CRS4

Centre for Advanced Studies,

Research and Development in Sardinia

Uta - (CA)

Programma ENEA-MURST

Obiettivo 7

Implementazione del codice Ares su architetture parallele

by:

S. Chibbaro and M. Mulas

Computational Fluid Dynamics Area

Abstract

This report describes the MPI parallelization of the combustion code **Ares**. First, the structure of the code is described, then a detailed analysis of the algorithms used to integrate the flow equation is carried out, with particular emphasis to the parallelizaztion issues. An analysis of few possible alternative approaches is then followed by the description of the chosen approach. A preliminary validation of the parallel code completes the report. Thourough validation on 3D complex test cases will be presented in the next report.

Contents

1 Structure of the code

Ares code belongs to a family of codes developed in the past years by the group of Fluid Dynamics of CRS4 (see references [1] and [2] for instance). It can be divided in three main parts.

- *INIT*: all variables are initialized and data are allocated.
- *FSSOLVER*: residuals are computed and all equations are advanced in time with an implicit fractional step method.
- *OUTPUT*: all the output fields are written.

This report begins with the description of the global structure of the code. Then the multi-block structure, the boundary conditions codification method and the data allocation are illustrated.

1.1 The Multi-Block Structure

In cell-centered Finite-Volume codes the boundary condition treatment is done by means of a layer of ghost cells surrounding the computational domain. In case of multi-block meshes, time-advancing iterations should be performed on each block independently and, eventually, in parallel, as shown in figure 1:

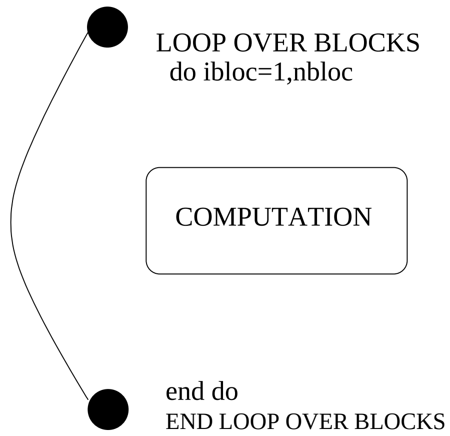

Figure 1: The block loop structure

Two layers of ghost cells are then used to store both the geometrical definition (volumes and surface normals) and field information (values of the dependent variables).

Figure 2 shows how the domain splitting process consists of constructing the appropriate ghost cells for each sub-block, which in turn means duplicating the information on both sides of the interface cut.

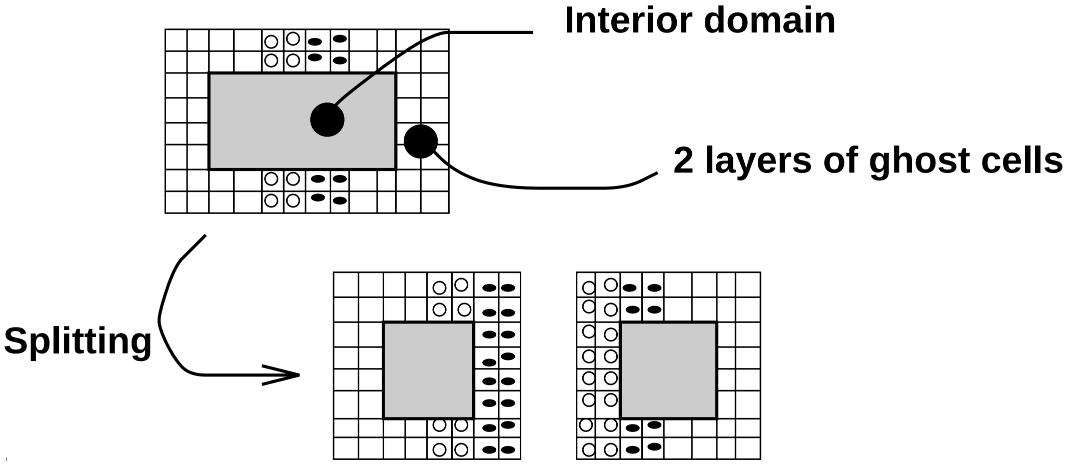

Figure 2: Domain splitting

If NI, NJ and NK represent the number of grid nodes of a structured mesh, then all of the field arrays within a given block are dimensioned in the following way:

```
ARRAY(-1:NI+1,-1:NJ+1,-1:NK+1)
```

no matter whether they represent cell-center values (such as volumes, dependent variables, time-steps, ...), node values (such as grid point coordinates), or values of physical and geometrical entities located at the cells' interfaces (such as for instance surface normals and fluxes). Figure 3 shows, for the sake of simplicity, a 2D grid made of $NI \times NJ$ nodes. Internal cells range from index $(1,1)$ (lower-left corner), to index $(NI-1, NJ-1)$ (upper-right corner), while node-centered entities arrays range from index $(1,1)$ (lower-left corner), to index (NI, NJ) (upper-right corner).

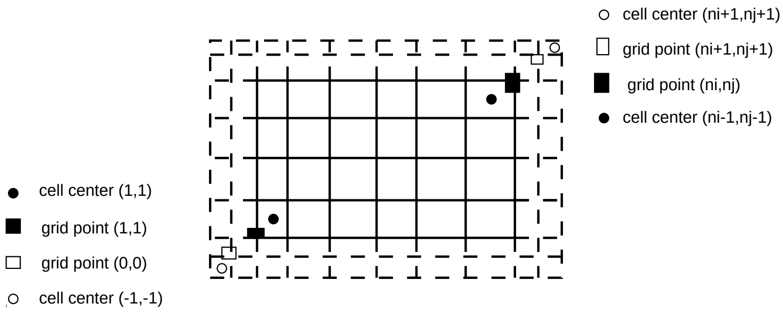

Figure 3: Grid nomenclature

Figure 4 shows how the different block face are identified for the application of boundary conditions. For generality and flexibility purposes, each face can be subdivided in various portions (segments), as shown in fig. 5, each segment corresponding to a different boundary treatment.

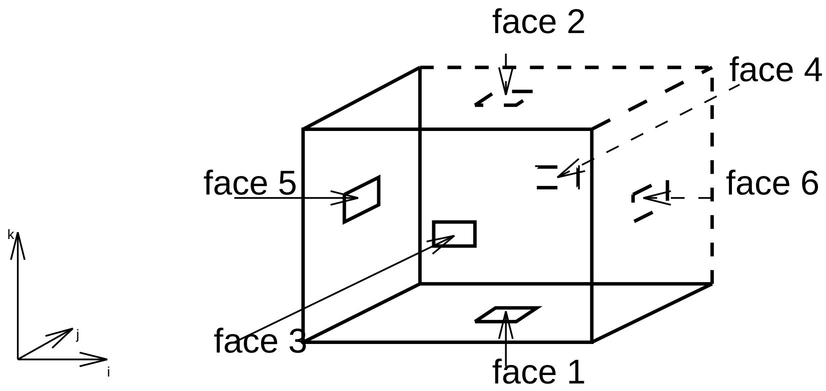

Figure 4: Face nomenclature

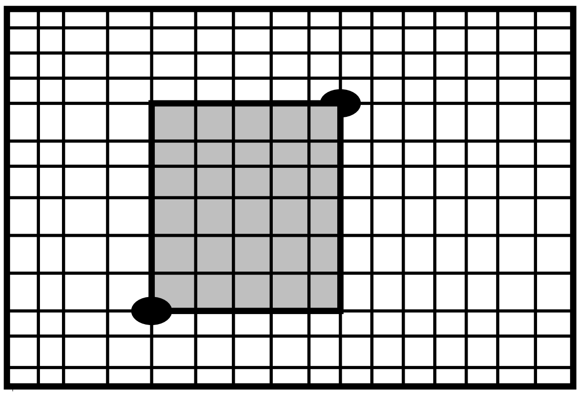

Figure 5: Face splitting

1.2 Data allocation

A one dimensional *double precision* array **work** is used to store all data: as shown in figure 6 the first part of **work** is reserved for permanent data, whereas a second part is used to dynamically store temporary data.

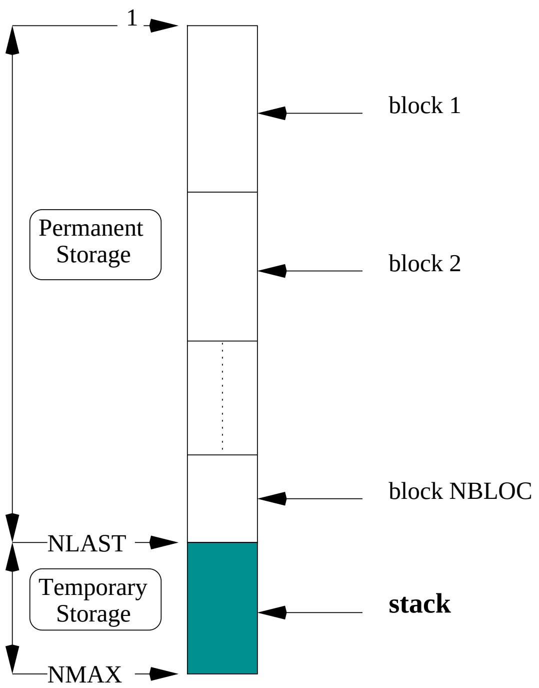

Figure 6: The `work` array structure

The **work** array is dimensioned in the main program as:

$$DOUBLE\ PRECISION\ WORK(NMAX)$$

where $NMAX$ is a parameter which can be modified by the user depending on the size of the test case. Figure 6 shows the array structure: memory is allocated for all permanent data of all blocks (the total number is $NBLOC$). $NLAST$ represents the last array location allocated for permanent data. The space left for temporary storage is then made by $(NMAX - NLAST)$ locations.

The temporary storage is managed as a stack with two utility routines which act as the **C**-functions *pop* and *push*.

Each block area, shown in figure 7, is organized in the same way. The data can be accessed by means of arrays of pointers, all of them dimensioned as

$$IPOINTER(NBLOC\_MAX)$$

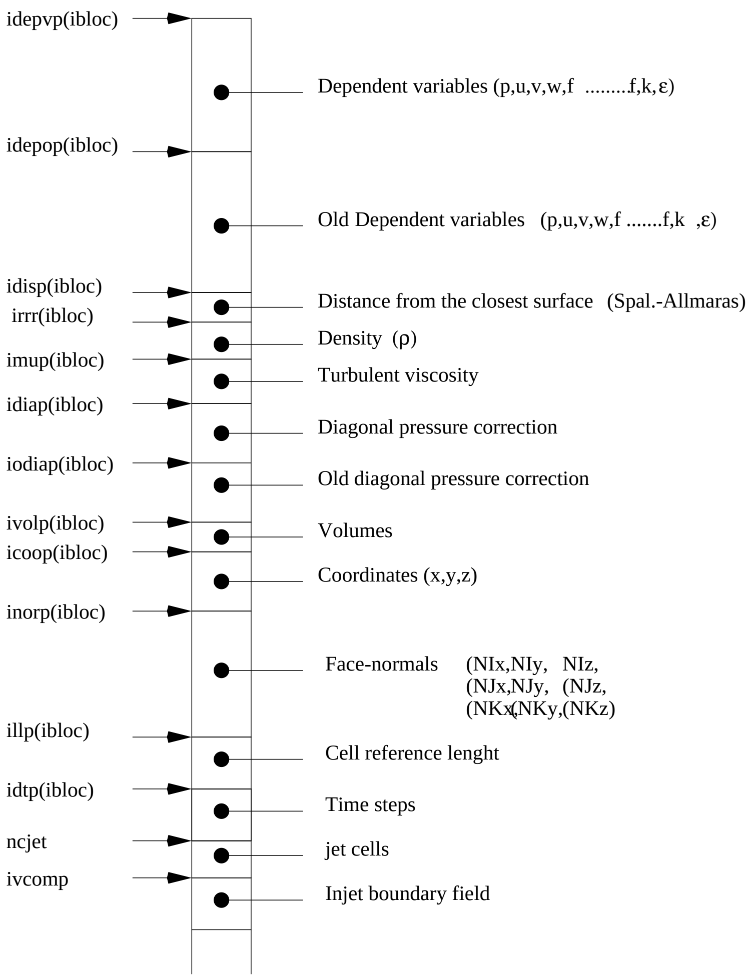

Figure 7: Details of the `work` array structure

where $NBLOC\_MAX$ is user defined parameter which identifies the maximum number of blocks allowed. All pointer arrays are stored in the COMMON block, together with the following ones:

$$NNI(NBLOC\_MAX)$$
$$NNJ(NBLOC\_MAX)$$
$$NNK(NBLOC\_MAX)$$
$$NCELL(NBLOC\_MAX)$$

which store the dimensions of the mesh (NI, NJ and NK) and the total number of cells (which includes the internal plus two layers of ghost cells) of the given block ($NCELL = (NI + 3) \times (NJ + 3) \times (NK + 3)$).

The table 1 summarizes all defined arrays of pointers: for each one a very brief description of the data stored and the number of memory locations needed is given (in the general case of turbulent combustion with four scalars); the size is to be intended in unit of $NCELL$.

The storage of old dependent variables is needed for the time advancing solution method.

Name	description	size
IDEPVP	dependent variables	10
IDEPOP	old dependent variables	10
IDISP	distance from solid walls[1]	1
IRRR	density	1
IMUP	turbulent viscosity	1
IDIAP	pressure correction	1
IODIAP	old pressure correction	1
IVOLP	volumes	1
ICOOP	coordinates	3
INORP	normals	9
ILLP	cell reference length	3
IDTP	time step	1

Table 1: List of arrays of pointers.

Besides the pointers to permanent memory locations, in the COMMON block six pointers are stored. They are used as temporary, and so *poped* and *pushed* whenever it is needed. Generally these arrays are not statically stored in COMMON but declared in the subroutines where they are used. These six arrays are stored in COMMON because they are used in the solver, at each time step, in many subroutines; so it is useful to have them statically stored; the effect is that just inside the solver they are treated as permanent pointers. They are *imat, irhs, ixcg*, that are used in solving a linear system; *ilprec, idprec, iuprec* that are used to precondition the matrixes of the linear system, whenever a preconditioner is used.

1.3 Boundary conditions

In *ARES* code a number of choices are available to impose physical boundary conditions.

In case of multi-block meshes where different blocks are connected, it is necessary to exchange data between them. Two layers of ghost cells surrounding the block computational domain are then used to store field quantities.

The data exchange is schematically shown in figure 8. After the data exchange, all blocks can compute the fluxes at the boundaries having all the necessary internal

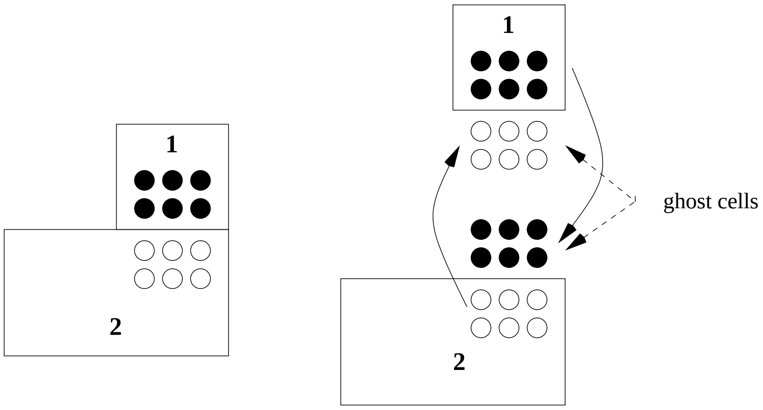

Figure 8: Data exchange

and external data, in such a way the fluxes are computed as if the block subdivision didn't exist.

The meshes used are structured and conformal. This implies that all the points are defined by three indices I, J, K. In subroutine *readbc* the parameters used for the boundary conditions setting are read. First, 5 indices are read for all the sides of each block:

itype, ist1, ien1, ist2, ien2.

They are 5 *integer* which represent the type of BC to be applied (inflow, outflow, connection), the value of the first and of the last node of the two indices which describe the side considered. For example for a side defined by the equation $I = 1$ the indices J and K can change, and the parameters could be, for instance, $ist1 = 1, ien1 = nj, ist2 = 1, ien2 = nk$. An important feature of *ARES* code is the generality of the orientation of each block with respect to the others, the only limitation is that only structured and conformal grids are permitted.

In the case the "itype" parameter identifies a block connection, a second line of parameters is also read. These are:

icsi, ics1, inc1, ics2, inc2.

They are 5 *integers* which indicate the side connected, the value of the indices at the first node and their increments (+ or - 1).

All these parameters are stored in integer pointers in the COMMON block.

In subroutine *dumcoo* a layer of ghost nodes is built up. So two layers of ghost cells are formed. In block-connections the value of the mesh at each node is filled with an exchange of data as shown in figure 8. After the ghost cells construction, all the indices of the boundary conditions are stored with new values. In fact they are written now for cell centered boundary and including the ghost cells. This is made because all quantities, but the mesh, are cell-centered entities.

In the connections among blocks it is basic to load and unload the array to be exchanged with the right order, because of the generality of the orientations of each block.

This is possible knowing the integer parameters read in *readbc*. These parameters describe how the side *iside* of the block *ibloc* is connected with the side *icsi* of the block *icbl*. A simply algorithm indicates also the orientation of the block connected and permits the loading of the array to be exchanged with the right order, it is:

$$\begin{aligned}
inc3 &= 1 \\
iss &= icsi + iside \\
iss2 &= (iss/2) * 2 \\
if(iss2.eq.iss) & \quad inc3 = -1 \\
in &= max(inc1 * inc2 * inc3, 0)
\end{aligned} \tag{1}$$

This procedure fixes the value of in, to be 0 or 1. If it is 0 the orientation of the blocks is such that the order of the indices must be inverted with respect of the natural order.

1.4 Time Integration

Equations for velocity, pressure and eventually scalars and turbulent fields are advanced in time by using an implicit integration method. The general form of the discretized system of partial differential equations to be solved can be written as:

$$\frac{\Delta Q}{\Delta t} = Res_{n+1} \tag{2}$$

By using an implicit scheme, the right hand side of system 2 at the temporal step *n+1* can be obtained by its linearization:

$$Res_{n+1} = Res_n + \left(\frac{\partial Res}{\partial Q}\right)_n \Delta Q \tag{3}$$

so that in order to advance the field from the time *n* to the next time *n+1* equation 2 can be rewritten as:

$$\Delta Q = Res_n \Delta t + \left(\frac{\partial Res}{\partial Q}\right)_n \Delta Q \Delta t \tag{4}$$

By defining the Jacobian matrix J_n as $J_n = \left(\frac{\partial Res}{\partial Q}\right)_n$ equation 4 becomes:

$$\left(\frac{\mathbf{I}}{\Delta t} - \mathbf{J}_n\right) \Delta Q = Res_n \tag{5}$$

which has the general matrix form:

$$\mathbf{M} \times \mathbf{x} = rhs. \tag{6}$$

where $\mathbf{M}$ is the matrix of the coefficients and $\mathbf{x}$ the unknown vector.

The right hand side of system 6 contains the cell residuals of the variables to be solved. The form of these fluxes depends on the spatial discretization scheme. With implicit time advancing method it is necessary to solve the linear system 6 at each time step. In *ARES* code there are two algorithms available: *CGS* and *BI-CGSTAB*. They are two standard libraries implemented in the code via public domain libraries *template*. They solve the linear system 6 through an iterative procedure. They have the same structure and in the following analysis both of them will be called simply CGS solver for the sake of simplicity. If requested a preconditioner is joined to the CGS solver.

The equation of motion, compressible Navier-Stokes, are solved with a fractional step method. In this method, first a velocity field is computed (that does not necessarily respect the steady continuity equation); then the correct pressure field is computed; later using the value of pressure field the velocity is corrected to enforce the continuity constraint.

2 Code description

2.1 Initialization

In the main program *init* is called.

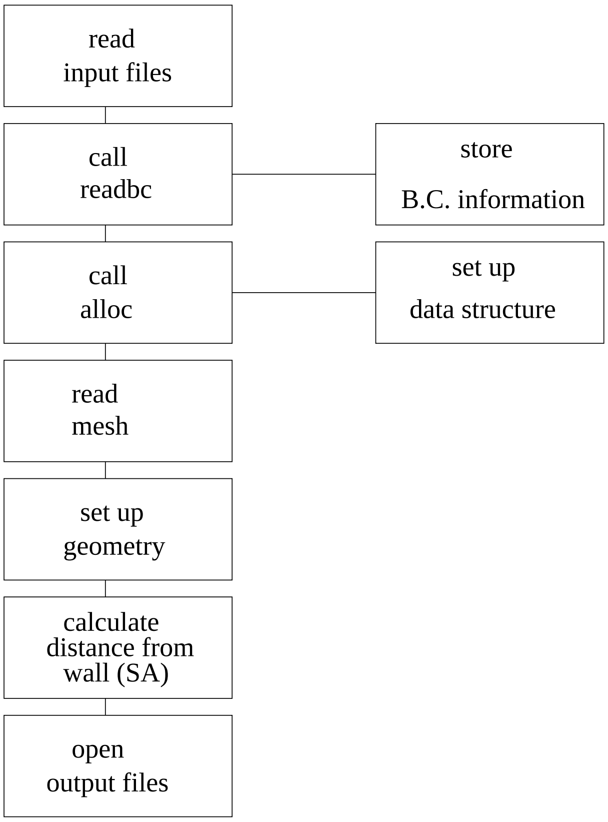

Figure 9: The `init` structure

In this subroutine all the initialization operations are made following the scheme of figure 9.

All the input files are read.

The subroutine *readbc* is called; here all the information about boundary conditions are read and stored.

The subroutine *alloc* is called, where the data structure described in paragraph 1.2 is set up.

The layer of ghost nodes is built up. All the geometrical cell variables (volumes and normals) are calculated. In the case of a turbulent simulation with Spalart-Allmaras model the distance from the nearest wall for each cell is computed.

Finally, all the output file are opened.

A detailed description of all the input and output files is reported in Appendix A.

2.2 The Solver

Once the initialization is finished the subroutine *fssolver* is called from main program. The solver is the core of the whole program and represents the time integration module.

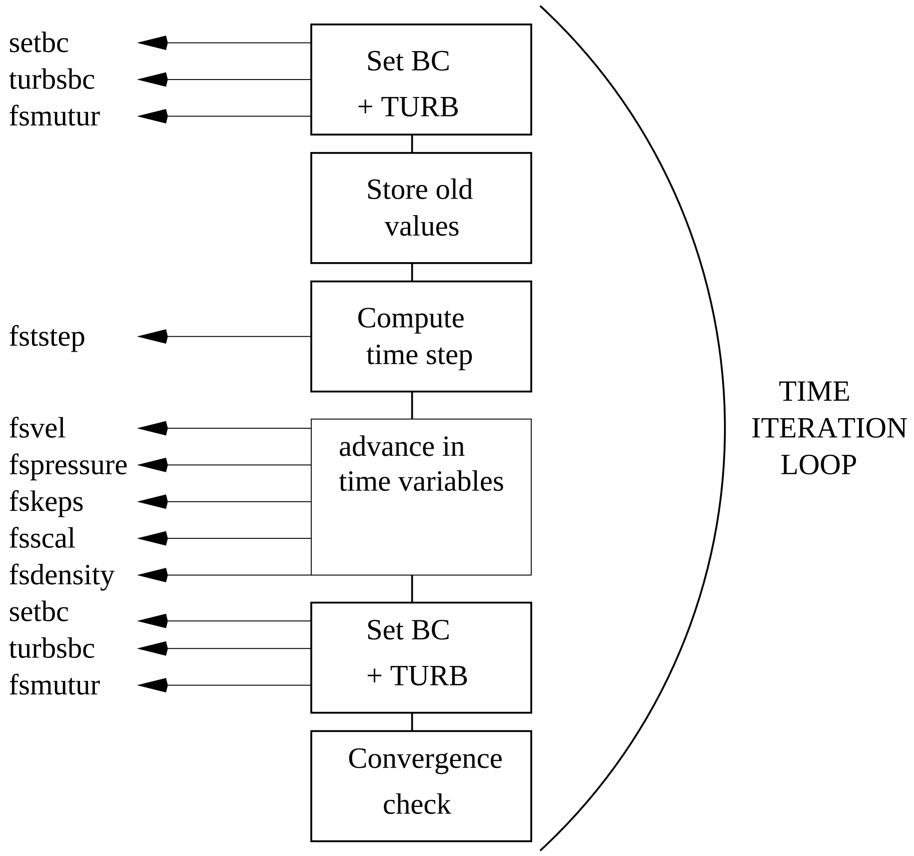

Figure 10: The `solver` structure

With reference to figure 10 at each iteration step the following is performed:

1. the subroutine *setbc* and *sabc* or *turbsbc*, respectively if Spalart-Allmaras or $\kappa-\epsilon$ turbulent model is used, are called; so boundary conditions are applied for all the dependent variables. In turbulent flows the subroutine *fsmutur* is called for computing turbulent viscosity.

2. values of dependent variables and of the pressure correction at previous time step are stored; the dependent variables, as shown in paragraph 1.2, are P, that is the pressure; U,V,W, which are the three velocity components; f_i, which are the scalars needed by the chosen combustion model; the turbulent variables.

3. Time step is computed. It is calculated on a local cell-basis, once the global CLF number is provided as an input information ($cfl = \frac{a\Delta t}{\Delta x} \leq 1$).

4. All the dependent variable fields are advanced in time. The subroutine *fsvel* is called that advances the velocity.

   The subroutine *fspressure* is called to advance the pressure and correct the velocity.

   The subroutine *fskeps* is called to advance the turbulent field following the chosen turbulent model.

   The subroutine *fsscal* is called to compute the scalar fields

   The subroutine *fsdensity* is also called. The density is calculated through the equation of state.

   The subroutine *fsmutur* is called, if the calculation is turbulent, for calculating the new value of the turbulent viscosity field.

5. Boundary conditions are applied to all dependent fields again.

6. Check on convergence level is made in order to test whether a prefixed value has been reached.

Block-loops are performed inside each module of routine *fssolver*. From a parallelization easiness point of view, the fact of having a loop over the blocks inside each one of the previously enumerated modules is quite appealing, because it fits with the natural block to processor approach.

All the modules but the fourth one perform their computation independently on each block and are intrinsically parallel. Communications among the blocks are where boundary conditions are imposed and in module 4. A more detailed description of what is performed at point 4 is presented in the next paragraph for a clarification.

2.3 Integration Module

In the above paragraph it has been shown, figure 10, that in order to advance in time the variable fields five subroutine named *fs"variable"* are called from the subroutine *fssolver*. It has been noted that the time marching algorithm is an implicit one and that it involves the construction and the solution of an algebraic linear system. This is made in these subroutines.

These subroutines have all the same structure, which is shown in figure 11. So only the first called, that is *fsvel*, will be analyzed, being implicit that the same arguments and observations will also be true for the other ones.

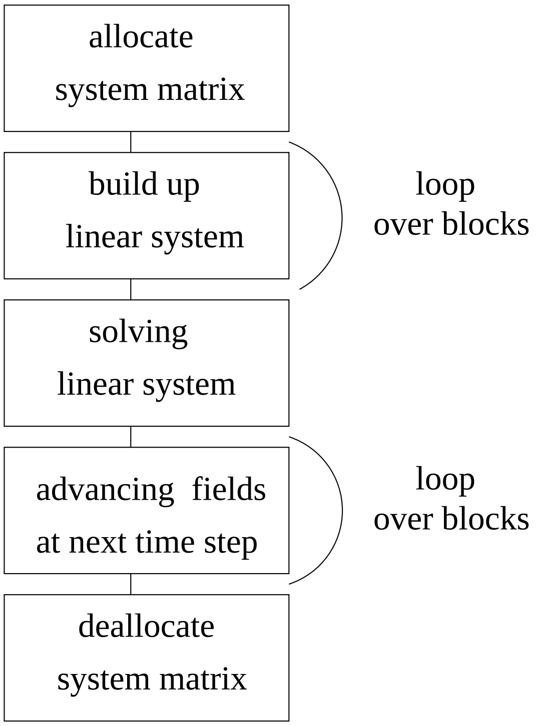

Figure 11: Structure of the generic *fs"variable"* subroutine. The second module is described in fig.13.

Some comments must be done on figure 11. To advance in time the variable fields an algebraic linear system of the form 6 has to be solved. At the beginning of all of these subroutines *fs"variable"* the matrix **M**, here called **A**, the unknown **x** and the right hand side **rhs** are allocated in **work** as temporary arrays, they are *pushed*. As noted in paragraph 1.2, the pointers which indicate the memory locations of these arrays are *imat* for **A**, *ixcg* for **x** and *irhs* for **rhs**. They are deallocated only at the end of the subroutine, figure 11, so they are treated as permanent arrays during all the calculation inside each subroutine *fs"variable"*. The difference between a real permanent array, for example that indicated by pointer idepvp, is that each variable needs a different pointer for a permanent array while the same pointers *imat*, *ixcg*, *irhs* are used in the solution of the linear system for each variable, that is in all *fs "variable"* subroutines. At this level it is known the size of these arrays. Dimensions of matrixes **A**, **x** and **rhs** are:

$$\begin{aligned}
\mathbf{A} &= (7 \times (NI+3) \times (NJ+3) \times (NK+3) \times nvar) \\
\mathbf{x} &= ((NI-1) \times (NJ-1) \times (NK-1) \times nvar) \\
\mathbf{RHS} &= ((NI-1) \times (NJ-1) \times (NK-1) \times nvar)
\end{aligned} \tag{7}$$

where $NI-1, NJ-1, NK-1$ are the total number of internal cells in all the computational domain, all blocks included; $nvar$ is the number of variables calculated and updated in the *fs"variable"* subroutine, for example they are 3 in *fsvel* for the velocity and 1 in *fspressure* for the pressure. The matrix corresponding to the pointer *imat* is not called **M** but **A** because of its dimensions. In the linear system 6 the matrix **M** must have a number of columns equal to the dimensions of the unknown **x**. Instead the memory allocation localized by the pointer *imat* for the matrix **A** is larger, for both internal and ghost cells are placed in the columns.

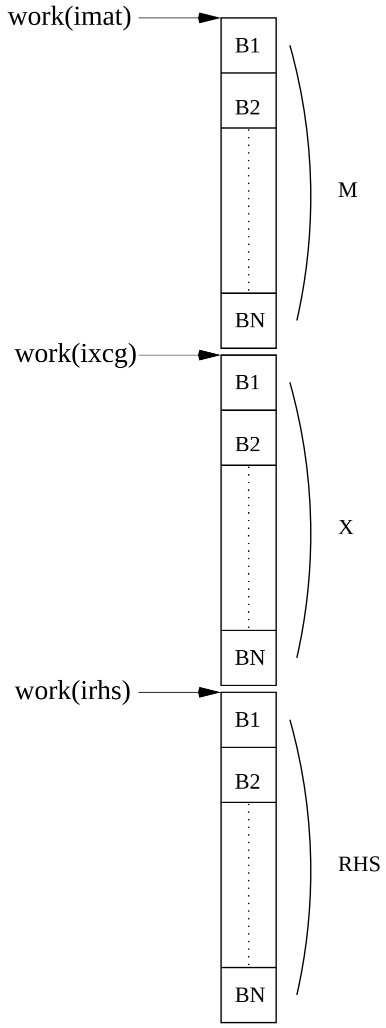

Figure 12: Matrixes of the linear system

Special attention has to be put on how the matrix **A**, the unknown vector **x** and **rhs** of system 6 are stored. The criterion used to store these arrays is different from the one that has been previously used to build-up the working array **work** (figure 6), as explained in figure 12.

The matrixes are evaluated inside a loop over blocks. At the end of the loop all matrixes , even if computed independently, are stored all together. So all the data concerning the linear system are stored in **work**. The fact that these arrays are constructed inside a loop over blocks is reflected by their form, as shown in figure 12, they are "block-matrixes".

At this level there is no *communication* among the blocks. At this point the overall system 6 has to be solved, where now **A** is a block matrix.

In figure 11 it has been underlined which parts of the subroutines *fs"variable"* are inside a loop over blocks. They concern the construction of the linear system and the updating of the variable fields, which takes place after the solution of the linear system. This is important because the calculation inside a loop over blocks is distributed among them and is computed independently by each one, in total agreement with the *one-block-one-processor* approach. The module concerning the solution of the linear system is outside from the loop. This means that not all the computation is performed independently by each block.

In the next paragraph the different modules of figure 11 will be "opened" for the analysis of the *fsvel* subroutine, as example for all the *fs"variable"* subroutines, in order to better comprehend the *ARES* structure and particularly its parallelization properties.

2.4 fsvel subroutine

With reference to figure 11, the first module contains the arrays allocation.

In the second module the linear system is built up. The flowchart of this module is shown in figure 13. It has been put in evidence in figure that the whole module is inside a loop over the blocks.

With reference to equation 6 it is necessary to calculate the matrix **M** and the right hand side. First the matrix **M** is calculated. This is done in three parts, as shown in figure 13. The Jacobian matrix is calculated sequentially for the inviscid terms (routine *fsjacinv*), the viscous terms (routine *fsjaclap*) and the unsteady term.

These terms are stored in the array **A** (the pointer *imat*).

It is remarkable to note that this structure is shared by all the subroutine *fs"variable"*.

The dashed line logic block in figure 13 is executed for the momentum equation only (fsvel). Here the pressure-velocity correlation is saved for the future use by the pressure correction scheme.

Once the construction of matrix **A** is finished, the right hand side **rhs** is calculated. It contains the cell residuals of the variables to be solved. As in the case of matrix **A**, first the inviscid terms are computed (subroutine *fsflxinv*), then the viscous contributions (subroutine *fsflxvis*). The construction of **rhs** also requires the

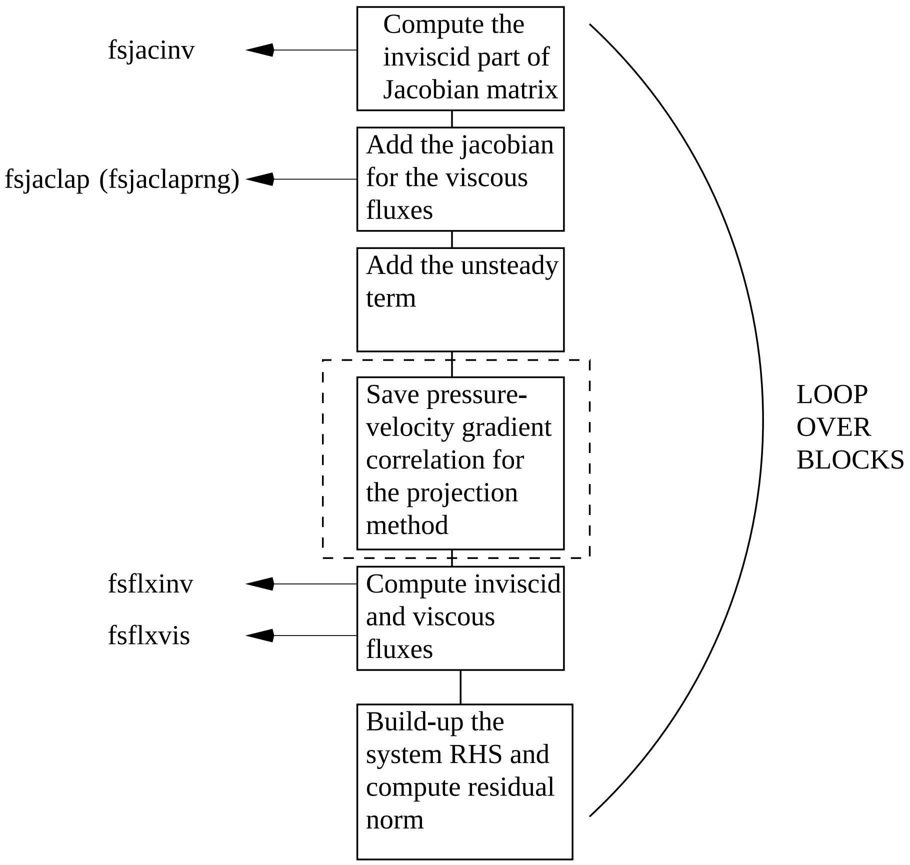

Figure 13: Implicit solver structure: first loop over the blocks

use of the ghost cells. The operations are made with the help of a temporary array **work**(*iflx*) This array is *pushed* with the right dimensions:

$$(NI + 3) \times (NJ + 3) \times (NK + 3) \times nvar.$$

When all the operations are made the internal cells of **work**(*iflx*) are copied into the array **rhs** and the temporary array **work**(*iflx*) is *poped*. This is made in the last module of figure 13.

In this module the maximum residual value is finally calculated for monitoring the convergence. At this point the linear system is built up and the loop over blocks ends.

The linear system is solved in the second part of the implicit solver, the third module of figure 11. The flow chart is represented in figure 14.

It has to be noted from figure 14 that the *old* values of the variables to be computed are firstly saved into a temporary array (subroutine *fsf2t*). This is done because their permanent allocation in the overall working array **work** is used inside *fscg*. Then their correct values are restored into **work** (subroutine *fst2f*).

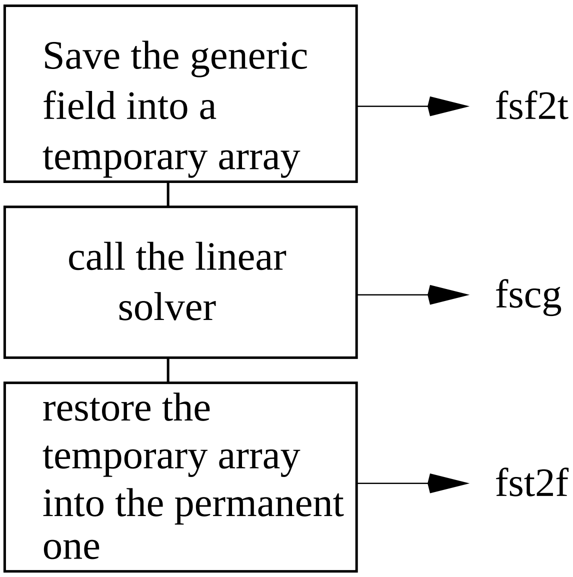

Figure 14: Implicit solver structure: call to linear solver

2.5 fscg subroutine

The structure of subroutine *fscg* is reported in figure 15. In *ARES* code a *conjugate gradient squared* algorithm is used for solving the algebraic linear system. This method may require a preconditioning of system 6. The preconditioning is carried out multiplying the vector **x** by a compatible matrix, composed by a choice of **A** coefficients. The necessary matrixes for preconditioning are allocated as temporary in **work**. There are four choices:

1. no preconditioning
2. diagonal preconditioning; the pointer *idprec* is allocated;
3. ILU preconditioning; the pointers *ilprec* and *iuprec* are allocated;
4. SOR preconditioning; the pointer *idprec* is allocated.

In the memory area localized by the pointers the coefficients needed for the preconditioning are calculated and stored. The subroutine *fsdiag* is called for a diagonal preconditioning, the subroutine *fspilud* for ILU preconditioning and the subroutine *fsdiag* for SOR preconditioning.

Then CGS solver is called. There are two available subroutines: *cgs* and *Bicgstab*. The system is preconditioned and solved in the unknown **x** with an iterative procedure, the details will be in the following.

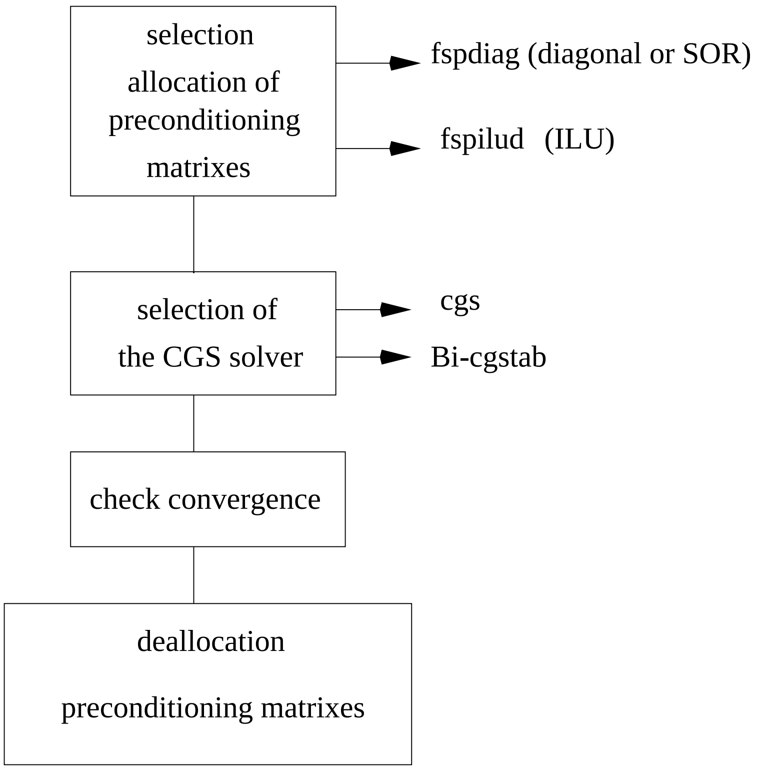

Figure 15: *fscg* structure

A check on the convergence of the solver iterative procedure is made at the end, if the tolerance requested is not reached, the array **x** is set to zero value.

In the last module of subroutine *fsgc* the temporary arrays allocated for the preconditioning are deallocated.

It has been noted that the two linear system solvers have the same structure, so a generic CGS solver is described. The observations are the same for both cases.

2.5.1 CGS solver

A schematic flow chart of the module "CGS solver" is given in figure 16.

With reference to figure 16, first the right hand side array content is copied in a temporary array for computing an overall norm. Both these operations are made for all the $(1 : ni-1, 1 : nj-1, 1 : nk-1, nvar, nbloc)$ elements of the RHS vector. It is important to note that while the first step (copying) can be easily parallelized without any needing of communication among processors, the second one requires message passing instructions.

In the third module of figure 16, all the scalar parameters requiring inversion are calculated. External standard libraries are used, they are *daxpy* and *dcopy*. These

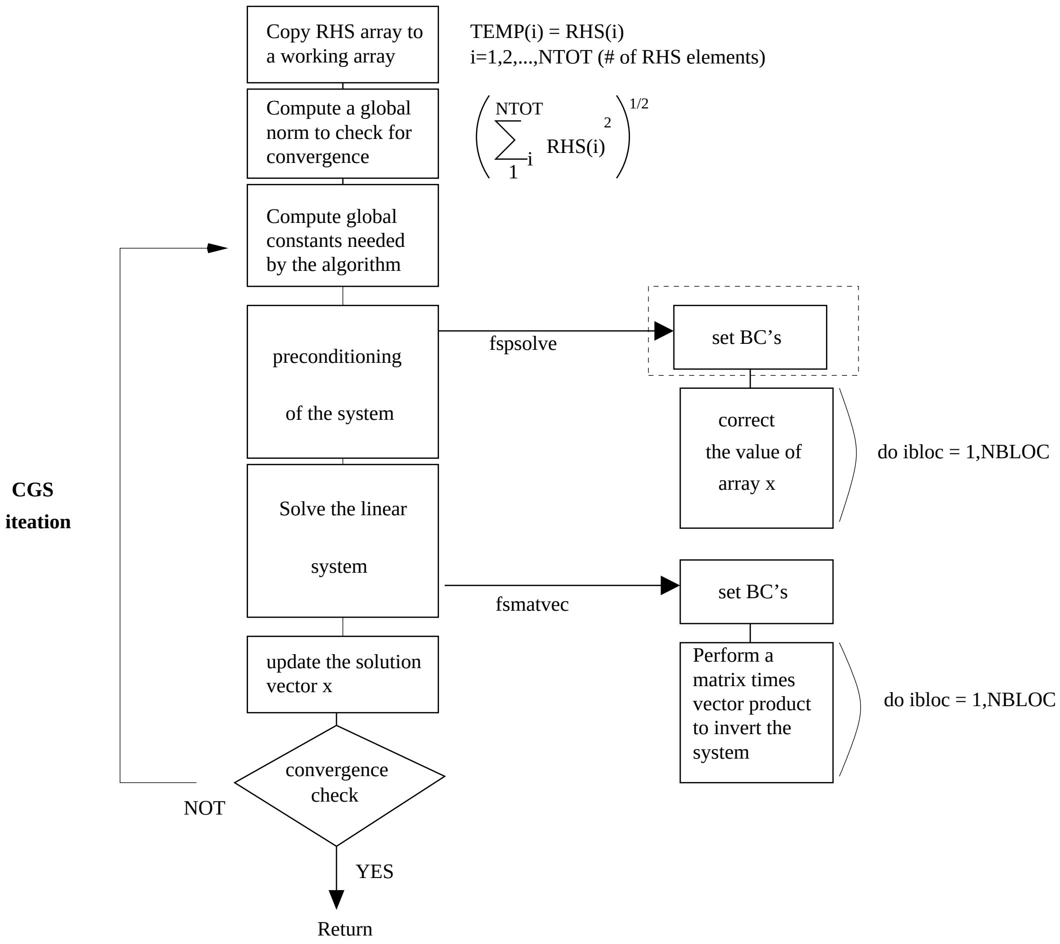

Figure 16: CGS solver structure

kind of libraries are totally local, that is of the form:

$$A(i) = \lambda B(i) + \mu C(i)$$

where λ and μ are parameters fixed by the algorithm, and A, B, C are some arrays. So they are made independently cell by cell, then they are intrinsically parallel.

Once all these preliminary operations are made the linear system can be inverted. First the system is preconditioned, if requested. The subroutine *fspsolve* is called. In figure 16 the first logical block joined to *fspsolve* is dashed because not all the preconditioner need the setting of boundary conditions.

As shown in equations 8, the dimension of the columns of the matrix **A** is not the same of the vector **x**, .

In order to resolve this difficulty the vector **x** is copied in the area of **work** memory dedicated to the corresponding dependent variable field; for example if the CGS solver is called from *fsvel* subroutine, it means that the velocity field is to be updated; in this case the vector **x** is copied in the **work** area localized by the pointer *idepvp*, from *idepvp* $+ NCELL$ for the first $3 \times NCELL$, that is where the velocity field has its permanent storage, figure 7. The resulting vector has a number of points equal to the points in the columns of **A**, being defined the dependent variables both in internal and in ghost cells. These points inside the subroutine *fspsolve* are reached by means of a local pointer named *icopy*. It is simply:

$$icopy = idepvp(ibloc) + ifld * ncell(ibloc)$$

where ifld is the parameter that indicates the field]considered, in the example its value is 1. The matrix operations are made with this array **work**(*icopy*) and at the end updated values are copied on **x**. The ghost cells are not used in these two kind of preconditioning.

In the case of diagonal and ILU preconditioning the structure is explained in figure 17.

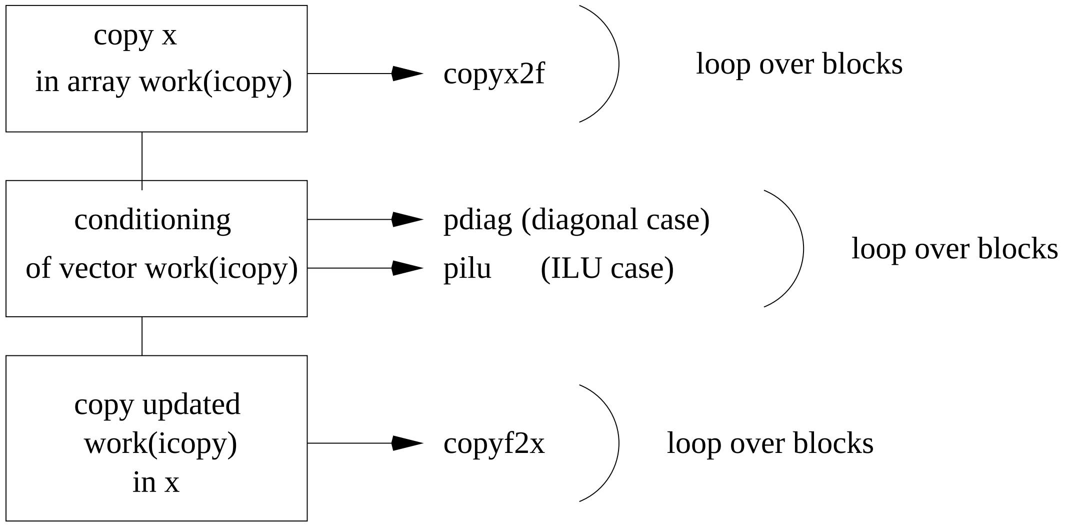

Figure 17: fspsolve structure in the case of diagonal or ILU preconditioning

Some columns of the matrix **A** are stored in the allocated array indicated by: *idprec*, in the diagonal case; *ilprec* and *iuprec* in the ILU case. The vector **x** is multiplied and divided by linear combination of them.

In the case of SOR preconditioning the operations are carried out following the figure 18. The structure is very similar. But in this case the preconditioner makes use also of the ghost cells to condition the internal values; so it is necessary to fill the ghost cells elements of the array **work**(*icopy*). It is important to note that only the internal cells are used for fields time updating; this is why the matrix **A**, and the vector **work**(*icopy*), are defined also for the ghost cells, but both **x** and **rhs** are not.

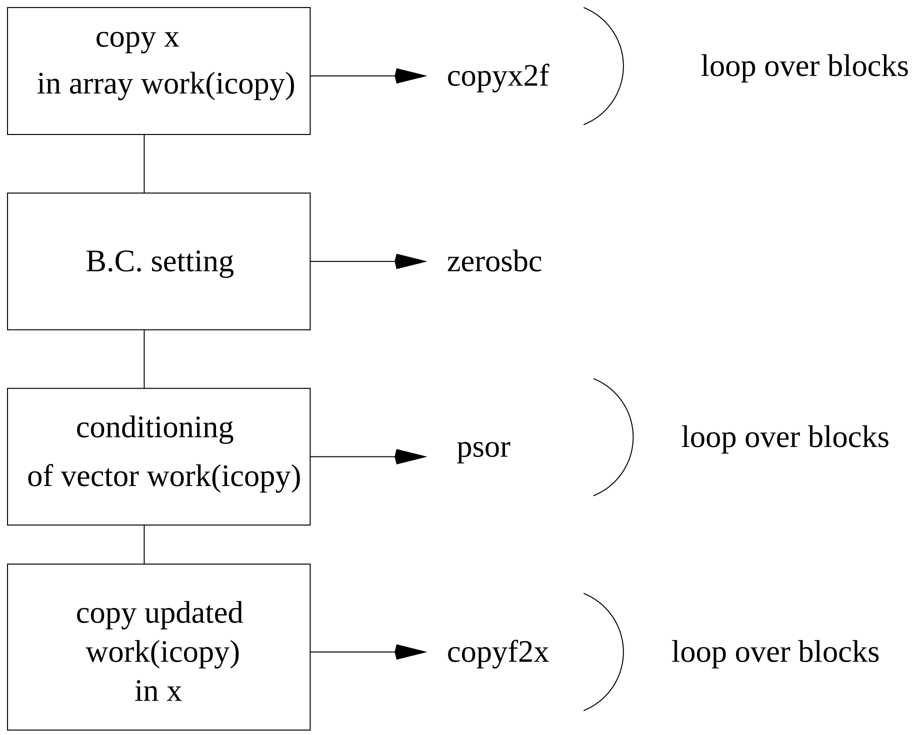

Figure 18: fspsolve structure in the case of SOR preconditioning

The ghost cells of vector **work**(*icopy*) are filled by setting boundary conditions like as for dependent variables, in the subroutine *zerosbc*; the operation was explained in paragraph 1.3 and figure 8.

As shown in figure 16, once the preconditioning is done the algebraic linear system is solved. Matrix-vector products are necessary for the scope. These operations are made in subroutine *fsmatvec*. The product is between the matrix **A** and the preconditioned vector **x**.

The subroutine *fsmatvec* follows a structure of the same kind of *fspsolve*, figure 19. The copy of the vector **work**(*icopy*) on the vector **x** at the end of the subroutine is not done here because the vector **x** is unchanged by this subroutine. Matrixes dimensions must be compatible, then the array **work**(*icopy*) is used with the ghost cells filled in the subroutine *zerosbc*. The presence of B.C. setting, as usual, implies the connection among blocks.

It must be noted that the area of permanent memory dedicated to the dependent variables (that is the area localized by pointer *idepvp*, figure 7) has been overwritten, solving the linear system. So it is basic the operation of saving dependent variables before calling *fscg* subroutine and of restoring them after it, in each *fs"variable"* subroutine, as described in paragraph 2.4, figure 13.

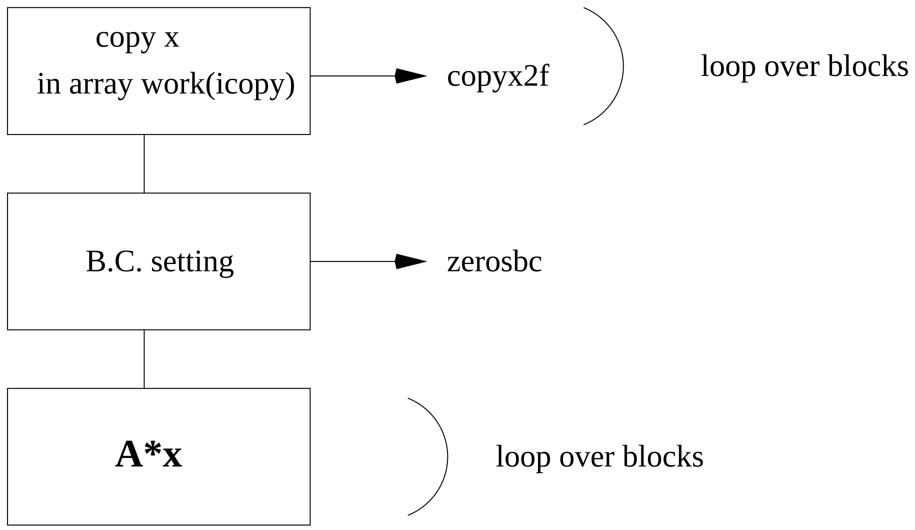

Figure 19: fsmatvec structure

The solver structure analysis leads to the conclusion that in the multi-block case a sort of *domain decomposition* is made, where the sub-domains are the blocks. It is important to note that this structure together with the implicit time advancing method causes differences between the multi-block and the one-block case.

When there is a block subdivision, a computational point is influenced by all the other points of the same block, and by the two layers of ghost cells, that represent the first two layers of cells of the adjacent blocks, figure 8, and not by the whole domain as in a one block configuration.

3 Analysis of parallelism

In the last section the blocks connection (B.C., global operations) and the block-loop structures have been underlined. Only the first one must be changed for the parallelization, because the block loops are intrinsically parallel. Three alternative choices are possible to achieve the goal of **ARES** parallelization, namely:

1. to parallelize only the first part of the code and not the solver;

2. as the previous point 1) but using parallel public domain libraries to solve the linear system;

3. to extend the parallelization also to the loops over the block inside the linear solver module;

Pros and cons of the above possibilities are discussed briefly in the following.

3.1 Approach # 1

It is the simplest one but it is also the less efficient. Substantially, the parallelization is limited to the building up of the linear system which is then solved by a serial algorithm. By following this approach the total number of the needed communication at each time step among the processors is $2 * nvar * ntxvol + nvar * ntcell * 7$. From previous experiences made on similar codes and because of the limited number of the communications among processors, it seems reasonable that the parallel part of the code would scale linearly with the processors number. In order to get some idea of the possible gain in term of speed up of the computation, a profiling of the scalar version of **ARES** has been run. Results proof that almost 50% of the run time is spent for the linear system solution, thus in the part of the code not interested by the parallelization procedure. In the ideal case of linear scalability of the parallel part and of an infinity number of processors, the maximum scalability factor would then be only 2.

3.2 Approach # 2

According to this second alternative, the same steps performed at point # 1 would be implemented, plus the use of public domain parallel linear solver libraries to be linked somehow to the existing code. In order to achieve such a way of parallelization some steps must be performed. Public domain libraries of *cgs* and *bi-cgstab* parallel solver have to be found. These libraries are built and optimized using a compact structure of the matrixes **A**, **x** and **rhs**. So a mapping of these matrixes is required for the transformation of the structure obtained in subroutines *fs"variable"* (figure 12) in the structure requested by the public domain library.

Problems connected to this second choice are in the first place that such suitable libraries have to be identified and implemented into the code and, secondly, that at this level, with only an "a priori" analysis of the code, it not possible to anticipate the code performances.

3.3 Approach # 3

This approach would consist in making parallel all the loops over the blocks which are used into the code. The same serial code structure would be substantially maintained, but the correct communications have to be ensured among blocks. This means to implement a message passing structure that replicates the serial B.C. setting; they are present in subroutine *fssolver*, inside the *SOR* preconditioning, subroutine *fspsolve*, and in the solution of matrix-vector products, subroutine *fsmatvec*. Moreover all the operations which were described as "global" need some communications among processors. This approach should provide a gain in performance, better than the first alternative and at least similar with respect to the latter discussed. At each iteration of the linear solver the setting of boundary conditions is called once or twice (in the case of SOR preconditioning) and other *global* operations are needed. This is an expensive computational cost in a parallel calculation, but since *fscg* represents about the 50% of the computation, it is expected not to be very relevant, as long as the blocks remain large enough (or the ratio $surface/volume$, which is a measure of comunication versus computation, is small).

4 Code Parallelization

In order to decide the kind of approach for the code parallelization, some tests with different linear solving algorithms were made. The failure of these algorithms and the structure of the code have suggested to follow the approach #3. The code is parallelized in a *block to processor* philosophy with the help of MPI message passing libraries. So the processors will execute the work made by each block in parallel, but at each time step and at each linear iteration all the boundary conditions are to be set and all the *global* quantities are calculated by the use of message passing.

In this section all the important changes made in the code to achieve the parallelization will be described.

4.1 Initialization

In the main program the MPI initialization is made. It means, simply, to write the three fortran lines:

$$CALL\ MPI\_INIT(error)$$

$$CALL\ MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, NPE, error)$$

$$CALL\ MPI\_COMM\_RANK(MPI\_COMM\_WORLD, MYPE, error)$$

In *init* the input data are read from files. Because of the *block to processor* approach every processor must perform an independent computation and so all of them must read the same data. A number of input files equal to the number of processors is necessary to can begin the calculation.

In the subroutine is called the routine *readbc*. In a parallel case connections between processors will be included. More information is requested than those needed by a block connection.

itype	description	bcvar(1)	bcvar(2)	bcvar(3)	bcvar(4)	bcvar(5)
1-200	Block Connection	icsi	ics1	ics2	inc1	inc2
400-600	Pe Connection	icsi	ics1	ics2	inc1	inc2

itype	description	bcvar(6)	bcvar(7)	bcvar(8)	bcvar(9)
1-200	Block Connection	-	-	-	-
400-600	Pe Connection	idsend	idrecv	N1	N2

Table 2: Boundary conditions input data.

The codification of boundary conditions for block connection and processors connection is explained in table 2.

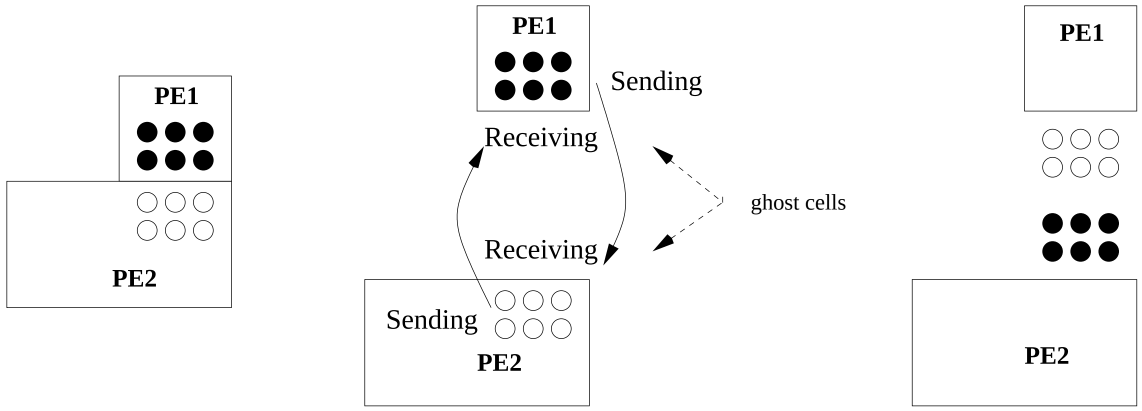

Figure 20: Data exchange between two processors.

Five integer index are assigned to set the connected boundary conditions. The connected side (*icsi*) of the connected block is identified by the value of $bcvar(1)$. The values of $bcvar(2)$ to $bcvar(5)$ identify the starting indices (ics1,ics2) and the increments (inc1,inc2) in the two directions of the side.

For connection between two processors, the first five index have the same meaning. In addition other four index are assigned. Infact for the success of the sending and receiving operations some tags are requested which identify the messages. These tags are read from input and they will remain the same for all the computation. $bcvar(6)$ is the tag assigned to the message to send and $bcvar(7)$ to the message to be received. $bcvar(8)$ and $bcvar(9)$ are the number of nodes in the two direction of the side.

In subroutine *alloc* all the pointers are defined. In this subroutine the number of cells of each processor is also calculated. In the parallel case it is useful to have also defined the total number of cells in the whole domain. For this scope a collective mpi routine is called that returns to all the processors this information. The mpi routine is called **MPI_ALLREDUCE** the syntax is:

$$call\ mpi\_allreduce(input, output, count, datatype, operation, comm, error)$$

In this case the input data are the ncell value of each block, the output is the sum of them and the operation is *mpi_sum*.

After the mesh is read the layer of dummy coordinates is set up. This is made in subroutine *dumcoo.f*. In the parallel case it is needed a data exchange between near processors. It is made as in a normal block connection, but the data exchange is made by using message passing as in figure 20.

The operation is divided in two parts.

First the each processor prepares in a buffer the array with the information to give to the connected one and this buffer is sent with mpi, figure 21. The routine

input	input data
output	output data
count	number of elements in send buffer
datatype	datatype of each send buffer element
operation	kind of operation to compute
comm	communicator
error	error number

that is called for sending the message is **MPI_IBSEND** and syntax is:

$$call\ mpi\_ibsend(buf, count, datatype, dest, tag, comm, request, error)$$

Second the processors receive the messages and store the new information. The steps are explained in figure 22. The subroutine called for the receiving operation is **MPI_RECV** with syntax:

$$call\ mpi\_recv(buf, count, datatype, dest, tag, comm, error)$$

The arguments meaning are described in table 3.

buf	initial address of send buffer
count	number of elements in send buffer
datatype	datatype of each send buffer element
dest	rank of destination
tag	message tag
comm	communicator
request	control number
error	error number

Table 3: Arguments in sending and receiving operation

4.2 Solver

Boundary conditions for connection among processors are treated in the same manner as for the dummy coordinates. This operation is repeated for all the dependent variables.

As shown in figure 16 inside the linear solver the boundary conditions are set and further some *global* operations are made.

The global operations, described in section 2.5, are Euclidean norms performed on the whole domain. In this kind of operations the commutative property of the

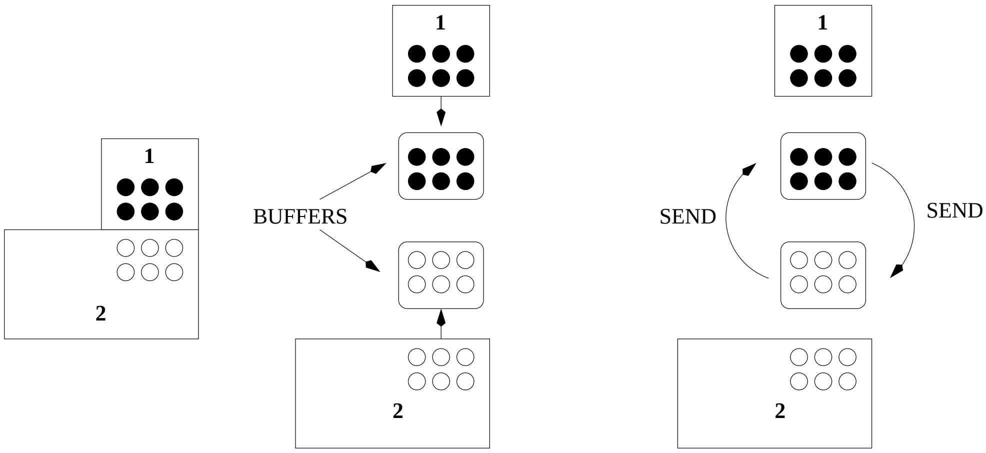

Figure 21: Sending operation.

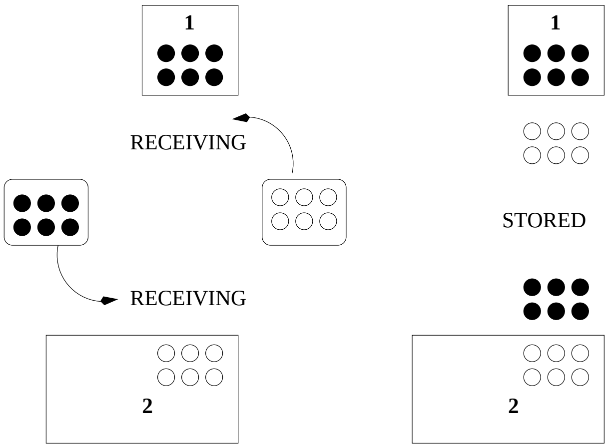

Figure 22: receiving operation.

operations is not conserved numerically, because of the truncation error; in some cases the change of the order can carry big differences in the result.

To reach the same results the procedure is divided in two parts, see figure 23. First, each processor makes its own serial computation. Then all the data are put in a single array in the first processor exactly as it would be in a serial calculation with n blocks. This operation is possible in parallel by calling a mpi collective subroutine called **MPI_GATHERV**. Processor #1 performs the norm having all the data needed and in the same order that in the serial case. Of course the norm performed in this way is safe for the result, but is very expensive in terms of computational time. In fact the norm is made in serial way, so as with a single processor, and moreover a collective mpi operation is called. This is an intrinsic limitation of this kind of algorithms that diminishes the performance of the parallel code hardly. Another method

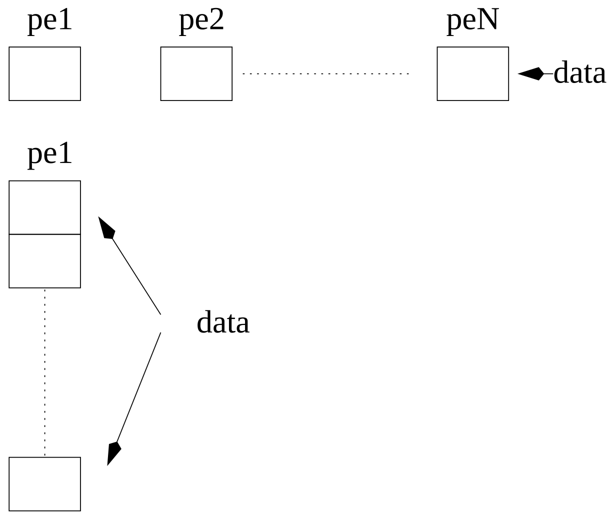

Figure 23: Norm in parallel.

was implemented in order to improve performances . The norms are calculated separately by each processor and the global norm is obtained summing the results with the help of mpi routine **MPI_REDUCE**. it does not maintain the serial order; the routine **MPI_REDUCE** has the same syntax of routine **MPI_ALLREDUCE**, the only difference is that **MPI_REDUCE** returns the result only to one processor. From a mathematical point of view the two methods are equivalent but not numerically. In all test cases performed the differences appear not important and the results are corrects, with a powerful gain in performance.

4.3 Convergence

Finally the history of convergence is written each time step . The maximum residual, the *rms* value and the position of the maximum residual are calculated for each dependent variable. These are *global* quantities so mpi collective operations are required for computing their correct values. This is made in subroutine *glob_con* called in *fssolver*. The maximum are calculated locally and then the maximum among all the processors is chosen with its position (also the processor). The rms value is computed summing the each processor value. The mpi routine called to do these operations is **MPI_REDUCE**. The operation computed inside the mpi routine is always *mpi_sum*.

5 Validation

At this stage of development the test cases performed had to assure the correct working of the code. In particular two boundary layer cases (a laminar and a turbulent flat plate), and a *Moreau* combustion chamber. All the cases are computed with both the algorithms available to solve the linear system, the *cgs* and the *bi-cgstab*. In all the cases the results are exactly equal to those obtained with the serial simulation. In figures 24, 25 and 26 are reported the convergence history in the three cases. It is not possible to distinguish between the serial and the parallel simulations.

The speed-up of the code is studied in one of the test case. The Moreau combustion chamber was analyzed varying the number of processors from #1 to #8. In the table 4 the result of the serial code against those of parallel one is shown.

proc.	Cpu time	SpeedUp
1(2 blocks)	438.26s	1.0
2(2blocks)	230.17s	1.91
1(4blocks)	482.29s	1.0
4(4blocks)	189.01s	2.56
1(8blocks)	535.97s	1.0
8(8blocks)	135.24s	3.96

Table 4: Performances

In the first column the number of processors used is shown. The cpu time grows with the number of blocks due to the increased comunication work inside the linear system algorithm; this is the reason why a number of 1-processor runs have been made with different number of blocks. Then cpu-times and speed-up factors are shown. From the results it is possible to see that the performance of the code, as expected, presents some scalability but the speed-up is far from the ideal linear speed-up. The performance of the parallel implementation will be assessed in more details in the coming report about the overall validation of the code.

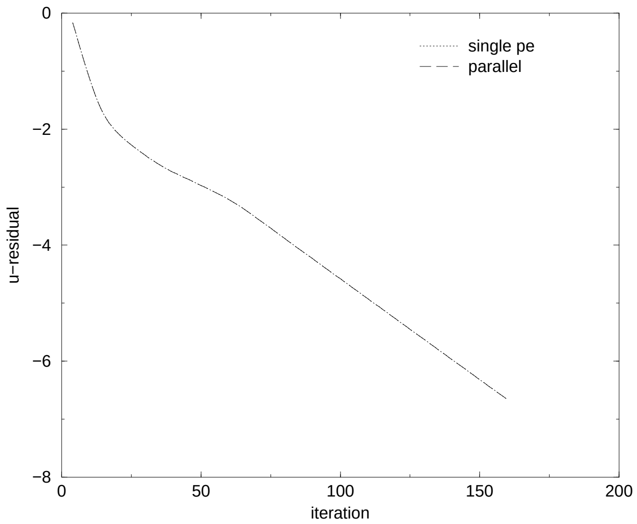

Figure 24: Convergence in the laminar flat plate.

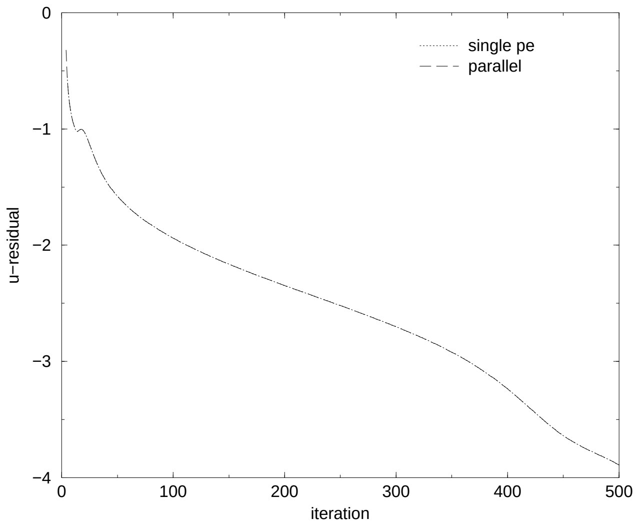

Figure 25: Convergence in the turbulent flat plate.

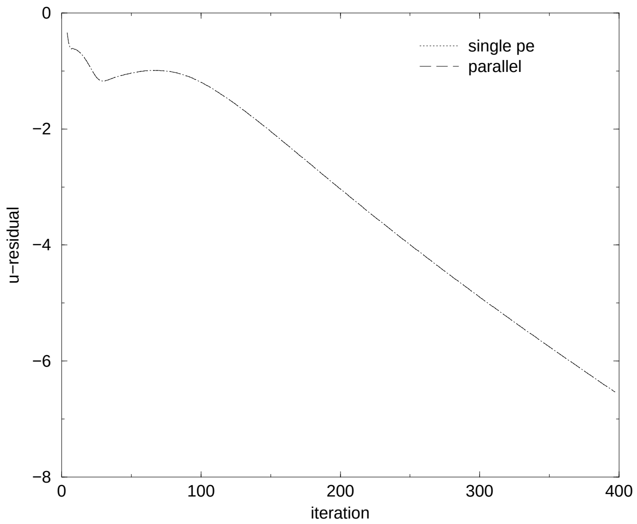

Figure 26: Convergence in the Moreau combustion chamber.

A Input and Output files

Many informations about the kind of simulation the code have to compute are written in input files. The input and output files follow the form

prefix.**ext**

where **prefix** is the name of the test-case. In the subroutine *init* these files are open and read from the file **PREFIX**, and **ext** corresponds to the following extensions:

dat, **msh**, **ini**, **con**, **out**, **tec**, **log**.

In the case of reacting calculation the files **prefix.pre** and **prefix.diff** can be also read. One or more files **prefix.jet***n* should be present if external velocity or turbulent profiles are used.

A.1 The dat file

Input data are read from the **dat** file. Here are all the informations needed to choose the kind of simulation (choice of algorithms, model of turbulence and combustion, geometrical informations etc.). Once the **dat** file has been read, if requested by the chosen model of combustion , other input files are read with the right value of the parameters used in the model.

A.2 The bc file

The file with suffix **bc** contains all the informations needed to a correct treatment of boundary conditions. The dimensions of grid blocks are read. The indices for all sides and segments are assigned. The kind of conditions and the number of cells for all the boundaries are read and stored in integer arrays. At this level the profiles of velocity and of turbulence variables are read if they are assigned.

Many kind of boundary conditions are allowed. There are definitions for different adiabatic walls, inlet and outlet profiles. There are all the parameters needed in block connections. The multi-block structure requests a change of data between near blocks to asses the correct value of residuals on the boundary. The data exchange is done in only one step. The memory locations corresponding to two layers of ghost cells are filled with values of the corresponding inner field dependent variables. These represent all the informations needed to compute the fluxes through connected boundaries.

Using the informations contained in **dat** and **bc** the **work** data structure is set up, assigning a value to all the pointers needed for the simulation.

A.3 The msh file

The mesh is read from the binary file **msh**. To treat correctly the boundary two layers of dummy coordinates are constructed around the boundaries of each block using the value of coordinates and normals of the boundary cells.

This is made in the subroutine *dumcoo*. All volumes and normals, also for the dummy cells, are set.

A.4 The con file

Because the simulations that must be done are almost always stationary, the calculation is expected to go to convergence after a number of iteration. It is important to monitor this behavior in order to decide when and if the computing is gone to success. The most common choice is to follow the advancing of the right hand side of the moment equations changed in sign, which, in a stationary calculation, goes to zero iteration after iteration. This hand of equations are the fluxes of the dependent variables which are called also residuals. In code **Ares** the residuals of all variables used in the calculation are stored. In the first 3 iteration their absolute value is stored and written in file **.con**. The third is taken as reference residual res_0 for the following iteration. At the iteration i the residuals are related to the reference value as

$$log_{10}(\frac{res_i}{res_0}),$$

this expression is evaluated for the sake of simplicity. The calculation is considered gone to engineering convergence when this expression is less then 10^5.

With the value of normalized residuals their location in the grid is also written. For each variable the indices i, j, k of the cell and the block where the maximum absolute residual is found are written.

A.5 Restart simulations

When the simulation is a restart of an older one, the given solution is read from the **ini** file, that contains the solution of a previous calculation. The code must also know the convergence history of the previous calculation. Hence the **con** file of the previous run must be present. The **con** file contains all the convergence informations, the last iteration, the time step and the values of max residuals.

A.6 The Output Files

The output files are opened at first iteration. The values, in every cell of the computational domain, of the field variables at the end of a run session are stored in the **out** file.

The **tec** file is an unformatted file intended for a successive management of the solution data (visualization, further elaboration, etc.). In the **tec** file the mesh coordinates and the value of fields variables are stored.

The **log** file is an *ascii* file reporting a summary of the run characteristics.

References

[1] Mulas, M., Beeri, Z., Golby, D., Surridge, M., and Talice, M. *"A parallel Navier-Stokes code for large industrial applications"*. P. Kutter et al. (eds), Lecture Notes in Physics, Springer 1997, pagg. 450-455.

[2] Mulas, M., Chibbaro, S., Delussu, G., Di Piazza, I., and Talice, M. *"The CFD code Karalis"*. CRS4-TECH-REP 00/87, 2000.

[3] Metcalf, M, Reid, J. FORTRAN 90/95 explained. Oxford University Press, 1996.

[4] XL FORTRAN for AIX, Language Reference. IBM, 1994.

[5] XL FORTRAN for AIX, User's Guide. IBM, 1994.

[6] Appunti scuola di parallelismo del CINECA. CINECA, 1998.

[7] MPI:A Message-Passing Interface Standard. 1995.