Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Enabling χ*navis* CFD Solver for Massively Parallel Simulations

Riccardo Broglia[a,*], Stefano Zaghi[a], Roberto Muscari[a], Francesco Salvadore[b], Soon-Heum Ko[c]

[a]CNR-INSEAN - National Marine Technology Research Institute, Via di Vallerano 139, Rome 00128, Italy
[b]CINECA, Via dei Tizii 6, Rome 00185, Italy
[c]NSC - National Supercomputing Centre, Linköping University, 58183 Linköping, Sweden

Abstract

In this paper, the work that has been performed to extend the capabilities of the χ*navis* software, a well tested and validated parallel flow solver developed by the research group of CNR-INSEAN, is reported. The solver is based on the finite volume discretization of the unsteady incompressible Navier-Stokes equations; main features include a level set approach to handle free surface flows and a dynamical overlapping grids approach, which allows to deal with bodies in relative motion. The baseline code features a hybrid MPI/OpenMP parallelization, proven to scale when running on order of hundreds of cores (i.e. Tier-1 platforms). However, some issues arise when trying to use this code with the current massively parallel HPC facilities provided in the Tier-0 PRACE context. First of all, it is mandatory to assess an efficient speed-up up to thousands of processors. Other important aspects are related to the pre- and post- processing phases which need to be optimized and, possibly, parallelized. The last one concerns the implementation of MPI-I/O procedures in order to try to accelerate data access and to reduce the number of generated files.

Project ID: 2010PA1461

1. Introduction

In this paper, the work that has been done to extend the capabilities of χ*navis* software is reported. χ*navis* is a general purpose solver developed at CNR-INSEAN; the code is based on the unsteady Reynolds averaged Navier-Stokes equations. The algorithm is formulated as a finite volume scheme, with the variables co-located at cell centers. Turbulent stresses are taken into account by the Boussinesq hypothesis; several turbulence models are implemented, ranging from classical algebraic (Baldwin-Lomax) to several differential models (k-epsilon, k-omega, Spalart-Almaras). The solver is also able to perform LES, DES or DDES simulations. Free surface flows can be treated by using a level-set algorithm. The code is based on a multi-block approach; complex geometries and multiple bodies in relative motion are handled employing an in-house dynamical overlapping grid approach.

Parallelization of the code is achieved by shared and distributed memory paradigms. The blocks are statically assigned to the available MPI processes. Due to the overlapping grids approach, data to be exchanged involve both block boundaries and inner computational cells. Besides, when dealing with moving grids, the send/receive patterns need to be adjusted (at runtime) at each time step. Communications are made by using both blocking and non-blocking MPI procedures. The computational work within each block is spread among the available OpenMP threads, increasing the parallelism level.

The code has been validated in a number of simulations, mainly in the framework of naval hydrodynamics [19, 10, 27, 7, 22], aerodynamics [16] and renewable energy [26] and it has been proven to scale when running on order of hundreds of cores (i.e. Tier-1 platforms) [3].

The availability of the code is subject to an agreement with CNR-INSEAN that regulates the use of the code and the extension of the license.

The objective of this work is to make the Computational Fluid Dynamics code χ*navis* capable of exploiting massively-parallel architectures, such as PRACE Tier-0 ones. It is worth noting that particular attention must be paid not only to the main solver but to the entire operating sequence needed to perform a simulation. As far as concerns χ*navis*, some pre-processing stages may become critical.

First, the *overset* algorithm, which is used to search for the donors/receivers map needed to handle the overlapping grids is dramatically memory demanding. Basically, such algorithm is devoted to describe how the blocks interact with each other. As it is often the case, in order to do that, the baseline χ*navis* preprocessor

*Corresponding author.
tel. +0-039-06-50299297 fax. +0-039-06-5070619 e-mail. riccardo.broglia@cnr.it

implementation of this algorithm requires the loading of the whole mesh into memory at the same time. It results that the algorithm is inherently very difficult to parallelize. Moreover, the *χnavis* specific algorithm is capable of handling a wide set of cases of interaction, thus increasing the parallelization effort. Obviously, the memory constraint becomes dramatic when the size of a mesh is huge and a strategy to limit it, as well as to possibly parallelize the execution flow, is crucial to handle huge grids.

Second, the balancing of the workload over the computational resources becomes of paramount importance on massively-parallel architecture. However, the complexity of the load balancing increases as the architecture parallelism grows. The baseline *χnavis* project did not have an automatic tool for load balancing, being based on a static human-dependent analysis of the workload. Consequently, to enable *χnavis* for massively-parallel architecture, an automatic load balancing preprocessor has been developed. This new preprocessor is based on a block-splitting algorithm being able to recursively balance the workload on huge computational resources.

Finally, parallel I/O is paid attention on large-scale numerical simulations these days as a reliable tool for accelerating the access to data files. Parallel I/O operation is made possible through the direct implementation of MPI-I/O functionality [12] or the deployment of high-end APIs (Application Programming Interface) such as NetCDF [20] and parallel HDF5 [15] libraries. High-end parallel APIs are recommended if the data format is easily converted to those library formats or the portability of the data file is an issue, i.e., data files are generated and visualised in different computing architectures. On the other hand, MPI-I/O enables the adoption of community-standard data format and in-depth tuning of I/O performance. In this work, we directly impose MPI-I/O functions onto *χnavis* solver since the code uses community-standardised I/O format and code developers share the common Linux-based computing environment.

The scalability of the solver is also presented. Several cases has been considered, including fix and moving grids tests. In the fix grid test, the numerical solution of the flow around a catamaran advancing in steady drift motion has been considered. This test allows strong scaling analysis on a rather large grid (80 million of volumes), the performances with up to 4096 cores have been investigated. The case which involves the moving grid algorithm concerns the simulation of a fully appended submarine advancing with a prescribed pure sway motion. In the latter case, a medium size grid (about 10 million of grid cells) has been taken into consideration; both strong and weak scalings have been tested, employing up to 4096 and 2048 cores, respectively.

In the paper, firstly a brief description of the mathematical and the numerical models will be provided. The work that has been done in the framework of the project will follow and it includes: parallelization and memory re-factoring of the overset preprocessor, development of an efficient and integrated block splitting/load balancing tool, parallel I/O and scalability analysis. A summary closes the paper.

2. Mathematical Model

The flow generated by a solid body moving on the free surface can be modeled by the unsteady Reynolds averaged Navier-Stokes equations. Within the assumption of an incompressible fluid, the set of equations is written in non-dimensional integral form with respect to a moving control volume $\mathcal{V}$ as

$$\begin{aligned} \oint_{S(\mathcal{V})} \mathbf{U}\cdot\mathbf{n}\ \mathrm{d}S &= 0 \\ \frac{\partial}{\partial t}\int_{\mathcal{V}} \mathbf{U}\ \mathrm{d}V + \oint_{S(\mathcal{V})} (\mathcal{F}_c - \mathcal{F}_d)\cdot\mathbf{n}\ \mathrm{d}S &= 0 \end{aligned} \tag{1}$$

where $S(\mathcal{V})$ is the boundary of the control volume, and $\mathbf{n}$ the outward unit normal; the equations are made non dimensional by a reference velocity (typically the free stream velocity U_∞) and a reference length L and the water density ρ. In equation (1), $\mathcal{F}_c$ and $\mathcal{F}_d$ represent Eulerian (advection and pressure) and diffusive fluxes, respectively:

$$\begin{aligned} \mathcal{F}_c &= p\mathbf{I} + (\mathbf{U}-\mathbf{V})\,\mathbf{U} \\ \mathcal{F}_d &= \left(\frac{1}{Rn}+\nu_t\right)\left[\mathrm{grad}\mathbf{U} + (\mathrm{grad}\mathbf{U})^{\mathrm{T}}\right] \end{aligned} \tag{2}$$

where $\mathbf{U}=(u,v,w)$ is the fluid velocity and $\mathbf{V}$ the local velocity of the boundary of the control volume. In the expression of the diffusive flux, $Rn = U_\infty L/\nu$ is the Reynolds number; ν being the kinematic viscosity, whereas ν_t denotes the non-dimensional turbulent viscosity. In the above equations, $p = P + z/Fn^2$ is the hydrodynamic pressure, i.e. the difference between the total P and the hydrostatic pressure $-z/Fn^2$, $Fn = U_\infty/\sqrt{gL}$ being the Froude number, g the acceleration of gravity, parallel to the vertical axis z (positive upward). In what follows, u_i is the i-th Cartesian component of the velocity vector (the Cartesian components of the velocity will be also denoted with u, v, and w). At the free surface, whose location is one of the unknowns of the problem, the dynamic boundary condition requires continuity of stresses across the surface; if the presence of the air is

neglected, the dynamic boundary conditions read:

$$
\begin{aligned}
& p = \tau_{ij} n_i n_j + \frac{z}{Fn^2} + \frac{\kappa}{We^2} \\
& \tau_{ij} n_i t^1_j = 0 \text{ when considering the vessel reference frame} \\
& \tau_{ij} n_i t^2_j = 0
\end{aligned}
\tag{3}
$$

where τ_{ij} is the stress tensor, κ is the average curvature, $We = \sqrt{\rho U_\infty^2 L/\sigma}$ is the Weber number (σ being the surface tension coefficient), whereas $\mathbf{n}$, $\mathbf{t^1}$ and $\mathbf{t^2}$ are the surface normal and two tangential unit vectors, respectively. The actual position of the free surface $F(x, y, z, t) = 0$ is computed from the kinematic condition

$$
\frac{\mathrm{D}\, F(x, y, z, t)}{\mathrm{D}\, t} = 0 \tag{4}
$$

Initial conditions have to be specified for the velocity field and for the free surface configuration:

$$
\begin{aligned}
u_i(x, y, z, 0) &= \bar{u}_i(x, y, z) \\
F(x, y, z, 0) &= \bar{F}(x, y, z)
\end{aligned}
\tag{5}
$$

3. Numerical Method

The numerical solution of the governing equations (1) is computed by means of the solver *χnavis*, which is a general purpose simulation code developed at CNR-INSEAN; the code yields the numerical solution of the unsteady Reynolds averaged Navier Stokes equations with proper boundary and initial conditions. It is formulated as a finite volume scheme, with variable co-located at cell centres. Turbulent stresses are taken into account by the Boussinesq hypothesis; several turbulence models (both algebraic and differential) are implemented. In the present computations, the one–equation model originally introduced by [23] has been used. Free surface effects are taken into account by a single phase level-set algorithm. Complex geometries and multiple bodies in relative motion are handled by a suitable dynamical overlapping grid approach; high performance computing is achieved by an efficient shared and distributed memory parallelization. In the following subsections the main features of the numerical algorithm are briefly recalled; the interested reader is referred to [6, 7, 8, 9] and [3] for details.

3.1. Spatial discretization

For the numerical solutions of the governing equations (1), the computational domain D is partitioned into N_l structured blocks D^l, each one subdivided into $N_i \times N_j \times N_k$ disjoint hexahedrons D^l_{ijk}. In the numerical scheme adopted here, the blocks are not necessarily disjoint, but can be partially overlapped, as it is explained in the following. Conservation laws are then applied to each finite volume:

$$
\begin{aligned}
& \sum_{s=1}^{6} \int_{(\mathcal{S}^l_{ijk})_s} \mathbf{U} \cdot \mathbf{n} \, \mathrm{d}S = 0 \\
& \frac{\partial}{\partial t} \int_{\mathcal{V}^l_{ijk}} \mathbf{U} \, \mathrm{d}V + \sum_{s=1}^{6} \int_{(\mathcal{S}^l_{ijk})_s} (\mathcal{F}_c - \mathcal{F}_d) \cdot \mathbf{n} \, \mathrm{d}S = 0
\end{aligned}
\tag{6}
$$

where $\mathcal{V}^l_{ijk}$ is the measure of the finite volume D^l_{ijk} and $(\mathcal{S}^l_{ijk})_s$ indicates its s-th face.

In order to achieve a second order accuracy in space, convective and viscous fluxes in the momentum equations, as well as surface integrals of the velocity in the continuity equation, are computed by means of the midpoint rule, and therefore all the quantities are evaluated at cell faces centroids. A standard second order centered scheme is used for the computation of viscous terms, whereas high order Godunov-type schemes [13] can be applied for the computation of Eulerian terms (see [8]): namely, a second order Essentially non Oscillatory (ENO) scheme [14], a third order upwind [25], or a fourth order centered scheme. For the simulation reported in the following, only the third order upwind scheme is adopted.

3.2. Temporal integration and dual time stepping

The semi-discrete system of equations can be recast in vector form as:

$$
\Lambda \left. \frac{\partial \mathcal{V} \mathbf{q}}{\partial t} \right|^l_{ijk} + \mathcal{R}^l_{ijk} = \mathbf{0} \tag{7}
$$

where $\mathcal{R}^l_{ijk}$ represents the flux balance for the finite volume D^l_{ijk}, $\Lambda = \mathrm{diag}\,(0,1,1,1)$ and

$$\mathbf{q} = \frac{1}{\mathcal{V}^l_{ijk}} \int_{\mathcal{V}^l_{ijk}} (p,u,v,w)^T \; \mathrm{d}V \tag{8}$$

is the volume average of the unknowns.

In order to remove any stability constraint on the time step, an implicit scheme is considered: the time derivative in the system (7) is approximated by a second order accurate three-points backward formula

$$\Lambda \frac{3\,(\mathcal{V}\mathbf{q})^l_{ijk}\,|^{n+1} - 4\,(\mathcal{V}\mathbf{q})^l_{ijk}\,|^{n} + (\mathcal{V}\mathbf{q})^l_{ijk}\,|^{n-1}}{2\Delta t} + \mathcal{R}^l_{ijk}|^{n+1} = \mathbf{0} \tag{9}$$

where the superscript n denotes the time level and Δt is the physical time step. In the solution of unsteady flows, equation (9) represents a system of coupled non-linear algebraic equations, that are solved iteratively by a dual time integration (see [17] for more details); to this end, a pseudo-time derivative is introduced in the discrete system of equations as

$$\widetilde{\Lambda} \frac{(\mathcal{V}\mathbf{q})^l_{ijk}\,|^{m+1} - (\mathcal{V}\mathbf{q})^l_{ijk}\,|^{m}}{\Delta\tau} + \Lambda \frac{3\,(\mathcal{V}\mathbf{q})^l_{ijk}\,|^{m+1} - 4\,(\mathcal{V}\mathbf{q})^l_{ijk}\,|^{n} + (\mathcal{V}\mathbf{q})^l_{ijk}\,|^{n-1}}{2\Delta t} + \mathcal{R}^l_{ijk}|^{m+1} = \mathbf{0} \tag{10}$$

$\Delta\tau$ being the pseudo time step, $\widetilde{\Lambda} = \mathrm{diag}\,(1/\beta,1,1,1)$ and β the pseudo-compressibility factor [5]. Then the solution is iterated to steady state with respect to the pseudo time τ for each physical time step; the system (10) is solved by means of the approximated factorisation scheme by [2]. Local dual time step and an efficient multi-grid technique are used in order to improve the convergence rate of the sub-iteration algorithm [11]. For steady problem, the solution can be achieved either as asymptotic solution of the unsteady problem or neglecting the physical time derivative in (10) and iterate in the pseudo-time only. The latter procedure has been used in the present work.

3.3. Single phase level set

As explained in the previous section, the free surface position, and thus the form of the computational domain, has to be found as a part of the solution. The presence of the free surface is simulated by means of a single-phase level-set approach [7], which is briefly recalled here. In classical level set approaches (see for example [21, 24]), a smooth function $\phi(x,y,z,t)$, whose zero level at $t=0$ coincides with the free surface, is defined in the whole computational domain (i.e. in both the air and water phases). The kinematic boundary condition (4) is extended to all the points in the domain, yielding a transport equation for the level set function

$$\begin{aligned} &\frac{\partial\,\phi(x,y,z,t)}{\partial t} + (\mathbf{U}-\mathbf{V})\cdot\nabla\phi(x,y,z,t) = 0 \\ &\phi(x,y,z,0) = d(x,y,z) \end{aligned} \tag{11}$$

$\mathbf{U}$ being the velocity of the underlying flow, $\mathbf{V}$ the grid velocity and $d(x,y,z)$ the signed distance from the free surface at $t=0$. The zero level of the level set function $\phi(x,y,z,t)$, being a material surface, represents the free surface location for $t>0$. Moreover, as $\phi(x,y,z,t)$ is initialised as the signed distance from the interface, the sign of the level set function remains unchanged at material points. In the single-phase algorithm adopted here, the Navier-Stokes equations are solved only in the liquid phase, identified by the negative values of the level set function. Outside, the velocity and pressure are simply extrapolated along the normal direction to the iso-surface of $\phi(x,y,z,t)$. The function $\phi(x,y,z,t)$ is re-initialized as a distance function at each time step in order to improve accuracy.

3.4. Overlapping grids approach

In order to cope with complex geometries or bodies in relative motion, the numerical algorithms were discretized on a block structured grid with partial overlapping, possibly in relative motion. This approach renders domain discretization and grid quality control much easier than with analogous discretization techniques implemented on structured meshes with abutting blocks. Of course, grid connections and overlapping are not trivial, as with standard multi–block approaches, but must be computed in advance. The algorithm used for this purpose is summarized in the following; the interested reader can look for details in [18, 9, 4].

- For each internal grid point, we check whether another block is overlapped to the same position. If so, of all possible overlapped grids, we retain only the local finest one, and we interpolate the solution on all the other coarser cells, marked as *holes*.
- For each block boundary that has not a simple connection with other blocks, or is not a physical boundary, as before we look for the smallest of all possible donor cells among the other blocks.
- Once a cell is identified as a *hole* or a *chimera boundary* and the main donor is found, a convex set of eight donors is searched for, and the solution is transferred to the *receiver* by a tri–linear interpolation.

When dealing with unsteady problems with moving boundaries, grid topology has to be recomputed at each time step. In order to render this computation efficient, donors and receivers are recomputed only when the blocks they belong to are in relative motion; this means that, for each grid point, we look for possible new donors and receivers only among the cells of blocks that are moving with the one under investigation, at the same time keeping fixed the connections between any block couple whose relative position does not change in time. Moreover, a nested search algorithm was developed, in which the multigrid structure of the code is fully exploited:

1. We start from the coarsest grid level where we check, for the possible receiver under analysis, every single cell of all other blocks in relative motion as possible donor: this is done in order to prevent stagnation of the search algorithm described at the next point when applied to non–convex blocks. This search is very quick, because each time that we go down one grid level, the number of cells is reduced by a factor eight (for instance, with three grid levels, the number of grid point to check is reduced by a factor $8^3 = 512$).
2. Once the donor is found on the coarsest grid, this point is used as the starting point of a minimization algorithm along the coordinate lines. Normally, 2 or 3 comparisons of squared distances are enough to locate the closest point in the Cartesian space. This location is then chosen as the initial guess for the next finer grid.
3. Once the finest level is reached and the closest point is located in the Cartesian space, the search is completed in contravariant coordinates to properly identify the closest point in the computational space and the convex set of eight donors.

There is another peculiarity in the algorithm that must be underlined, because it is not standard in chimera–type algorithms. In our approach, the interpolated solution is transferred among different chimera subgrids in a *body-force* fashion, without the need of fringe points. In particular, for the cells marked as "hole", a source term is added to the discrete equations:

$$\mathbf{q}^{n+1}_{\mathsf{chimera}} = \mathbf{q}^{n}_{\mathsf{chimera}} - \Delta t \left[\mathcal{R}^n + \frac{\kappa}{\delta} \left(\mathbf{q}^{n}_{\mathsf{chimera}} - \mathbf{q}^{n}_{\mathsf{interp}} \right) \right] \tag{12}$$

where $\mathcal{R}$ is the vector of the residuals, $\kappa = \mathcal{O}(10)$ is a parameter chosen through numerical tests, δ is the minimum between (non dimensional) cell diameter and time step, and $\mathbf{q}^n_{\mathsf{interp}}$ is the solution interpolated from the chosen donors belonging to overlapped grids. It can be easily seen that the last term acts as a forcing action that drives the solution toward the interpolated value $\mathbf{q}^n_{\mathsf{interp}}$. It is very convenient to write the equations in this form because, by doing so, we can retain the data organization of a structured code; consequently, it is very easy to retain, for instance, the multigrid iteration in its standard form, in spite of the *holes* in the grid. Moreover, these procedures are very convenient when dealing with moving blocks, where some points becomes active after being marked as holes. In fact, for these new points, we must know the solution at previous time steps, in order to compute the time derivatives. If the solution were not forced to the correct value, some special procedure should be masterminded to get the proper value.

3.5. Code parallelization

Coarse/fine grain parallelization of the unsteady RANS code has been achieved by distributing the structured blocks among the available distributed (nodes) or shared memory (threads) processors, and by spreading the computational work to be done (mostly in terms of do loop inside each block among available shared memory processors). Useful pre-processing tools, which allows the splitting of the structured blocks and the distribution of them among the processors, were developed for load balancing, whereas fine tuning is left to the user. Communication between processors for the coarse grain parallelization is obtained by using standard Message Passing Interface (MPI) library, whereas fine grain (shared memory) parallelization is achieved by means of the Open Multi Processing (OpenMP) library. The efficiency of the parallel code has been investigated in [3] (to which the reader is referred for details), where satisfactory speed up performances have been shown in different test cases.

4. Parallelization of Overset Pre-processor

4.1. Algorithm and implementation

As described in section 3., $\chi navis$ is capable of handling overlapping grids so that very complex geometries may be treated by means of structured multi-block meshes. When two or more grids overlap, $\chi navis$ assumes that for each (overlapping) volume there is exactly one "donor" grid and one or more "receiver" grids. For the donor volume, the system of equations is regularly solved, while for the receiver volume an additional term is added to the equations to force the solution to approach the donor corresponding value.

The most important task of the overset preprocessing phase is the detection of the overlapping zones, the definition of the receivers and the search of the correspondign donors. The algorithm used by the $\chi navis$ overset pre-processor to select the role of each volume is rather complex due to the several cases which have to be taken into account for the cell interactions.

In the baseline code, the preliminary step is to create the ghost cells for the blocks according to the adjacency boundary conditions. After that, the metric features (norms, volumes, ...) of the grid are evaluated for each

block and for each multigrid level. Then, another loop over blocks and multigrid levels performs the donor search, all the while distinguishing between the types of points, e.g. internal points and ghost points. Finding donors is mandatory for ghost cells where the equations cannot be solved. Moreover, boundary layers are treated with special algorithms to avoid unphysical behaviors. In order to allow an adequate donor search, blocks are initially grouped according to levels and priorities defined in the starting mesh file.

The most difficult issue arising when considering the overset search in the context of High Performance Computing is the memory demand. Basically, the algorithm requires the entire grid to be allocated when analyzing the dependencies among the mesh volumes. From the point of view of parallel computing, it is difficult to devise a domain decomposition because each block may potentially interact with any other block. Using a parallel paradigm based on one-sided communications (MPI 2 RDMA or Fortran coarrays) could be a solution to keep the algorithm unchanged. However, the performance would be very difficult to predict and, in any case, the source code would need to be strongly refactored.

On the other side, we decided to employ the MPI domain decomposition approach in a code adapted in order to minimize the usage of memory. The devised approach may potentially apply for a serial code but the MPI version was also implemented to enhance the performances of the final code.

We summarize the main changes to the existing code:

- The memory structures have been strongly modified using the Fortran 2003 syntax for derived types: this change allowed to read from file and easily store a selected subset of blocks from the whole grid. In the original code, for a given quantity, the values for all the blocks and multigrid levels are stored into a single one-dimensional array. A set of offsets is defined to select the block and the multigrid level to access. The implementation is a good example of mixed Fortran 77/90 style but it seemed to be not adequate to handle the loading/unloading of blocks to be processed and/or hosted. We defined three layers of storage using derived types: an array stores the different blocks, each block stores the array of buffers for different multigrid levels, and each buffer stores the values for each point. Fortran 2003 is the required standard since it allows for defining allocatable arrays inside derived types. A section of the Fortran module defining the structure of grid storage follows.

```
   type point_values
! index: buffer index
     type(point), allocatable :: val(:)
   end type

  type block
     logical :: myblock=.false.
     logical :: hostedblock=.false.
     ...
! index: mgr level
     type(point_values), allocatable :: bufgrd(:)
     type(point_values), allocatable :: bufcen(:)
     type(point_values), allocatable :: bufnsi(:)
     ...
  end type

! index: block index
  type (block), allocatable, dimension(:) :: blocks
```

- All the arrays of blocks and grids are allocated by each process when the code execution starts. But the val arrays storing the actual data for each point are allocated or deallocated during the code execution when needed. To manage this mechanism, for each block two logical variables are used to mark the blocks according to their role at a certain moment:
  - myblock: true for the processed block (one at a time), i.e., the block for which the donor search is being performed
  - hostedblock: true for all the hosted blocks, i.e., the blocks needed to perform the donor search for the processed block
- The preliminary stage is still devoted to build the ghost cells from adjacency boundary conditions. For each processed block, the required hosted blocks are the adjacent blocks. After all the blocks have been extended, the resulting mesh is written to file.
- The donor search is now performed one block at a time for all multigrid levels and in two different stages. For each stage, loops over blocks are employed and different computations are performed according to the role of the block in the context (processed block, hosted block, or none of these). The first stage needs to be completed for the whole grid before starting the second stage, while the temporary results are stored on the disk.

Stage 1: donor search for non-boundary layer cells (both inner and ghost cells): in addition to the processed block, the meshes of blocks "possibly" overlapping with the processed block are loaded into memory. To detect the possibly overlapping blocks, a bounding box rule is employed: if a cell of the block is inside the bounding box of another block, this block is marked as "possibly overlapping", i.e. as hosted block

Table 1: Performance comparison of the baseline OpenMP code with the parallel MPI code, devised to reduce the memory usage. The scalability of the MPI code is remarkably better than the OpenMP one. However, as expected, the performance loss is significant but still acceptable considering the achieved memory reduction.

# procs	serial (OpenMP)	Parallel (MPI)
1	26 s	118 s
8	12 s	32 s
16	11 s	25 s

Stage 2: donor search for boundary layer cells (both inner and ghost cells): in addition to the processed block, the meshes of the blocks corresponding to the same physical body are loaded into memory as hosted blocks. The algorithm to search a donor for the boundary layer points is not trivial. However, the detailed description of the changes in the code aimed at memory minimization and parallelization goes beyond the scope of the present paper and will not be reported here.

In the end, we could maintain all the features of the original code even in the final parallel code except for a particular case occurring at a few points of one of the test cases. In such a case, the standard donor search failed for a few ghost cells for which finding a donor was mandatory. Hence, the original code tried to search a donor as the nearest cell from the whole grid. This feature could not be acceptably implemented in the final code. Hence we decided to use another boundary condition for that case, exploiting the dynamical overset search of the *χnavis* main solver.

Using the sketched technique, we managed to lower the amount of memory required to perform an overset search. However, the stage (2) is still significantly memory demanding because the accuracy of the bounding box criterion is poor. Hence, we decided to add a sub-blocking approach to handle the stage (2). Basically, when processing a block, the memory needed to host the "possibly overlapping" blocks is computed and, if the value exceeds the memory availability, the processed block is split and the memory demand is recomputed considering only the considered sub-block. The splitting procedure continues until the memory constraint is fulfilled. Implementing this technique allowed to strongly reduce the memory requirements. It is worth underlying that at present only the processed block may be split in the implemented algorithm because of low-level code complexity which is not possible to accurately describe here.

From a programming point of view the changes to the code required a hard coding effort because of the complexity of the algorithms involved. As already mentioned, in addition to the memory minimization technique the code has been parallelized using a standard MPI decomposition. However, to keep the memory minimization feature each MPI process analyzes one block at a time as described above. The adopted load balancing technique is simple and an optimal balancing cannot be achieved because it would require a block splitting strategy, which is performed after the overset preprocessing stage, as detailed in the next section. The MPI/I-O paradigm has been also extensively used to read/write the data of different blocks from/to a single file, thus allowing a complete data interoperability between the baseline and the parallel overset preprocessor.

4.2. Test cases and performances

The overset MPI code was extensively tested and an accurate debugging phase was carried out to check the results in the several conditions allowed by the implemented algorithms. Finally, an analysis of performances was performed, as well. As stated above, the main task of the overset preprocessor parallelization is not to lower down the execution time but to limit the usage of memory. It is, in fact, expected that the technique employed to reduce the usage of memory tends to produce significant performance losses due to the intensive I/O operations involved and to the replicated computations of the metric properties of the grid. The original code is serial except for some sections which are parallelized using the OpenMP directives. On the other hand, the parallel MPI code may exploit inter-node parallelization thus allowing to recover or even to improve the performances compared to the original code.

At first, we tried to establish the significance of the performance loss comparing the original code to the MPI one when using a single node. The performance was measured on the EURORA cluster hosted at CINECA. It is an heterogeneous cluster but, for our purposes, we just exploited the Sandy Bridge cores with the following specifics:

- 2 eight-core Intel(R) Xeon(R) CPU E5-2658 @ 2.10GHz, 16 GB RAM

We measured the performance obtained by varying the number of cores used either by OpenMP threads (for the original code) or MPI processes (for the MPI code). As a first test, we selected a case for which the subblocking technique is not needed. The results are provided in table 1.

The performance loss using 1 core is large (more than 4x) but when using the entire node the scalability of the MPI code is much better than the OpenMP one. Actually, the OpenMP parallelization may be performed at present only on a minor part of the code. On the other side, the MPI code needs only a little amount of communications and the scalability limit is probably due to the RAM memory bandwidth and to the I/O hardware performances. However, when using the entire node the performance loss is less than 2x. Obviously,

Table 2: Hybrid analysis of the MPI code using 1 node (procs-MPI x threads-OpenMP). As expected, the OpenMP only gives a minor performance improvement since it is applied only to a small part of the code.

(1x16)	(2x8)	(4x4)	(8x2)	(16x1)
103 s	60 s	40 s	33 s	24 s

Table 3: Plot of the elapsed time of the MPI code versus the imposed memory limit for a 500 MB mesh. As expected, the performance loss is significant but the memory reduction is large, up to 6 times compared to the full usage.

Memory (MB)	Elapsed time (s)	Notes
500	22	No sub-blocking needed
400	22	No sub-blocking needed
300	26	Using sub-block splitting
200	29	Using sub-block splitting
150	26	Using sub-block splitting
100	36	Using sub-block splitting
80	42	Using sub-block splitting
60	Fails	Cannot split more

due to the large weight of the I/O operations, the performances of the MPI code significantly depend on the state/usage of the file-system when running the code.

We have to stress that the effect of block loading/unloading could be minimized by creating different categories of loaded blocks and by trying to keep the blocks hosted as long as possible. This further progress will be the subject of future works.

In order to fully exploit the parallelization layers of the code, we performed a hybrid analysis on the MPI code, since it also features the OpenMP original parallelization. As expected, the decomposition using MPI is much better with respect to performances and the advantage of adding OpenMP seems questionable for this case, as can be seen in table 2.

The second set of tests concerns the actual possibility of reducing the memory required by the code. Since processing each block requires to host the "possibly" overlapping blocks, it is clear that the required memory cannot be lowered down indefinitely. At first glance, it is expected that, the larger the total number of blocks is, the larger is the memory effective reduction. We performed some preliminary tests which confirm this assumption but the actual results strongly depend on the specific geometrical configuration of the grid.

We report here the results obtained with a mesh made of more than 400 blocks, and requiring a full amount of 500 MB of memory. This is obviously a very small case but the results still apply within a fair level of approximation when considering the same geometry and refining the mesh.

In order to test the memory reduction, we ran the code imposing different values of memory constraints. The results are provided in table 3. Imposing the 500 MB (i.e., no constraint) or the 400 MB limits, no subblocking is needed and, hence, the elapsed time remains constant. When decreasing the size of available memory, the sub-blocking is needed and, as expected, the performance loss becomes more remarkable, i.e. up to 2x when imposing the 80 MB limit. If we try to impose a 60 MB limit, the code executions fails. Inspecting the mesh, it results that there are some cells where the overlapping of several blocks occurs. This means that, even refining the subblock decomposition, there is no chance to reduce the size below the size of the blocks overlapping in a single cell. At present, to further decrease the required memory the only possible strategy consists in decomposing the initial mesh in order to increase the number of blocks and potentially reduce the size of the blocks overlapping in a single cell.

The last set of tests is an inter-node scalability analysis performed for a rather large case, requiring an amount of memory close to 22 GB. We employed again the EURORA cluster and decided to run 2 MPI processes per node. The memory limit to be imposed is 8 GB per MPI process since the EURORA node has 16 GB per node. As it is shown in table 4, the scalability is acceptable considering the realistic use cases for this preprocessor. The limitations are probably due to two main issues: (a) the load unbalance due to the naive assignment of blocks to the MPI processes; (b) the intensive usage of I/O operations.

5. Load Balancing

The χ*navis* solver is based on structured, body-fitted multi-block grids allowed to (dynamically) overlap each other (dynamic overset technique). Consequently, the efficient parallelization of large-scale simulations can be conveniently achieved by means of coarse-grain approach: each (structured) block is entirely distributed to one processor, thus the parallelization relies on the original domain decomposition performed during the realization phase of the numerical grids. In the coarse-grain approach the load balance is obtained through packing blocks

Table 4: Inter-node scaling of the overset MPI pre-processor. The scalability is acceptable and enables to recover the performance loss due to the memory minimization strategy adopted.

#nodes (2 procs per node, mem < 8 GB)	Elapsed time (s)
1	5760
2	3720
4	2280
8	1320

into groups according to some algorithms. Generally, this problem can be difficult to solve due to non-matching shapes of blocks-loads/block-number/processors-number. In particular, when the number of processors grows the problem becomes stiff. In a typical χ*navis* work-flow the load balancing is done without any automatism, but with only a trial-and-error approach that necessarily requires strong human effort. One of the aims of the present work is the development (from scratch) of an automatic load balancing tool that must be able to deal with an arbitrary number of processors.

5.1. Algorithm

The developed tool, referred in the following as *Load Balance*, is based on a block-splitting algorithm and it is statically used as a pre-processor. Presently, it has been assumed to take the load (per processor) balance as the only objective (cost) function, neglecting any eventual issues related to the increase of communications overhead when the blocks-split is applied. This is justified by a preliminary profiling work, which showed that, for typical simulations, the communication CPU time counts for less than 15% of the total CPU time.

Concerning CFD applications (same computations on all grid cells), it is reasonable to assume that the workload of a block is directly proportional to the number of cells. In a block structured block algorithm as χ*navis*, it reduces to

$$W_b = N_i \times N_j \times N_k \quad b = 1, 2, ..., N_B \tag{13}$$

where N_i, N_j and N_k are the number of cells along each Cartesian direction, and N_B is the number of structured blocks. As a consequence, the workload for each processor is defined as

$$W_p = \sum_{b=1}^{N_b^p} W_b \tag{14}$$

where the sum is done over all blocks assigned to the "p-th" processor, namely N_b^p. The algorithm developed is a modification of the one proposed in [1]. The descending-sorted blocks list is mapped over the processors list (assuming that the processors have homogeneous performances): the blocks list is skimmed through assigning each block to the processor with minimum workload. In order to describe the block-split balancing algorithm it is necessary to introduce some quantities. Firstly, the average workload per processor is defined as

$$\overline{W_p} = \frac{\sum_{b=1}^{N_B} W_b}{N_P} \tag{15}$$

where the sum is done over all the blocks, N_P being the number of processors. Then, the workload load ratio of each processor is defined as

$$W_{pr} = \frac{W_p}{\overline{W_p}} \tag{16}$$

Using such quantities, let us define the following constraint for the blocks-split algorithm: the blocks splitting process must produce a W_{pr} lying into a specified (threshold) range. After one assignment iteration, it is checked if all processors respect the W_{pr} constraint (i.e. if the workload is properly balanced). If this check fails, namely some processors are unbalanced, the blocks-split algorithm is activated.

In our implementation, different strategies are available for splitting the blocks:

- the largest block is split;
- the smallest block is split;
- a user-defined combination of large/small blocks are split using the user-specified threshold value.

The splits are always performed along the Cartesian direction having the largest number of cells, with a check on the next smaller one if the *multigrid* constraint is not verified. Indeed, the *multigrid* constraint must always be respected, i.e. the blocks-split algorithm must produce blocks allowing the same number of multigrid levels as the original non-split block. This constraint is respected allowing only splits being multiple of $(2_L^N)^3$, N_L being the number of multigrid levels used.

Once the blocks-split algorithm has been applied, the descending-sorted blocks list is reconstructed (placing the new blocks arising from the splitting in the proper position of the list) and a new assignment iteration is started. The algorithm ends when all processors have a W_{pr} less than the user-specified value.

The algorithm described above is summarized in the pseudo-code snippet 1.

Data: Blocks lists and processors number
Result: Balanced distribution of blocks on processors
compute the initial workload of each original block: W_b;
compute average workload: $\overline{W_p}$;
for *block* $\in$ *blocks* $-$ *list* **do**
  recursively split each block heavier than the average workload;
end
initialize descending-sorted blocks list;
while *workload is not balanced* **do**
  for *processor* $\in$ *processors* $-$ *list* **do**
    nullify the workload of each processor: $W_p = 0$;
  end
  for *block* $\in$ *blocks* $-$ *list* **do**
    assign block to processor-with-minimum-workload (p_{mw});
    update processor workload: $W_{p_{mw}} = W_{p_{mw}} + W_{block}$;
  end
  for *processor* $\in$ *processors* $-$ *list* **do**
    compute the workload ratio: $W_{pr} = \frac{W_p}{\overline{W_p}}$;
  end
  if *all* W_{pr} $<$ *user-specified value* **then**
    exit main loop;
  else
    apply blocks-split algorithm;
    recompute descending-sorted blocks list with the new (split) blocks;
  end
end

Algorithm 1: Load balancing algorithm

It is worth noting that the user-specified threshold (which defines the splitting strategy) has been revealed fundamental to limit the final number of generated blocks.

At present, the *Load Balance* pre-processor has been proven to be effective and reliable. However, it is planned to extend the blocks-split and blocks-assignment algorithms in order to improve the parallelization quality. In particular, the communications' overhead will be taken into account in future developments: an improved algorithm should be able to quantify the increase of communications' overhead due to a block split in order to perform a multi-objectives optimization (optimizing the workload balance and minimizing the communications' overhead). Moreover, the improved assignment algorithm should be able to take into account the "physical proximity" of blocks in order to assign interacting blocks to the same processor.

5.2. Implementation

Load Balance is written in Fortran (standard 2003 compliant). The Object Oriented Programming (OOP) paradigm has been partially used (through the module-features of modern Fortran standards): two main objects have been defined, namely the block and the processor objects. Each object has its own type-bound procedures in a way that mimics a complete class definition.

In listing 1 the definition of *Type_Block* object is reported. It is worth nothing that this object implements a triple-linked list: there are the pointers to previous and next elements in the *leaf* (blocks) list (*bpl* and *bnl*) and a pointer to the children blocks (*bs*) that can be dynamically created (or destroyed) by the blocks-split algorithm. This object defines the main data structure, the descending-ordered blocks list that is defined as type(Type_Block), pointer:: leaf=>null().

```
type, public :: Type_Block
  ! block indexes
  integer(I4P)::                a=0   ! Index of ancestor block.
  integer(I4P)::                l=0   ! Level of splitting.
  integer(I4P)::                ba=0  ! Absolute block index.
  integer(I4P), allocatable:: b(:)  ! Blocks map, history of splitting [1:Nl].
  integer(I4P)::                p=-1  ! Processor assigned to.
  ! block data
  integer(I4P)::                Ni=0,Nj=0,Nk=0          ! Number of cells along each directions.
  integer(I4P), allocatable:: pfi(:),pfj(:),pfk(:)  ! Prime factors of Ni, Nj and Nk.
  integer(I8P)::                Wb=0                    ! Work load of each block.
  ! children blocks
  logical::                     spt=.false.     ! Flag for checking if the block is split.
  integer(I4P)::                dir=0           ! Direction of split.
```

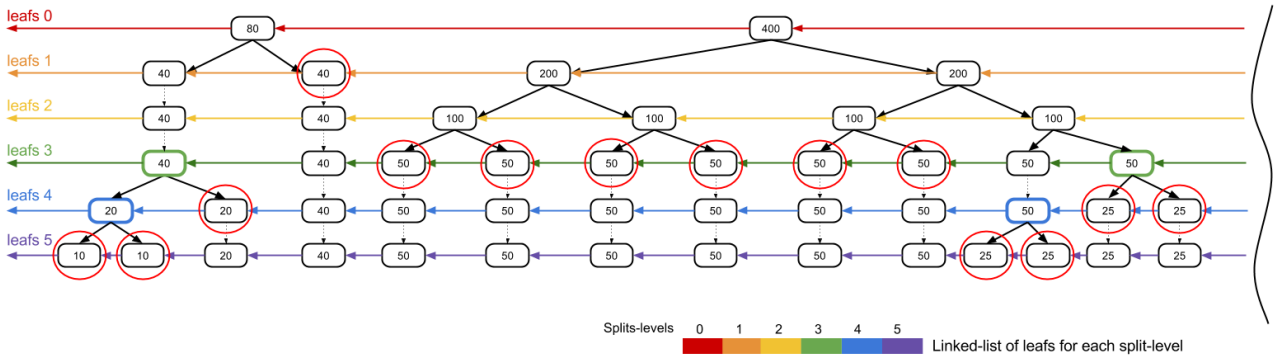

Fig. 1: Chronological representation of the tree data structure of blocks-split; the red circles indicate the final blocks

```
  integer(I4P)::                  Ns=0              ! Number of split (children) blocks.
  integer(I4P)::                  Nijk=0            ! Index of cell where split happens.
  type(Type_Block), pointer::     bs(:)=>null()     ! Split (children) blocks [1:Ns].
  ! leafs list
  type(Type_Block), pointer::     bnl=>null()       ! Next element in the leafs list.
  type(Type_Block), pointer::     bpl=>null()       ! Previous element in the leafs list.
  contains
    procedure:: init => init_block                                ! Initializing block data.
    procedure:: set => set_block                                  ! Setting block data.
    procedure:: rm_from_leaf => remove_block_from_leaf            ! Removing block from leafs list.
    procedure:: split => split_block                              ! Splitting block.
    procedure:: mgl_check => mgl_check_block                      ! Checking multigrid levels (¿ mgl).
    procedure:: get_Nbs => get_block_Nbs                          ! Computing the number of (split) sub-blocks.
    procedure:: get_lmax => get_block_lmax                        ! Computing the maximum level of splits.
    procedure:: print => print_block                              ! Printing information of block.
    procedure:: save => save_block                                ! Saving block splitting history.
endtype Type_Block
```

Listing 1: Type_Block definition

In listing 2 the definition of *Type_Proc* object is reported. The processor object is static (without any pointers) because it is assumed that the processors list is static and the processors are homogeneous.

```
type, public:: Type_Proc
  ! processor data
  real(R8P)::      Wpr=0._R8P          ! Work load ratio of the processor.
  integer(I8P)::   Wp=0                ! Work load of the processor.
  logical::        balanced=.false.    ! Flag for checking if the processor is balanced.
  integer(I4P)::   Nb=0                ! Number of blocks of the processor.
  contains
    procedure:: init => init_proc      ! Procedure for initializing processor data.
    procedure:: set => set_proc        ! Procedure for setting processor data.
endtype Type_Proc
```

Listing 2: Type_Proc definition

The main code uses these two base objects for building the blocks and processors lists and to handle them (mapping blocks over processors). As a matter of fact, *Load Balance* is strongly based on tree data structure (based on Type_Block pointers) in order to minimize memory requirements. In particular, blocks-split and blocks-assignment algorithms rely on a recursive approach where pointers of linked-lists are dynamically modified. It is worth noting that this pointer-based data structure allows very efficient memory management (data insertion, removal, relocation, etc.) avoiding the overhead of an explicit memory allocation/deallocation.

In figure 1 an example of a typical tree-data structure is reported. In this figure the blocks-split algorithm is applied six times to two blocks having a work load of 80 and 400. In this example it is assumed that the average work load is $\overline{W_p} = 60$, thus all blocks heavier than the average work load are split. All (split) blocks become lighter than the average work load after the third split (green linked-list of leafs in the figure 1). During the fourth and fifth splits the blocks-split algorithm is applied to the (first) lightest and heaviest blocks.

Although *Load Balance* is currently a pre-processor, its data structure can be potentially integrated directly into χ*navis* solver allowing dynamical load balancing at runtime.

5.3. Results

The developed code, *Load Balance*, has been proven to be effective and reliable. Several tests have been done on Fermi architecture. In particular, *Load Balance* has been able to achieve satisfactory balancing (below 5%)

with a total workload ranging from 10 to 80 millions of cells, with a total number of blocks ranging from 50 to 500 and with a number of (MPI) processors up to 256. It is worth noting that this enables the efficient usage of Fermi architecture. In fact, *Load Balance* has been able to successfully balance a production-sized numerical grid over 256 MPI processes that allows the use of 256*16 = 4096 cores. In particular, during the strong and weak scaling analysis *Load Balance* has been extensively used providing well balanced distribution as the satisfactory scaling performances prove.

6. Deployment of Parallel I/O Technique onto *χnavis* Solver

6.1. Deployment of MPI-I/O Functionality

Parallel I/O capability has been deployed on *χnavis* solver by exploiting the MPI-I/O functionality. MPI-I/O was first developed in 1994 and has been incorporated into MPI library since the release of MPI-2 standard in 1997. The MPI-I/O function is basically a collective operation: entire CPU ranks under the same MPI communicator are asked to access the same file, conduct the same number of read/write operations at the same time. Thus, the massive data input needs not being physically partitioned as long as the partitioning map (i.e., the information on the allocation of blocks to individual rank) is provided, while a little tweak is necessary if a non-equal number of blocks is assigned to each rank.

χnavis solver accesses 4 different sets of input data at the initiation of simulation: 2 data sets of (multi-grid) mesh files and overset connectivity information, 1 file with partitioning map, and 1 input parameter file. Also, the code finishes with producing 2 sets of outputs, one with the coordinate of the mesh points at each multi-grid level and another with the final solution. Out of these files, input parameter and partitioning map are fairly small in size that the current implementation of the sequential read and broadcast operation performs well. Other files are quite large in size since they contain the physical coordinates (mesh files), or the information provided by the overset pre-processor (i.e. connectivity information file), or the flow properties (solution file) over the entire mesh points. These files are prepared and produced on a per-rank basis: the pre-processor partitions the entire mesh and connectivity files to individual chunks and the post-processor merges individually-produced output into a single file. Therefore, the objective of implementing parallel I/O operation is to get rid of the extra cost on splitting from / merging to a single global file at pre- and post-processing stages, as well as potential gain on I/O performance during the simulation.

Two versions of parallel I/O implementations have been incorporated in *χnavis* solver. The first implementation calls the shared collective I/O routines such as *MPI_File_Read_Ordered* and *MPI_File_Write_Ordered*. The primary strength of using the shared file pointer is that it needs not take into account different numbers of allocated blocks to each rank and their dispersed location inside the data file. The operation with the shared file pointer is very intuitive since the I/O operation starts from the very beginning of a file without hopping the file pointer's location. The only difference from the serial file read is that participants are multiple MPI ranks. As is depicted in figure 2, I/O files are structured with 3 different fields: number of blocks, size of each block, and entities at each block. The number of blocks and the header part is accessed by the master rank. Then, the shared file pointer scans from the start of the second field and let the allocated rank to read/write the block size information. The same procedure applies in reading/writing the data set at each block. The shared file pointer implementation lets the dedicated rank do the actual I/O operation whereas other ranks conduct the NULL operation. For example, when reading the size of the 1^{st} block in the case of figure 2, the p^{th} rank performs the actual read/write operation while other ranks call the I/O operation of 0 byte data. Therefore, the I/O performance is ideally expected to be the same as the sequential file I/O operation over the entire data file.

The number of I/O calls could be reduced by applying the individual file pointer operation using normal *MPI_File_Read/Write* functions. By using the individual file pointer and effectively scheduling I/O calls, the NULL I/O calls which is observed at the shared file pointer implementation can be minimized. For example, p'^{th} rank can be assigned to read the information of the 2^{nd} block at the same time when the p^{th} rank reads the 1^{st} block in figure 2. The number of I/O calls will reduce to the maximum number of allocated block to individual MPI rank, instead of having to call the number of entire blocks in shared file pointer implementation. On the other hand, the implementation becomes more complicated because each rank shall compute the exact displacement size to move the file pointer for the next I/O operation.

The primary technical difficulty lies in the portability between Fortran's file format and MPI-I/O implementation. The *"unformatted"* binary format of the Fortran language adds the record size at the start and end of each record itself.Thus, these record fields should be disregarded when the Fortran unformatted file is read through MPI-I/O; the extra header should be explicitly written for writing Fortran-compatible file through MPI-I/O, which is even more tricky since each Fortran compiler has its own standard format for headers. Furthermore, the endianness problem is harder to resolve. Indeed, most MPI libraries except Intel MPI do not implement the standard mechanism to handle interoperable files (*"external32"* format). Therefore, we finally decided to manage the little/big endian conversion by means of an additional preprocessing tool.

6.2. Performance of Parallel I/O Operation

The performance of parallel I/O has been investigated through the weak and strong scaling benchmarks. The baseline mesh file contains 200 blocks whose total number of cells is 1.3 million. Since we apply 3-level multigrid

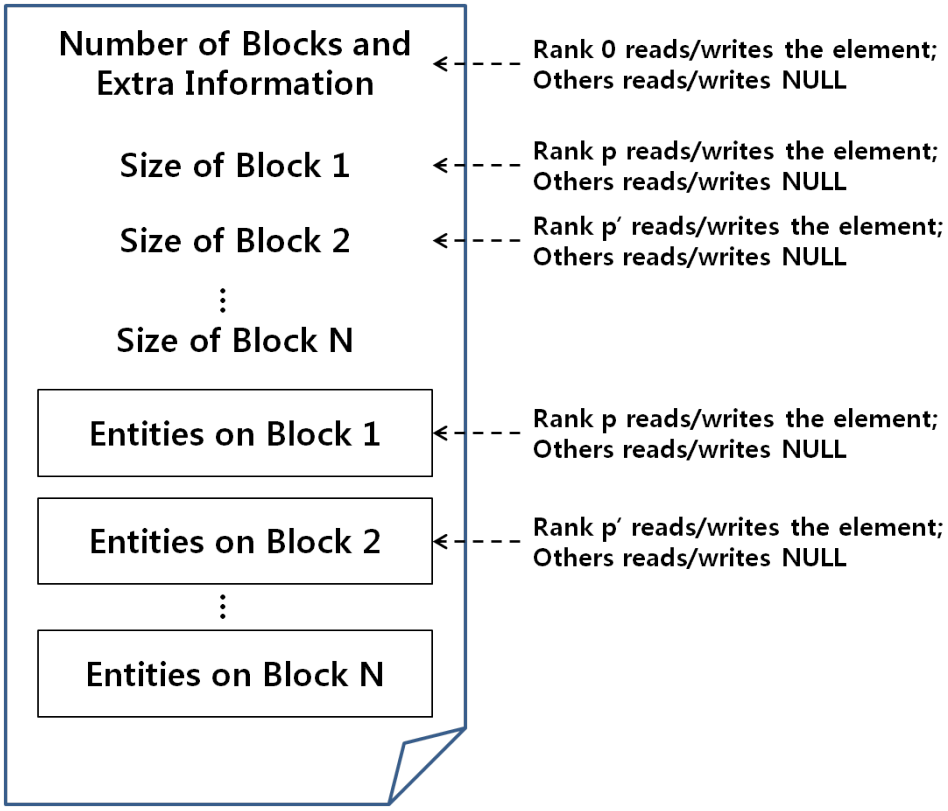

Fig. 2: Structure of data file and I/O access pattern in using shared file pointer

simulation, 3 grid files are accessed: 63, 14 and 4.2 megabytes from finest to coarsest level. For the weak scaling experiment, the same mesh system as the baseline mesh file is replicated and assigned to individual rank. The experiments have been performed on 8 and 64 MPI ranks. The strong scaling experiment been conducted on 8 CPU ranks after partitioning the global mesh into MPI ranks in balanced manner. All experiments have been performed on an x86-64 cluster which has 8 CPU cores per node and uses the GPFS (General Parallel File System) as the file system.

The result on the strong scaling benchmark is presented in table 5. With regard to the performance on reading, parallel I/O operations via individual and shared file pointers are comparable with the sequential I/O performance. In case of writing operation, parallel I/O operation overall takes more time due to the overhead of copying the solution data to an extra buffer array so as to minimize the number of I/O calls. Comparing the individual and the shared file pointer operations (*MPI_File_Read* VS *MPI_File_Read_Ordered*), shared file pointer operation usually takes more time since it calls more MPI I/O functions than the use of an individual file pointer.

Table 5: Strong Scaling Experiment of MPI-I/O. Results describe the elapsed time for I/O operations in seconds.

Number of Cores	Serial I/O	MPI_File_Read	MPI_File_Read_Ordered
1	2.52	1.41	2.39
8	0.84	1.38	3.43
Number of Cores	**Serial I/O**	**MPI_File_Write**	**MPI_File_Write_Ordered**
1	2.41	6.85	6.19
8	1.34	6.35	10.05

Weak scaling results at 1, 8, and 64 MPI ranks are presented in table 6. In the experiment with 8 MPI ranks, the individual file pointer operation takes roughly 2.2 times longer than the sequential I/O operation. Implementation with the shared file pointer even takes almost 4 times longer than the sequential operation. Though parallel I/O takes more time, it still seems to value since the extra cost of splitting and merging files at pre- and post-processing stage are not necessary any more. However, the cost with parallel I/O in case of 64 MPI ranks becomes excessively large compared with the serial I/O operation. We presume that it might be because input and output file sizes (the mesh file for the finest grid is 4 GB and the output data file is 7 GB) might exceed a certain limit on MPI-I/O implementation and/or MPI-I/O performs significantly worse as

the number of I/O ranks increases. It is indisputable that the current implementation performs strange at this experimental condition, which should be investigated in detail.

Table 6: Weak Scaling Experiment of MPI-I/O. Results describe the elapsed time for I/O operations in seconds.

Number of Cores	Serial I/O	MPI_File_Read	MPI_File_Read_Ordered
1	2.52	1.41	2.39
8	6.24	14.08	24.50
64	6.73	404.47	11013.78
Number of Cores	**Serial I/O**	**MPI_File_Write**	**MPI_File_Write_Ordered**
1	2.41	6.85	6.19
8	10.75	22.63	36.81
64	9.92	1654.88	5673.69

7. Scaling

The first strong scaling analysis has been performed based on a 80 million of grid points case with fixed grids. The test case is the simulation of the viscous free surface flow around a catamaran in a steady drift advancement. The drift angle is 5°, whereas the speed of advancement U corresponds to a Froude number ($Fr = U/\sqrt{(gL_{pp})}$, g and L_{pp} being the gravitational acceleration and the length between perpendicular of the catamaran) equal to 0.5. The flow field is characterized by the presence of several vortical structures, generated either along the main hulls (due to the mean cross flow) and on the free surface (due to the breaking and the following splash up of the bow eaves). An overview of the flow field is given in figure 3, where vortical structures are identified as negative values of the second greatest eigenvalues (denoted as λ_2) of the $\boldsymbol{S}^2 + \boldsymbol{\Omega}^2$ tensor, being $\boldsymbol{S}$ and $\boldsymbol{\Omega}$ the symmetric and the antisymmetric components of $\boldsymbol{\nabla u}$; i.e. $S_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i})$ and $\Omega_{ij} = \frac{1}{2}(u_{i,j} - u_{j,i})$.

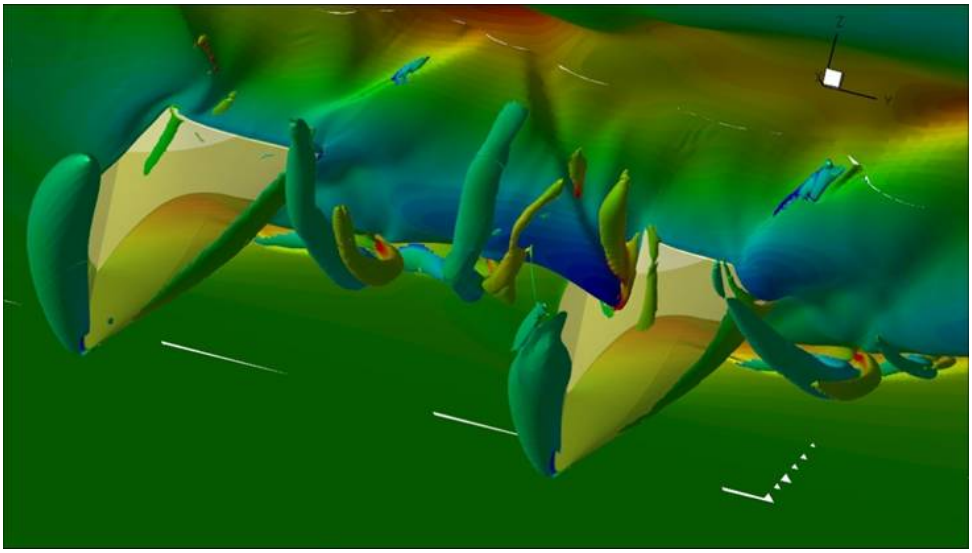

Fig. 3: Catamaran advancing in steady drift motion.

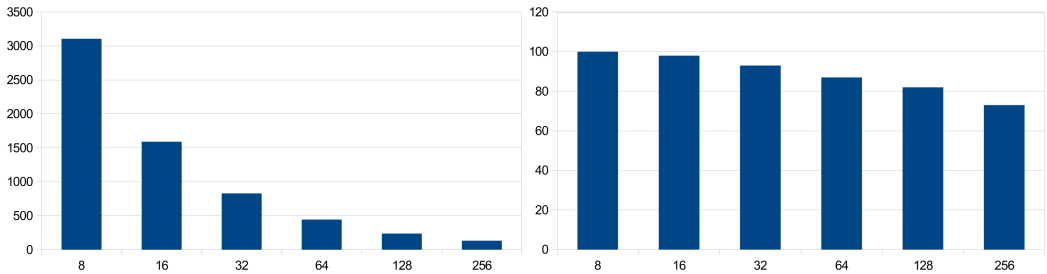

Fig. 4: Strong scaling, fixed grids. Left: elapsed times; right, efficiency.

Table 7: Strong scaling, fixed grids.

Nodes	#Cores	Time [s]	Speed-Up	Efficiency
8	128	3106	1.00	100%
16	256	1589	1.95	98%
32	512	829	3.74	93%
64	1024	444	6.99	87%
128	2048	237	13.10	82%
256	4096	133	23.35	73%

The analysis has been executed on the Fermi machine (Blue Gene/Q at CINECA). Since each node of Fermi has 16 cores we used 16 OpenMP threads (using 32 or 64 threads to exploit Fermi hardware threads gives no significant performance gain using $\chi navis$) while the MPI processes are distributed one per each Fermi node. As a test case, we simulated 10 temporal iterations and using 3 multi-grid levels. In this test case, moving grids algorithm has not been activated. The results of the analysis are provided in figure 4 and table 7; elapsed times are given in seconds. The efficiency is based on the 8 Nodes case. The overall efficiency is rather good, with a slight degradation when a large number of cores is used.

The scaling analysis has been also conducted considering a test case in which bodies in relative motion are present; in this case, the grid topology must be recomputed at each time step. The largest grid size simulated is composed by 11 million grid points. The test case refers to the numerical analysis of a submarine undergoing a prescribed oscillatory motion. The strong scaling results are provided in figure 5 and in table 8. The efficiency are now based on the 16 nodes case. The efficiency is still satisfactory, with values close to the fixed grid case.

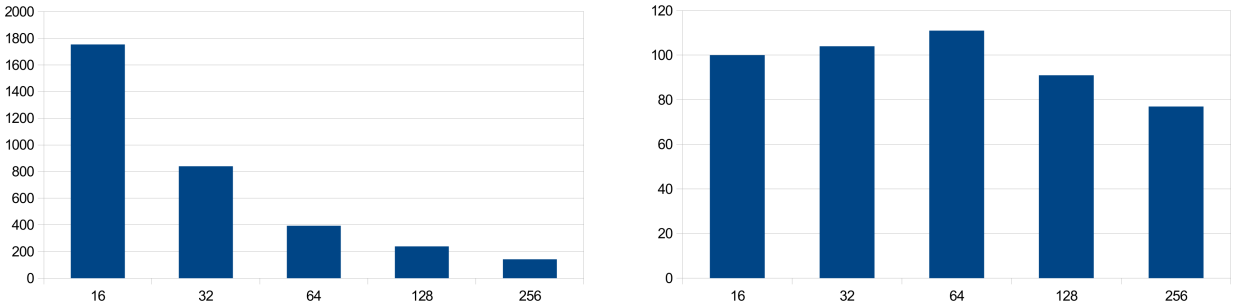

Fig. 5: Strong scaling, moving grids. Left: elapsed times; right, efficiency.

Table 8: Strong scaling, moving grids.

Nodes	#Cores	Time [s]	Speed-Up	Efficiency
16	256	1754	1.00	100%
32	512	840	2.09	104%
64	1024	393	4.46	111%
128	2048	239	7.34	91%
256	4096	142	12.35	77%

In order to evaluate the performance trend when enlarging the grids, a weak scaling analysis has been pursued. As test case, the oscillatory motion of a submarine has been considered. The results of the weak scaling analysis are very good, since the elapsed times are nearly constant as expected in the ideal case (see figure 6).

To have a complete overview of the code/machine performances, we also addressed a scalability analysis using Intel Sandy-Bridge architectures; this test can be of paramount importance when different TIER-0 architectures are taken into consideration. We tested the code with two different types of Sandy-Bridge nodes: the first type is a 2 eight-core Intel(R) Xeon(R) CPU E5-2658 2.10 GHz Dual socket, the second one is a 2 eight-core Intel(R) Xeon(R) CPU E5-2687W 3.10 GHz. It is highlighted a performance gain close to 10 for the 3.1 GHz nodes and 6.5 for the 2.1GHz

Table 9: Sandy-Bridge scaling comparison

Machine	Elapsed Time
Fermi Blue Gene/Q	264
Dual Sandy-bridge E5-2687W 3.1 GHz	26
Dual Sandy-Bridge CPU E5-2687W 2.1 GHz	39

Grid	Nodes	#Cores	Time [s]	Speed-Up	Efficiency
Coarse (0.17 mil)	2	32	264	1.00	100%
Medium (1.5 mil)	16	256	283	0.93	92%
Fine (11 mil)	128	2048	255	1.03	103%

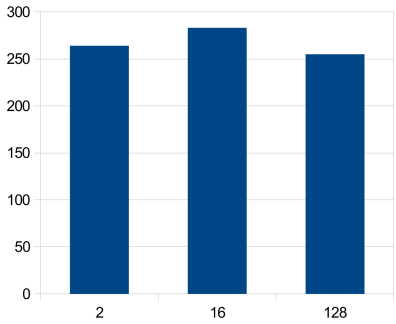

Fig. 6: Weak scaling, moving grids

comparing with Fermi cores. Results are reported in table 9.

8. Summary

The capabilities of the CFD solver *χnavis* have been extended to efficiently run on current massively parallel HPC facilities provided in the Tier-0 PRACE context.

In particular, the work involved several aspects of the operating sequence needed to perform a simulation. Namely:

- the memory re-factoring and the parallelization of the *overset* preprocessor, which is used to search for the donors/receivers map needed to handle the overlapping grids;
- the development and implementation of an integrated block splitting/load balancing preprocessors, which is used for an automatic distribution of the workload among the available processors;
- the development of a parallel I/O, which is crucial to accelerate the access to data files when large-scale numerical simulations are performed;
- measurement and assessment of strong and weak scalability of the CFD solver.

To summarize, all the previous points have been performed, results are very satisfactory. The code has been proven to scale efficiently (strong scaling) up to 4096 cores for both large and medium size grids, regardless the use of the moving grids algorithm. Weak scaling has been tested with satisfactory results up to a total of 2048 core. The algorithm developed for the automatic balancing of the work loads has been demonstrated to efficiently split and distribute the blocks among the available processor, within a prefixed unbalance tolerance (generally a maximum unbalance of 5% has been requested). The memory re-factoring of the *overset* algorithm allowed to generated grids and topology files for all the cases tested, regardless the size of the grid. The great improvement has been the possibility to generate input files for large size grid (order of hundred millions of cells) on normal size RAM machine (around 16Gb). The *overset* code has been also parallelized (hybrid MPI/OpenMP), resulting in a satisfactory speed up of the performances. Finally, the parallel I/O algorithm developed allowed the use of a single input and output files, avoiding the generation of input and output files for each processor, which can be unaffordable when a large number of processors are used. Unfortunately, the current implementation does not show any significant improvement in terms of I/O CPU time, some issues must be investigated and will be matter of near future activity.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement nos. RI-283493 and RI-312763. The work was primarily achieved using the PRACE Research Infrastructure resources at the FERMI (CINECA) supercomputer.

References

1. K.P. Apponsah. A Load-balancing Tool for Structured Multi-block CFD Applications Applied to a Parallel Newton-Krylov Algorithm. Master's thesis, University of Toronto, November 2012.
2. R. M. Beam and R. F. Warming. An Implicit Factored Scheme for the Compressible Navier-Stokes Equations. *AIAA Journal*, 16:393–402, 1978.
3. R. Broglia, A. Di Mascio, and G. Amati. A Parallel Unsteady RANS Code for the Numerical Simulations of Free Surface Flows. In *Proc. of* 2^{nd} *International Conference on Marine Research and Trasportation*, Ischia, Naples, Italy, 2007.
4. R. Broglia, A. Di Mascio, and R. Muscari. Numerical Study of Confined Water Effects on a Self–Propelled Submarine in Steady Manoeuvres. *Int. J. of Offshore and Polar Engineering*, 17:89–96, 2007.

5. A. Chorin. A Numerical Method for Solving Incompressible Viscous Flow Problems. *J. Comput. Phys.*, 2:12–26, 1967.
6. A. Di Mascio, R. Broglia, and B. Favini. *A Second Order Godunov–Type Scheme for Naval Hydrodynamics*, pages 253–261. Kluwer Academic/Plenum Publishers, 2001.
7. A. Di Mascio, R. Broglia, and R. Muscari. On the Application of the One–Phase Level Set Method for Naval Hydrodynamic Flows. *Computer and Fluids*, 36(5):868–886, 2007.
8. A. Di Mascio, R. Broglia, and R. Muscari. Prediction of hydrodynamic coefficients of ship hulls by high-order Godunov-type methods. *J. Marine Sci. Tech.*, 14:19–29, 2009.
9. A. Di Mascio, R. Muscari, and R. Broglia. An Overlapping Grids Approach for Moving Bodies Problems. In *Proc. of* 16^{th} *Int. Offshore and Polar Engineering Conference*, San Francisco, California (USA), 2006.
10. D. Durante, R. Broglia, R. Muscari, and A. Di Mascio. Numerical Simulations of a turning circle manoeuvre for a fully appended hull. In *Proc. of* 28^{th} *Symposium on Naval Hydrodynamics*, Pasadena, California, 2010.
11. B. Favini, R. Broglia, and A. Di Mascio. Multi–grid Acceleration of Second Order ENO Schemes from Low Subsonic to High Supersonic Flows. *Int. J. Num. Meth. Fluids*, 23:589–606, 1996.
12. Message Passing Interface Forum. Chapter 13: I/o. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.
13. S. K. Godunov. A Finite Difference Method for the Numerical Computation of Discontinuous Solutions of the Equations of Fluid Dynamics. *Mat. Sb.*, 47:271, 1959.
14. A. Harten, B. Engquist, S. Osher, and S. R. Chakravar thy. Uniformly High Order Accurate Essentially Non–Oscillatory Schemes. *J. Comput. Phys.*, 71:231–303, 1987.
15. Parallel HDF5. `http://www.hdfgroup.org/HDF5/PHDF5`.
16. A. Di Mascio, S. Zaghi, R. Muscari, R. Broglia, B. Favini, and A. Scaccia. On the Aerodynamic Heating of VEGA Launcher: Compressible Chimera Navier-Stokes Simulation with Complex Surfaces. In 7^{th} *European Aerothermodynamics Symposium*, Brugge, Belgium, 9-12 Maggio, 2011, 2011.
17. C. L. Merkle and M. Athavale. Time–Accurate Unsteady Incompressible Flow Algorithm Based on Artificially Compressibility. *AIAA paper*, 87–1137, 1987.
18. R. Muscari and A. Di Mascio. Simulation of the flow around complex hull geometries by an overlapping grid approach. In *Proc.* 5^{th} *Osaka Colloquium*, Osaka, Japan, 2005.
19. R. Muscari, M. Felli, and A. Di Mascio. Analysis of the flow past a fully appended hull with propellers by computational and experimental fluid dynamics. *Journal of Fluids Engineering*, 133(6), 2011.
20. Parallel netCDF: A Parallel I/O Library for NetCDF File Access. `http://trac.mcs.anl.gov/projects/parallel-netcdf`.
21. S. Osher and J. A. Sethian. Fronts Propagating with Curvature–Dependant Speed: Algorithms Based on Hamilton–Jacobi Formulations. *J. Comput. Phys.*, 79:12–40, 1988.
22. Roberto Muscari and Andrea Di Mascio and Roberto Verzicco. Modeling of vortex dynamics in the wake of a marine propeller. *caf*, 73(0):65 – 79, 2013.
23. P. R. Spalart and S. R. Allmaras. A One–Equation Turbulence Model for Aerodynamic Flows. *La Recherche Aérospatiale*, 1:5–21, 1994.
24. M. Sussman, P. Smekerda, and S. J. Osher. A Level Set Approach for Computing Solutions to Incompressible Two–Phase Flow. *J. Comput. Phys.*, 114:146–159, 1994.
25. B. Van Leer. Towards the ultimate conservative difference scheme V. A second-order sequel to Godunov's method. *J. Comput. Phys.*, 32:101–136, 1979.
26. S. Zaghi, Muscari R., and A. Di Mascio. Cnr-insean: Numerical simulations of wind turbines by means of dynamic overset grids. In *"Blind test 2" Workshop Calculations for two wind turbines in line*, Trondheim, Norway, 2012.
27. Stefano Zaghi, Riccardo Broglia, and Andrea Di Mascio. Analysis of the interference effects for high-speed catamarans by model tests and numerical simulations. *Ocean Engineering*, 38(1718):2110–2122, 2011.