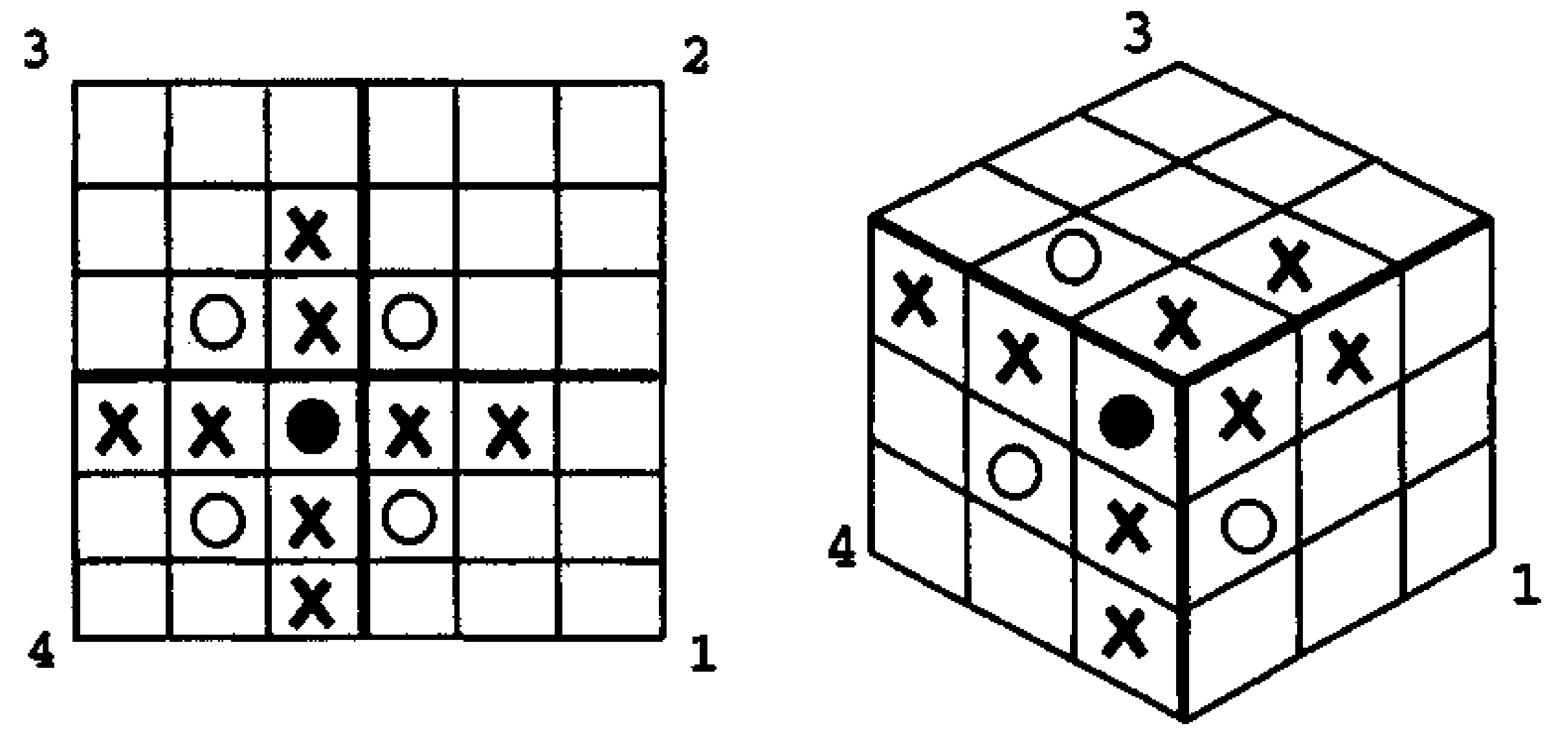

FIGURE 12.21 The figure shows the computational stencil. Points marked by a cross are used for inviscid flux computation. Diagonal points (circles) are needed to compute the mixed derivatives in the viscous fluxes. Care has to be taken when a face vanishes and 3 lines intersect.

block topology. The only restriction comes from the computation of flow variables along the diagonals on a face of a block (see Figure 12.20), needed to compute the mixed derivatives in viscous terms.

It would be uneconomical to send these diagonal values by message passing. Imagine a set of 27 cubes with edge length $h/3$ assembled into a large cube of edge length h. The small cube in the middle is surrounded by 26 blocks that share a face, an edge, or a point with it. Thus, 26 messages would have to be sent (instead of 6 for updating the faces) to fully update the boundaries of this block. Instead, the missing information is constructed by finite difference formulas that have the same order of truncation error, but may have larger error coefficients.

To continue the discussion of convergence speed it should be remembered that for steady-state computations implicit techniques converge faster than fully explicit schemes. The former are generally more computationally efficient, in particular for meshes with large variations in grid spacing. However, since a full coupling is not required by the physics, decomposing the solution domain should result in a convergence speed-up, since the inversion of a set of small matrices is faster than the inversion of the single large matrix, although boundary values are dynamically updated. On the other hand, if the decomposition leads to a block size of one point per block, the scheme is fully explicit and hence computationally less efficient than the fully implicit scheme. Therefore, an optimal decomposition topology must exist that most likely depends on the flow physics and the type of implicit solution process. So far, no theory has been developed.

Second, domain decomposition may have a direct influence on the convergence speed of the numerical scheme. In this chapter, the basis of the numerical solution technique is the Newton method, combined with a conjugate gradient technique for convergence acceleration within a Newton iteration. In the preconditioning process used for the conjugate gradient technique, domain decomposition may be used to decrease the condition number (ratio of largest to smallest eigenvalues) of the matrix forming the left-hand side, derived from the discretized N–S equations. In other words, the eigenvalue spectrum may be compressed, because the resulting matrices are smaller. Having smaller matrices the condition number should not increase; using physical reasoning it is concluded that in general the condition number should decrease.

From these remarks, it should be evident that only a combination of grid generation scheme, numerical solution procedure, and domain decomposition approach will result in an effective, general numerical solution strategy for the parallel N–S equations on complex geometries. Because of their mutual interaction these approaches must not be separated. Thus, the concept of numerical
solution procedure is much more general than devising a single numerical scheme for discretizing the N-S equations. Only the implementation of this interconnectedness in a parallel solver will lead to the optimal design tool.

12.11 Future Work in Parallel Grid Generation and CFD

Since neither vector nor parallel computing is of interest to the scientist or engineer who has to compute an application, a simple but general rule is that scalar architectures requiring the smallest number of processors to provide a certain computing power should be favored. As experience shows, it is the input and output that becomes cumbersome when a large massively parallel system is used. The paradigm of having each processor read its own file and write its own file starts to tax the file system greatly. This is because there is a single disk controller converting file names into disk track locations, and this constitutes a sequential bottleneck. It is better to have all the processors opening a single large file, and each reading and writing large records from that file whose size is a power of two number of bytes. For instance, this I/O approach has been implemented for the Intel version of *ParNSS* running on several hundred processors.

One of the most challenging tasks is the development of algorithms that scale numerically. The so-called **Tangled Web** approach, see Section 12.10, based on the idea of a varying coupling strength among grid points during the solution process, will be one of the most important novel techniques that might have the potential to achieve this objective.

Acknowledgment

We are grateful to our colleagues Jean Muylaert and Martin Spel from ESTEC, Noordwijk, The Netherlands for many stimulating discussions. This work was partly funded under EFRE Contract 95.016 of the European Union and the Ministry of Science and Culture of the State of Lower Saxony, Germany.

References

1. **Bruce, A.**, et al., JPL sets acoustic checks of cassini test model, *Aviation Week and Space Technology,* 143(9), pp. 60–62, 1995.
2. **Eiseman, P.**, et al., GridPro/AZ3000, User's guide and reference manual, *PDC,* 300 Hamilton Ave, Suite 409, White Plains, N, 10601, pp. 112, 1996.
3. **Häuser, J., et al.**, Euler and N–S grid generation for halis configuration with body flap," *Proceedings of the 5th International Conference on Numerical Grid Generation in Computational Field Simulation,* Mississippi State University, pp. 887–900, 1996.
4. NASA Reference Publication 1338, NASA geometry data exchange specification for CFD, (NASA IGES), Ames Research Center, 1994.
5. **Saad, Y.**, *Iterative Methods For Sparse Linear Systems,* PWS Publishing, 1996.
6. **Venkatakrishnan, V.**, Parallel implicit unstructured grid Euler solvers, *AIAA Journal,* Vol. 32, 10, 1994.
7. **Williams, R.**, Strategies for approaching parallel and numerical scalability in CFD codes, *Parallel CFD,* Elsevier, North-Holland 1996.