Optimisation of the parallel performances of a CFD solver for emerging computational platforms

Francesco Gava

Normandie Université

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité Energétique

Préparée à l'INSA de Rouen Normandie

Optimisation des performances parallèles d'un solveur CFD pour les plateformes de calcul émergentes

présentée et soutenue par

Francesco Gava

Thèse soutenue publiquement le 4 février 2022 devant le jury composé de		
M. Pérache	Chargé de recherche et HDR, CEA	Rapporteur
E. Jeannot	Directeur de recherche, INRIA	Rapporteur
L. Giraud	Directeur de recherche, INRIA	Examinateur
G. Staffelbach	Chercheur senior, CERFACS	Examinateur
V. Moureau	Chargé de recherche CNRS, CORIA - CNRS	Encadrant de thèse
A. Berlemont	Directeur de recherche CNRS, CORIA - CNRS	Directeur de thèse

Thèse dirigée par, Vincent Moureau et Alain Berlemont, laboratoire CORIA (UMR 6614 CNRS)

Résumé

L'importance de la dynamique des fluides numérique dans le processus de conception industrielle a augmenté de façon spectaculaire au cours des deux dernières décennies. Cette évolution est principalement due aux progrès technologiques qui ont permis de disposer de clusters plus puissants et moins chers pour effectuer des simulations. Les CPU multi-cœurs et les GPGPU étant la nouvelle norme pour la construction de clusters, les plateformes deviennent de plus en plus variées. Cependant, la plupart des codes CFD ont été conçus pour les anciennes architectures et sont désormais incapables d'exploiter pleinement les machines les plus modernes. Cette thèse tente d'aborder certains des problèmes de performance que les codes CFD peuvent rencontrer sur les plateformes émergentes.
Les performances du solveur incompressible massivement parallèle YALES2 ont été analysées, afin d'identifier les points forts et les goulots d'étranglement dans les parties les plus importantes du solveur. Un accent particulier a été mis sur le solveur linéaire, où la plupart du temps de calcul est passé.
Un modèle de performance extrêmement simpliste a été obtenu pour toutes les parties fondamentales du solveur de Poisson. Afin de résoudre certains des problèmes mis en évidence par cette analyse, une nouvelle structure de données a été introduite dans le code, ce qui a permis de réduire le temps de calcul et de rendre le code lui-même plus flexible, en découplant deux concepts fondamentaux tels que les groupes d'éléments utilisés pour le cache-blocking et la grille de déflation du PCG.
En outre, pour essayer d'exploiter les processeurs modernes à mémoire partagée, deux modèles hybrides MPI+OpenMP différents ont été mis en œuvre.
Tout d'abord, une tentative a été faite avec un modèle OpenMP fine-grain, basé sur la parallélisation des boucles. Ce modèle semblait particulièrement adapté à la majeure partie de la structure de YALES2, mais présentait quelques difficultés qui le rendaient moins efficace dans certaines autres parties clés du solveur. Malgré les nombreuses tentatives d'optimisation, ce modèle n'a finalement pas permis d'améliorer les performances de l'implémentation MPI pure.
Ensuite, afin de surmonter les problèmes de l'implémentation fine-grain, un modèle OpenMP coarse-grain a été introduit. Principalement en raison du fait que les bibliothèques MPI sont séquentialisées en interne pour maintenir la thread-safety, cette mise en œuvre n'a pas non plus été fructueuse. Elle a cependant permis d'introduire certaines caractéristiques de modernisation du code, telles qu'une thread-safety complète et une API de communication abstraite.
En conclusion, ce travail montre les défis que représente l'adaptation d'un solveur de CFD incompressible low-Mach tel que YALES2 aux architectures les plus modernes actuellement disponibles. Malgré la faible efficacité de la plupart des solutions proposées, plusieurs questions intéressantes ont été soulevées, et de nombreuses perspectives d'optimisations futures ont été facilitées grâce au travail de fond sur la structure du code effectué au cours de cette thèse.

Abstract

The importance of Computational Fluid Dynamics in the industrial design process has increased dramatically in the last two decades. This is mainly due to the technological advancements that have brought on more powerful and cheaper clusters to run simulations on. With multi-core CPUs and GPGPUs being the new standard for cluster builds, platforms are becoming more and more varied. However, most CFD codes were designed for the older architectures and are now incapable of fully exploiting the most modern machines. This thesis tries to address some of the performance issues that CFD codes might encounter or the emerging platforms.
The performances of the massively parallel low-Mach incompressible solver YALES2 have been analysed, in order to identify strong points and bottlenecks in the most important parts of the solver. Particular focus was put on the linear solver, where most of the computational time is spent.
An extremely simplistic performance model has been obtained for all fundamental parts of the Poisson's solver.
In order to solve some of the issues uncovered by the performance analysis, a new data structure has been introduced into the code, which has allowed to reduce the computational time and make the code structure itself more flexible, decoupling two fundamental concepts such as the groups of elements used for the cache blocking and the Poisson's PCG deflation grid.
Furthermore, to try and exploit modern shared memory processors, two different hybrid MPI+OpenMP models have been implemented.
First, an attempt was made with a fine-grain OpenMP model, based on loop parallelisation. This model seemed particularly adapted for most of the YALES2 structure, but presented some challenges that rendered it less effective in some other key parts of the solver. In spite of the many optimisation attempts, in the end this model did not bring any improvement to the performance of the pure MPI implementation.
Then, in order to overcome the issues of the fine-grain implementation, a coarse-grain OpenMP model has been introduced. Mainly due to the fact that MPI libraries are internally sequentialised to maintain thread-safety, this implementation was also unfruitful. It has however allowed to introduce some modernising features to the code, such as a full thread-safety and an abstracted communication API.
In conclusion, this works shows the challenges to adapt a low-Mach incompressible CFD solver such as YALES2 to the most modern architectures currently available. Despite the poor effectiveness of most of the solution proposed, several interesting issues have been raised, and many perspectives on future optimisations have been made easier thanks to the ground work laid down during this thesis on the code structure.

Contents

Chapter 1

HPC for CFD

The last two decades have seen an exponential increase in the importance of CFD (Computational Fluid Dynamics) simulations in the process of industrial design. This is due to the enormous technological advances that happened in both the software and hardware side. On one hand, CFD software and numerical methods have become more reliable and precise, on the other, the computational power has increased massively, allowing for more complex and refined geometries to be simulated in reasonable time. In particular, architectures have become massively parallel, i.e. composed of hundreds of thousands of computational cores communicating with each other, and nowadays we are entering the so called exascale era [1], which means that machines are now able to execute a number of operations per second of the order of 10^{18} (exaFLOPs). Computational power can not grow indefinitely at this pace, as computing chips are hitting technological barriers and energy efficiency is becoming an issue [2]. Processors have evolved taking into consideration all these aspects, and multicore chips are now the norm in modern HPC (High Performance Computing) clusters [3, 4], together with GPGPUs (General Purpose Graphic Processing Unit) [5, 6]. Consequently, software need to constantly adapt to the evolving hardware if the full computation capability of such machines is to be exploited.
The objective of this work is to analyse the issues and bottlenecks that prevent a CFD code to scale efficiently, i.e. to maintain the same level of performance when running on a large number of cores, and to propose solutions to the overcome such impediments.
This first chapter gives an overview of CFD simulation and HPC systems, explaining the challenges and limits of CFD software on such platforms, concluding with a more detailed description of the motivations of this work, and the layout of this document.

1.1 Navier-Stokes Equations

The motion of newtonian fluids is described by the Navier-Stokes equations. They are a system of partial-derivative equations, derived from the laws of mass, momentum and energy conservation which link together the density, velocity, pressure and energy field. Since the number of unknowns is greater than the number of equations, one supplementary equation is needed in order to find a solution to this system. A supplementary equation is supplied by the thermodynamic state relation for the internal energy. This however introduces a new variable, the temperature. Closure

is finally provided by the equation of state, which creates a relation between the temperature, the pressure and the density.

1.1.1 Continuity equation

The continuity equation creates a relationship between the density and the velocity through the mass flow rate across the surface $\partial\Omega$ of a fixed (Eulerian) control volume Ω. The variation of mass inside the control volume is equal to the mass flow that crosses its boundary, as expressed by Equation 1.1, where ρ is the density of the fluid, $\boldsymbol{u}$ the velocity vector and $\bar{\boldsymbol{n}}$ is the outwards-directed unit normal vector to the surface $\partial\Omega$.

$$\int_{\Omega} \frac{\partial \rho}{\partial t} d\Omega = -\int_{\partial\Omega} (\rho \boldsymbol{u}) \cdot \bar{\boldsymbol{n}} dA \,. \tag{1.1}$$

Equation 1.2 is obtained applying the Gauss theorem to the right-hand side of Equation 1.1

$$\int_{\Omega} \left(\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) \right) d\Omega = 0 \,. \tag{1.2}$$

Since Equation 1.2 is valid for any control volume Ω the local form of the continuity equation can be written as in Equation 1.3.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \,. \tag{1.3}$$

1.1.2 Momentum conservation equation

The momentum equation is obtained from the second of Newton's laws, which states that the variation of momentum of a body is equal to the sum of the forces acting on it. For a fluid in a Eulerian control volume Ω those forces are induced by the gravity field $\boldsymbol{g}$, the pressure field P and the shear stress tensor $\tau = \mu \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T \right)$ of the fluid, with μ being its dynamic viscosity. The momentum of a body can be expressed as in Equation 1.4.

$$\boldsymbol{m} = \int_{\Omega} \rho \boldsymbol{u} d\Omega \,. \tag{1.4}$$

Applying again the Gauss theorem, and considering all the above mentioned effects on the control volume, Equation 1.5 is obtained.

$$\int_{\Omega} \left(\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) \right) d\Omega = \int_{\Omega} \left(\rho \boldsymbol{g} - \nabla P + \nabla \cdot \tau \right) d\Omega \,. \tag{1.5}$$

Again, Equation 1.5 is valid for any control volume Ω, consequently the local form can be obtained as in Equation 1.6.

$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) + \nabla P = \rho \boldsymbol{g} + \nabla \cdot \tau \,. \tag{1.6}$$

1.1.3 Energy conservation equation

The conservation of energy expressed by Equation 1.7 states that the temporal variation of the total specific energy e_T in the control volume Ω (left-hand side) is due to the convective fluxes of specific energy through the boundary $\partial\Omega$, the heat diffusion $\boldsymbol{q}$ and the work of the forces applied to the control volume (right-hand side).

$$\int_\Omega \frac{\partial \rho e_T}{\partial t} d\Omega = -\int_{\partial\Omega} \rho\,(e_T \boldsymbol{u}) + \boldsymbol{q}\cdot\bar{\boldsymbol{n}} dA - \int_{\partial\Omega} (P\boldsymbol{u})\cdot\bar{\boldsymbol{n}} dA + \int_{\partial\Omega} (\tau\boldsymbol{u})\cdot\bar{\boldsymbol{n}} dA + \int_\Omega \rho\boldsymbol{g}\cdot\boldsymbol{u} d\Omega\,. \tag{1.7}$$

Once again this is valid for any control volume Ω, which allows to write the differential form of the energy conservation as in Equation 1.8.

$$\frac{\partial \rho e_T}{\partial t} + \nabla\cdot(\boldsymbol{u}\,(\rho e_T + P)) = \nabla\cdot(\tau\boldsymbol{u}) - \nabla\cdot\boldsymbol{q} + \rho\boldsymbol{g}\cdot\boldsymbol{u}\,. \tag{1.8}$$

As stated above, in order to be able to solve the system, another equation is needed. Since Equation 1.8 is expressed in terms of specific energy, Equation 1.9 can be used, where ϵ is the specific internal energy.

$$e_T = \epsilon + \frac{1}{2}\boldsymbol{u}\cdot\boldsymbol{u}\,. \tag{1.9}$$

1.1.4 Equation of state

For a calorifically perfect gas, ϵ can be derived from the thermodynamic state Equation 1.10:

$$\epsilon = c_v T\,, \tag{1.10}$$

which relates the internal energy and the temperature T through the specific heat at constant volume c_v. As the temperature appears, a final equation is necessary for the complete closure of the system. This is achieved introducing equation of state, which, for a perfect gas, links density, pressure and temperature via the specific gas constant r as in Equation 1.11:

$$P = \rho r T\,. \tag{1.11}$$

1.2 Incompressible Navier-Stokes equations

Traditionally flows with a Mach number $M < 0.3$ are considered incompressible, which means that acoustic effects on the flow are negligible, hence they can be ignored. If the temperature is also constant, then the density does not vary as well. This means that only the velocity $\boldsymbol{u}$ and pressure P fields are to be determined. This can be done using only the continuity and momentum equation, which are further simplified by the hypothesis of constant density $\frac{\partial\rho}{\partial t} = 0$. Equation 1.3 becomes then the incompressibility equation 1.12.

$$\nabla\cdot\boldsymbol{u} = 0\,. \tag{1.12}$$

Equation 1.12 has implications on the momentum equation 1.6. First, the divergence of the shear stress τ, for a constant dynamic viscosity μ becomes:

$$\nabla \cdot \tau = \mu \left(\nabla^2 \boldsymbol{u} + \nabla \left(\nabla \cdot \boldsymbol{u} \right) \right) = \mu \nabla^2 \boldsymbol{u} \,. \tag{1.13}$$

The term $\nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{u})$ is also impacted:

$$\nabla \cdot (\boldsymbol{u} \otimes \boldsymbol{u}) = (\boldsymbol{u} \cdot \nabla + \nabla \cdot \boldsymbol{u}) \, \boldsymbol{u} = (\boldsymbol{u} \cdot \nabla) \, \boldsymbol{u} \,. \tag{1.14}$$

Finally, substituting equations 1.13 and 1.14 into Equation 1.6, simplifying by ρ and defining the constant cinematic viscosity $\nu = \frac{\mu}{\rho}$ the incompressible momentum equation becomes:

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla) \, \boldsymbol{u} + \frac{1}{\rho} \nabla P = \boldsymbol{g} + \nu \nabla^2 \boldsymbol{u} \,. \tag{1.15}$$

1.3 Numerical solution of the Navier-Stokes equations

In spite of their innumerable applications, there is still no proof that a solution for the Navier-Stokes equations in the 3D space always exist, and whether these solutions are smooth if they exist. Instead of an analytical solution, a numerical approximation is then sought in order to study the characteristics of a flow under specific conditions.

The first step of a numerical simulation is to define the physical space in which the phenomenon is to be studied. Then, such physical volume is discretised in several sufficiently small parts, called cells or elements. This process is called meshing, as the ensemble of interconnected points that form the cells are called *mesh*. In general, the finer the mesh, i.e. the smaller the elements, the more precise the solution will be. However this comes at a cost in terms of resolution time, as the latter increases with the number of cells in the mesh. The following step is the definition of the boundary and initial conditions, i.e. respectively the values of the physical properties on the boundaries of the domain and at their initial state in the entire volume. The good definition of these properties is paramount as the same problem will have a different solution for different boundary and initial conditions.

Another important aspect to take into account for a CFD simulation is that turbulent phenomena span multiple length scales. Different numerical approaches can be defined in function of how they take into account the different turbulence scales, the main one being DNS, LES and RANS. Direct numerical simulations (DNS) solve all turbulent scales, which means that the domain discretisation must be fine enough to capture the smallest scales of turbulence. This makes the computational cost of DNS prohibitive, consequently their use is limited to low-Reynolds flows and simple geometries. The opposite approach, in which the entire turbulence spectrum is modelled is based on the Reynolds Averaged Navier-Stokes (RANS) paradigm. The reduced numerical cost of RANS simulations comes at the detriment of the solution precision, since the only information that can be obtained by such simulation is the average velocity field and the modelling of the turbulent phenomena is extremely

complex. An intermediate approach between RANS and DNS is represented by Large-Eddy Simulations (LES). Based on Kolgomorov's turbulence cascade theory, LES aims at improving the precision obtained by RANS simulation while avoiding the extreme costs of DNS. This is done by defining a threshold length scale above which the turbulence is resolved and below which it is modelled, consequently transient high-Reynolds number flow can be simulated with a coarser discretisation with respect to DNS.
For the LES approach, the separation between resolved and modelled scales is obtained via a filter for the conserved quantities:

$$\overline{\phi}(\mathbf{x},t) = \left[G_\Delta \star \phi\right](\mathbf{x},t) = \int_\Omega \phi(\mathbf{y},t)\; G_\Delta(\mathbf{x}-\mathbf{y})\; d^3\mathbf{y}\,, \tag{1.16}$$

where G_Δ is the filter kernel of size Δ. The DNS approach can be se as the asymptotic case of an LES with a filter of size zero. In other terms, if the mesh is fine enough, a Large-Eddy Simulation can be equivalent to a DNS.

1.3.1 Projection Method

A modified version of the projection method first proposed by [7] and improved by [8] and [9] can be used for the time advancement of the incompressible Navier-Stokes equation. This method, often used in the simulation of incompressible flows, relies on the Helmoltz-Hodge decomposition theorem, that states that a vector field smooth enough can be expressed as the sum of an irrotational, a solenoidal and a harmonic field. For the velocity field $\boldsymbol{u}$ it means that:

$$\boldsymbol{u} = \boldsymbol{u}_i + \boldsymbol{u}_s + \boldsymbol{u}_h\,, \tag{1.17}$$

where:

$\boldsymbol{u}_i$ is the irrotational component of the velocity field, which verifies $\nabla \times \boldsymbol{u}_i = 0$. This is associated to the net flux through the infinitesimal control volume surface.

$\boldsymbol{u}_s$ is the solenoidal component of the velocity fields, which is such that $\nabla \cdot \boldsymbol{u}_s = 0$.

$\boldsymbol{u}_h$ is the harmonic component of the velocity fields, which satisfies $\Delta \boldsymbol{u}_h = 0$. This last component is often omitted from the analysis as it depends solely on the velocity field boundary conditions and it is null for an infinite or periodic domain.

$\boldsymbol{u}_i$ can be defined as the gradient of a scalar field ϕ as $\boldsymbol{u}_i = \nabla\phi$. Applying the divergence theorem to Equation 1.17 gives:

$$\nabla \cdot \boldsymbol{u} = \nabla \cdot \boldsymbol{u}_i = \nabla^2\phi\,. \tag{1.18}$$

Equation 1.18 shows that the scalar field ϕ can be determined solving a Poisson equation from $\boldsymbol{u}$ in order to compute the irrotational part of the velocity field $\boldsymbol{u}_i$. This method allows to solve the Navier-Stokes equations in three steps:

1. Prediction step:
   A first estimate of the velocity field $\boldsymbol{u}^{n+1}$ from $\boldsymbol{u}^n$ can be obtained rewriting the momentum equation without the pressure gradient, which contributes only to the irrotational velocity field.

$$\frac{\boldsymbol{u}^\star - \boldsymbol{u}^n}{\Delta t} = \nu \nabla^2 \boldsymbol{u}^n - (\boldsymbol{u}^n \cdot \nabla)\boldsymbol{u}^n \,. \tag{1.19}$$

   In order to have a more precise numerical method the contribution of the known pressure gradient $\nabla P^{n-\frac{1}{2}}$ is added. Since this pressure gradient is not expected to be much different from the one to be determined, it allows a better estimation of the velocity $\boldsymbol{u}^\star$ reducing the errors due to the time step splitting. Equation 1.19 becomes:

$$\frac{\hat{\boldsymbol{u}} - \boldsymbol{u}^n}{\Delta t} = \nu \nabla^2 \boldsymbol{u}^n - (\boldsymbol{u}^n \cdot \nabla)\boldsymbol{u}^n - \frac{1}{\rho}\nabla P^{n-\frac{1}{2}} \,. \tag{1.20}$$

   $\boldsymbol{u}^\star$ is then obtained subtracting the pressure gradient contribution:

$$\frac{\boldsymbol{u}^\star - \hat{\boldsymbol{u}}}{\Delta t} = \frac{1}{\rho}\nabla P^{n-\frac{1}{2}} \,. \tag{1.21}$$

   This approach is only useful for multi-step time advancement methods such as fourth order Runge-Kutta [10] or TFV4A [11] and it is absolutely unnecessary for explicit Euler methods.

2. Correction step:
   The velocity is then corrected with the new pressure gradient as:

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^\star}{\Delta t} = -\frac{1}{\rho}\nabla P^{n+\frac{1}{2}} \,. \tag{1.22}$$

   In Equation 1.22 the pressure gradient at the time step $n + \frac{1}{2}$ is of course unknown, but applying the divergence operator to this equation, and imposing the incompressibility constraint $\nabla \cdot \boldsymbol{u}^{n+1} = 0$ one obtains:

$$\nabla^2 P^{n+\frac{1}{2}} = \frac{\rho}{\Delta t}\nabla \cdot \boldsymbol{u}^\star \,, \tag{1.23}$$

   which is a Poisson equation for the pressure P. Finally the velocity field $\boldsymbol{u}^{n+1}$ can be obtained with:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^\star - \frac{\Delta t}{\rho}\nabla P^{n+\frac{1}{2}} \,. \tag{1.24}$$

1.3.2 Solution of the Poisson equation for the pressure

In subsection 1.3.1 it was shown how, thanks to the projection method, the solution of the incompressible Navier-Stokes equations can be split in few steps. This method produced Equation 1.23, a Poisson equation for pressure. The solution of this equation is the most important step in the entire procedure, both numerically

and in terms of computational cost. Equation 1.23 has to be solved for each control volume of the discretised domain. The pressure field is obtained as the solution of the resulting system of equations, one for each control volume. This system of equations can be written in the classical form of Equation 1.25, where A is the matrix representing the Laplacian operator ∇^2, x and b are respectively the vectors of the unknown values of the pressure P and the known term $\frac{\rho}{\Delta t}\nabla \cdot \boldsymbol{u}^\star$ of each control volume.

$$Ax = b\,. \tag{1.25}$$

The efficiency of the method used to find the solution of Equation 1.25 depends on the property of the matrix A. A is a square matrix whose number of rows and columns correspond to the number of control volumes in the domains. For massive CFD simulations, direct method such as the Gauss-elimination can not be used as the size of A can exceed the billion of elements. A is a definite symmetric positive matrix, in the considered case of a proper discretisation of the Laplacian operator and for constant Neumann boundary conditions, however it is non-strictly diagonally dominant. Finally even if A is sparse, its inverse is not, hence it would require large amount of memory for its storage. Consequently Krylov methods, such as the Conjugate Gradient (CG), are then generally preferred to other iterative methods such as Gauss-Seidel and Jacobi to solve this kind of system for CFD simulations. Solving Equation 1.25 with A a definite symmetric positive matrix, is equivalent to minimising a functional $f(x)$ equal to:

$$f(x) = \frac{1}{2}x^T Ax - x^T b \quad \text{with} \quad \nabla f(x) = Ax - b\,. \tag{1.26}$$

Equation 1.25 is obtained when $\nabla f(x) = 0$. This can be rewritten as a minimisation problem for another functional f' as in Equation 1.27:

$$f'(x) = \frac{1}{2}(x - x_f)^T A(x - x_f)\,. \tag{1.27}$$

where $Ax_f = b$. The difference between f and f' is a constant equal to $x_f^T b/2$. Solving Equation 1.25 is therefore equivalent to finding the minimum of a multi-dimensional quadratic function with a considerable number of dimensions. The functional f' can be seen as the squared distance between x and x_f for the A-scalar product. To minimise f', N projection steps are required for an orthogonal basis of N vectors. For each step, x is projected onto the orthogonal basis obtained by the orthogonalisation of a Krylov subspace:

$$\mathcal{K}_N(A, b) = \text{span}\left\{b, Ab, A^2b, \ldots, A^{N-1}b\right\}. \tag{1.28}$$

The Conjugate Gradient method works as in Algorithm 1, where $-\alpha_k p_k$ is the projection of x_k onto the current basis vector p_k and which is removed from x_k and ϵ is a convergence threshold for the residual r. This method also guarantees that all p_k vectors are A-orthogonal to each other.

Algorithm 1: Conjugate Gradient method

$r_0 = b - Ax_0$, $p_0 = r_0$, $k = 0$;
$\rho_0 = r_0^T r_0$;
$\sigma_0 = p_0^T A p_0$;
while *not done* **do**
  $\alpha_k = \frac{\rho_k}{\sigma_k}$;
  $x_{k+1} = x_k + \alpha_k p_k$;
  $r_{k+1} = r_k - \alpha_k A p_k$;
  if $\|r_{k+1}\| < \epsilon$ **then**
    exit the loop
  end
  $\rho_{k+1} = r_{k+1}^T r_{k+1}$;
  $\beta_k = \frac{\rho_{k+1}}{\rho_k}$;
  $p_{k+1} = r_{k+1} + \beta_k p_k$;
  $\sigma_{k+1} = p_{k+1}^T A p_{k+1}$;
  $k = k + 1$;
end
The result is x_{k+1}.

The inequality 1.29 relates the A-norm of the error at a step k to the initial A-distance to the solution and gives an estimation of the convergence of the Conjugate Gradient method.

$$\|x_f - x_k\|_A \leqslant 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x_f - x_0\|_A \,, \tag{1.29}$$

where $\kappa(A)$ is the condition number of the matrix. From a computational point of view Algorithm 1 requires only a single matrix-vector product and a few scalar products per step, however the number of steps is bounded by the possibly extremely large number of lines in the A matrix. The CG method relies on a recurrence which performs the orthogonalisation on the fly in successive steps, which makes it extremely efficient but also prone to the accumulation of round-off error, which then can often require the re-initialisation of the method in order to reach convergence. The Conjugate-Gradient method is consequently not directly suited to solve the large linear system resulting from CFD problems. However the system can be preconditioned in order to accelerate the algorithm. Preconditioning simply consist in finding a matrix K for which $K^{-1}A$ has a better condition number than A and solving the problem $K^{-1}Ax = K^{-1}b$. In order to have some gain in the solution time the cost of computation for K^{-1} must be contained. A standard choice for the preconditioning matrix is the inverse of the diagonal of A: $K = \mathrm{diag}(A)$, also known as the Jacoby preconditioner [12]. However, the only interest of this preconditioning is the non-dimensionalisation of the system and an improvement on the round-off errors, as this choice for K does not change the condition number

of the system. The so-called Preconditioned Conjugate Gradient (PCG) method is detailed in Algorithm 2.

Algorithm 2: Preconditioned Conjugate Gradient method

$r_0 = b - Ax_0$, $p_0 = r_0$, $k = 0$;
$w_0 = K^{-1} r_0$;
$\rho_0 = r_0^T w_0$;
$\sigma_0 = p_0^T A p_0$;
while *not done* **do**
  $\alpha_k = \frac{\rho_k}{\sigma_k}$;
  $x_{k+1} = x_k + \alpha_k p_k$;
  $r_{k+1} = r_k - \alpha_k A p_k$;
  if $\|r_{k+1}\| < \epsilon$ **then**
    exit the loop
  end
  $w_{k+1} = K^{-1} r_{k+1}$;
  $\rho_{k+1} = r_{k+1}^T w_{k+1}$;
  $\beta_k = \frac{\rho_{k+1}}{\rho_k}$;
  $p_{k+1} = w_{k+1} + \beta_k p_k$;
  $\sigma_{k+1} = p_{k+1}^T A p_{k+1}$;
  $k = k + 1$;
end
The result is x_{k+1}.

1.3.3 The DPCG algorithm

As stated above, the main disadvantage of the CG method is the large number of iterations that can be necessary to reach convergence. Preconditioning the system can help reduce the number of iterations, however there is a trade-off between the gain in terms of number of iteration and the cost of computing the preconditioning matrix K^{-1}. Another solution, called deflation, is to project the system of Equation 1.25 on a sub-space $\mathbb{S}^m$ of $\mathbb{R}^n$. This is done by constructing a matrix W of size $n \times m$ with $m \ll n$ such as the columns of W form the sub-space $\mathbb{S}^m = \text{span}\{w_1, \ldots, w_m\}$. The residual r must be projected onto the sub-space orthogonal to $\mathbb{S}^m$, which means that its components along the vectors $\{w_i\}_i^m$ are removed. If $\hat{A} = W^T A W$ is the projection of A onto $\mathbb{S}^m$, then the vector of the projected residuals d is found by solving Equation 1.30.

$$\hat{A} d = W^T A r \,. \tag{1.30}$$

If the same projection is applied to a system with a preconditioning matrix K, Equation 1.31 is obtained.

$$\hat{A} d = W^T \left(A K^{-1} - I \right) r \,. \tag{1.31}$$

In [13] several different techniques of preconditioning and deflation for Conjugate Gradient methods are compared, and the method of Algorithm 3, here called Deflated Preconditioned Conjugate Gradient (DPCG), is evaluated as being the most stable and with superior performance of all the considered alternatives. It is important to notice that if A is positive semi-definite then $\hat{A}$ has the same characteristics, consequently all sub-systems like Equation 1.30 and Equation 1.31 can be solved with a CG or PCG method as well.

Algorithm 3: Deflated Preconditioned Conjugate Gradient method

$\hat{A} = W^T A W$;
Solve $\hat{A} d_{-1} = W^T b$;
$x_0 = W d_{-1}$;
$r_0 = b - A x_0$;
Solve $\hat{A} d_0 = W^T \left(A K^{-1} - I \right) r_0$;
$w_0 = K^{-1} r_0 - W d_0$;
$p_0 = w_0$ $\rho_0 = r_0^T w_0$;
$\sigma_0 = p_0^T A p_0$;
$k = 0$;
while *not done* **do**
  $\alpha_k = \frac{\rho_k}{\sigma_k}$;
  $x_{k+1} = x_k + \alpha_k p_k$;
  $r_{k+1} = r_k - \alpha_k A p_k$;
  if $\|r_{k+1}\| < \epsilon$ **then**
    exit the loop
  end
  Solve $\hat{A} d_{k+1} = W^T \left(A K^{-1} - I \right) r_{k+1}$;
  $w_{k+1} = K^{-1} r_{k+1} - W d_{k+1}$;
  $\rho_{k+1} = r_{k+1}^T w_{k+1}$;
  $\beta_k = \frac{\rho_{k+1}}{\rho_k}$;
  $p_{k+1} = w_{k+1} + \beta_k p_k$;
  $\sigma_{k+1} = p_{k+1}^T A p_{k+1}$;
  $k = k + 1$;
end
The result is x_{k+1}.

The increased complexity of the deflated algorithm is justified by its advantages in term of solution cost. Since by definition $m \ll n$, an iteration on the much smaller deflation system of m equations is significantly cheaper than one on the original system of size n. Performing several iterations to converge the deflation system in order to reduce the number of iterations on the larger one is consequently algebraically advantageous, as the total number of operations to be performed is finally smaller than the one required by the solution of the un-deflated system.

1.4 HPC challenges for CFD codes

Depending on the mesh size and the precision required, solving a CFD problem can require a significant amount of computational resources. Although simple benchmarks can be run on a laptop or a common desktop station, industrial problems must be computed on dedicated architectures called supercomputers in order to have reasonable return times. In order to run on High Performance Computing (HPC) clusters, the CFD computation must be split in several sub-problems, which are computed in parallel by the different processors on the cluster. This section aims at giving a summary presentation of the hardware structure of a typical HPC supercomputer, and to present the main challenges that CFD codes face on an HPC environment.

1.4.1 HPC hardware structure

There are many vendors who build and configure their own commercial HPC clusters, each with its own characteristics. However, in almost all cases, conceptually a modern supercomputer can be seen as a network of interconnected computing nodes. A node is an independent unit including one or more processing unit, memory and a network connection. This section will give a more detailed description of the structure of the most common clusters available at the present time, including those used during this work. The vocabulary used to define the various components of a cluster can sometimes be quite ambiguous and confusing. The aim of this section is also to give a clear definition of such components, in order to avoid such ambiguity, at least in the context of this work.
The most important part of each computing system is the *processor*, or Central Processing Unit (CPU). The design of processors is in continuous evolution, however the basic principle remains unchanged. The arithmetic and logical operations are performed by the Arithmetic Logic Unit (ALU), fetching its operand and storing its results in *registers*, while the control unit (CU) orchestrate the work of the ALU, registers and other components, fetching data and instructions from memory. The ensemble of ALU, registers and CU is also-called a *core*.
Memory is another fundamental aspect of any computing system. All data and instructions needed for a code to run must be loaded from the storage disk into the main memory, hence the bigger the data and the code the larger the memory requirement. The CPU has then to access data in memory in order to be able to perform any computation on it. Short memory access time is then primordial for an efficient execution. However, the faster the memory, the more expensive. A typical computing node counts several GB of random access memory (RAM), and it would be prohibitively expensive to build if the RAM main memory would have to be the fastest one available. To overcome this economical impasse without having to renounce performance completely, vendors provide their processing unit with a so-called cache memory. This is much faster than RAM, but it is installed in much smaller quantities, typically few MB. Cache memory is also organised hierarchically

in different levels (often 3), each larger but slower moving further away from the processing unit. Figure 1.1 gives a schematic representation of a typical memory

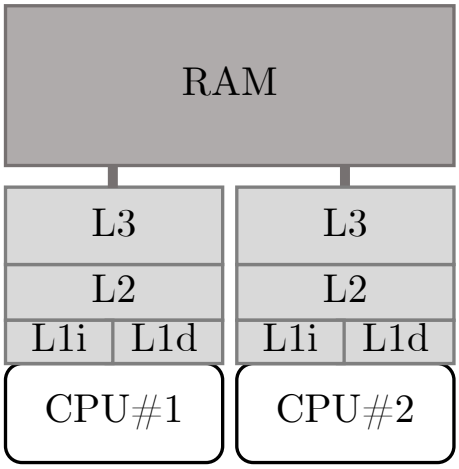


Figure 1.1: Schematic representation of two mono-core processors and their memory hierarchy

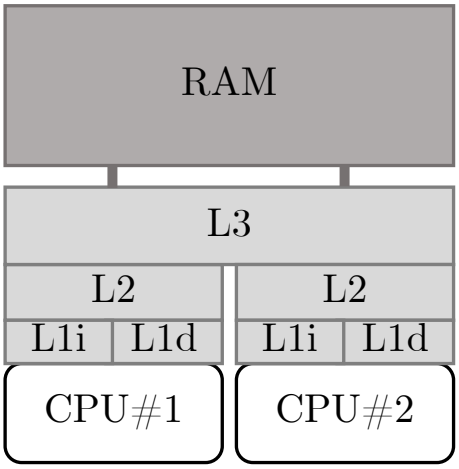


Figure 1.2: Schematic representation of a 2-cores processor with its memory hierarchy

hierarchy with 3 levels of cache (L1, L2, L3) and RAM memory, while some orders of magnitude of access time for each level of memory are given in Table 1.1. In most modern systems CPUs are provided as an integrated circuit chip, which also contains its cache memory. An extremely important milestone in the CPU evolution has been the introduction of *multi-core* processors, where more than one core is present on the same integrated circuit. As opposed to the classical *mono-core* ones which are completely independent from each other, the cores of these processors often have their own lowest levels of cache but share the outermost ones, usually L3. Figure 1.1 and Figure 1.2 give a schematic representation of the two kind of processors. L1 cache is often separated in L1i and L1d, respectively for instructions and data. Another term that is often used is *socket*, which is the set of hardware components that provide mechanical and electrical connection between the processor and the motherboard, however with the ever more popular installation of multi-core processors, it has de-facto become a synonym of processor, i.e. the ensemble of multiple cores and their cache memory, while processor is often confused with core. In the remaining of this work, all terms will be used as described above, hence processor and socket could be used interchangeably, and the former term should never be interpreted with the meaning of core.

The different nodes of a cluster are interconnected via a communication network. There are a few different technologies and standard for the network. Probably the most known is Ethernet, which is widely used for LAN and home application, however for HPC the most used are the Infiniband [14] and Intel Omni-Path [15] standards. Both aim to maximise the throughput, i.e. the rate of successful message delivery, while minimising the latency. While a commonly agreed definition of

Memory	L1	L2	L3	RAM	Disk
Capacity	O(KB)	O(100KB)	O(MB)	O(GB)	O(TB)
Latency [cycles]	O(1)	O(10)	O(100)	O(500)	O(10000)

Table 1.1: Typical order of magnitudes for memory hierarchy capacities and latencies per core [22, 23].

latency does not exist [16], in general terms it can be seen as the time needed for data to travel from a source to its destination [17].
With the advent of machine learning and other particularly favourable applications, clusters have started to include more and more GPU-nodes partitions. Historically, Graphics Processing Units have been used to speedup image processing and rendering software, which naturally offers great parallelism. They are based on a SIMD (single instruction-multiple data) paradigm, in which the same operation is applied to large vectors of data. Due to their potential for a wide amount of embarrassingly parallel applications General Purpose GPUs have become more popular and specific programming languages have been created, such as the NVIDIA proprietary CUDA [18] or the open source OpenACC [19] and OpenCL [20]. Another aspect that makes GPGPUs interesting for parallel computation it that, compared to CPUs, they are more energy efficient, and in fact most of the Green500 [21], the 500 most powerful efficient clusters in the world, are GPU based.
A compromise between CPUs and GPUs can be found in many-core architectures such as the Intel Many-Integrated Core (MIC) Xeon-Phi. With respect to standard processors, these accelerators are composed by a larger amount of cores sharing a high bandwidth 3D-stacked memory called MCDRAM, which allows the exploitation of shared memory programming techniques. This and the support of SIMD instructions makes them suitable also for those applications where GPUs generally perform best. This type of architecture is present on several of the Top500 clusters. However, Xeon-Phi technology have been discontinued by Intel in 2018.

1.5 Performance limits of CFD codes on modern clusters

The TOP500 [24] is a periodically updated ranking of the most powerful computing system in the world. Looking at the clusters belonging to this list over the years can give an idea of the trends of their characteristics. Figure 1.3, shows that there is a clear reduction of the growth rate of their performance starting from around 2010. Moore's law [25] states that the number of transistors in a CPU is bound to double every two years, and with it the computational power. This law, established in 1965, hold for several decades, however in the early 2010s physical limits of the materials, heating problems and power consumption impeded the processor frequencies to continue to rise, hence Moore's law is no longer valid, at least performance-wise. In order to continue to improve performances, CPUs manufacturer introduced logical

and physical multi-core processors. The principle of physical multi-core processors has been explained in the previous section, while logical multi-core rely on the principle that multiple *threads* of instructions can be executed independently on the same physical core. The introduction of this type of architecture is evident in Figure 1.4, where it's shown that the number of physical cores per socket on the top 500 constantly increases and becomes more varied. The continuous growth in size of supercomputer have proved somehow hard to be followed by software development. The theoretical throughput, measured in Floating Point Operations per Second (FLOPS) that can be obtained from such cluster has long surpassed the PFLOPS (Peta-FLOPS) and we are now on the verge of the so-called exascale era. Unfortunately it is very hard for a computer program to reach these theoretical values and to exploit the full capability of such clusters. This can be seen it the ever increasing gap between the LINPACK [26] and HPCG[27] benchmark performances. The first consists in the resolution of a dense system of linear equations and it is widely recognised as a performance indicator for a supercomputer, as it can almost reach the theoretical potential of the machine. The second is a high performance conjugate gradient, which is more representative of real HPC software, particularly CFD codes, and whose performance are far from the theoretical peak [28]. This type of software is based on the computations of operators which consist of simple sums and multiplications. This means that the computation on a chunk of data is extremely quick but frequent load-store memory operations are necessary to feed the ALU with new data to compute. Historically memory performance has always struggled to keep up with the advancement made by the CPUs, hence the struggle of CFD codes, which are limited by the memory access time rather than the CPU FLOPS. This behaviour is explained by the so-called roofline model [29]. The hierarchical structure of the cache memory can help alleviate this negative effect if the data structure of the code is able to exploit it.
Another limit might be represented by the parallel efficiency or speedup of the code. The speedup is defined as the ratio of the execution times of a same task on different computational resources. If the execution time for a computation on one process is T_1, then the theoretical time for running the same problem divided on n processes would be $T_n = T_1/n$. In CFD codes, although the computation mesh is split among different processes, the sub-domains are not independent from each other and data must be exchanged among the different processes. Communication in particular and other operations can not be executed entirely in parallel. The real speedup for the parallelised execution is then $S_n = T_1/T'_n < T_1/T_n$ and it depends on the ratio p between the parallelised and sequential parts of the code according to Amdahl's law [30] as in Equation 1.32.

$$S_n = \frac{1}{1 - p + \frac{p}{n}} \leq \frac{1}{1 - p}\,. \tag{1.32}$$

Amdahl's law is based on what is called strong scaling, i.e. using an increasing number of processes to parallelise a task of fixed size. Gustafson [31] argues that parallelism should aim at exploiting a larger amount of resources to execute bigger

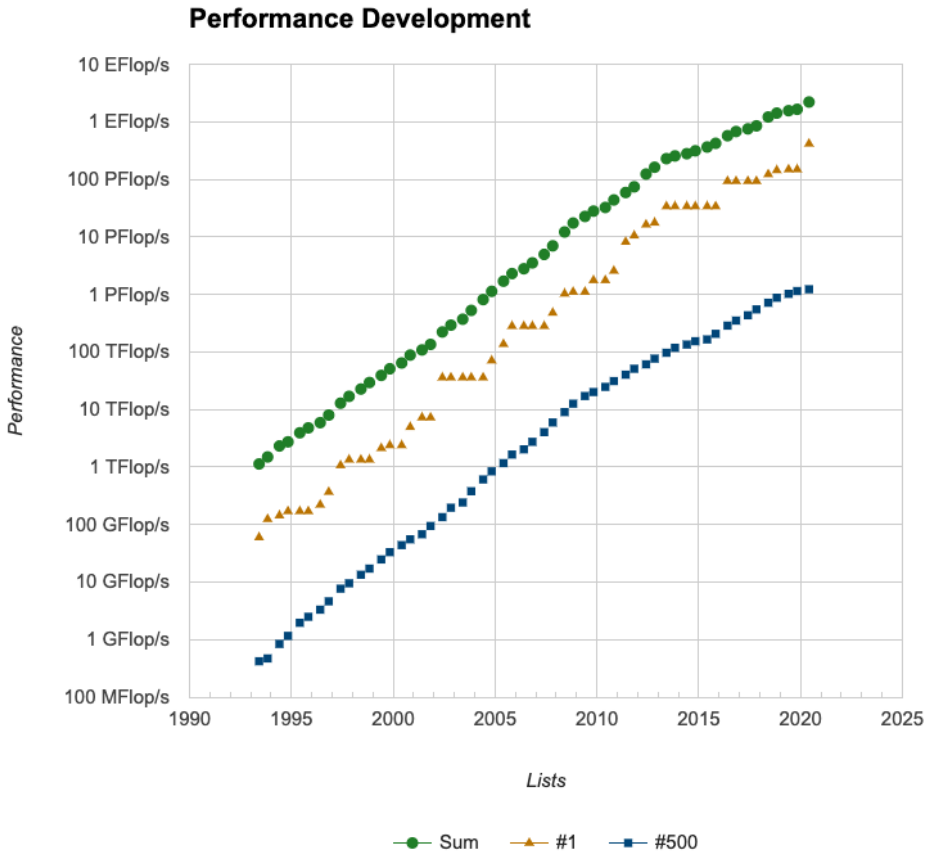


Figure 1.3: Performance of the top 500 supercomputers. Figure generated by [24].

tasks. This process of increasing the problem size with the resources used to compute, keeping a constant task size per process, is called weak scaling. Gustafson's law defines then the speedup as in Equation 1.33.

$$S_n = 1 - p + np\,. \tag{1.33}$$

Both strong and weak scaling approaches are used in CFD. A certain task might need to be executed on a larger number of processes in order to reduce the computation time, on the other hand, the need of ever increasing precision produce larger or denser domains which require more and more computational resources. The main sources of loss of parallel efficiency in CFD are to be sought in I/O operations and in inter-process communication. Reading and writing data to and from disk in parallel can be challenging, especially on files shared by multiple processes, however these operations usually do not represent a large portion of the execution time, or at

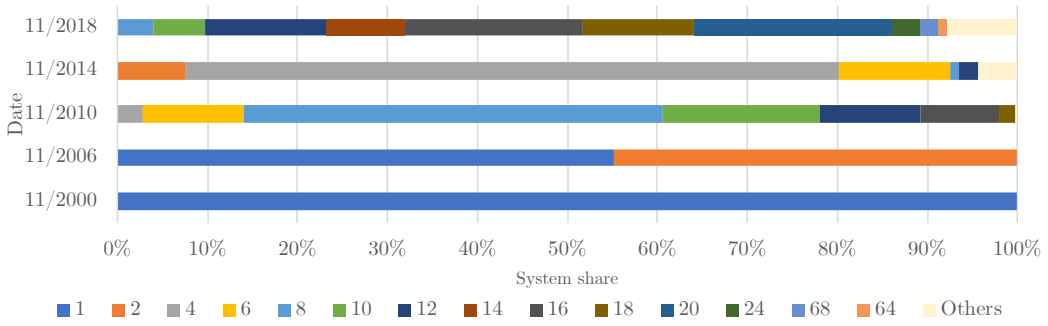


Figure 1.4: Evolution of the cores per socket in the Top 500

least are not bound to be executed many times. Conversely, communication among processes happens constantly during the code execution, moreover its cost increases with the number of processes participating in the communication. Reducing the negative effect of the data exchange among processes is then primordial for an efficient CFD code on the modern HPC architectures.

1.6 HPC communication paradigms

Parallel programming relies on the fact that multiple cores can execute instruction and work on data at the same time. The hardware components that allow data exchange and interaction among the various cores have been presented in Subsection 1.4.1, however the different processes need a set of instruction to be coded in the program in order to know when to perform such interactions. Several different paradigms and standards have been created with this scope. The main ones being MPI, OpenMP and PGAS. The different characteristics of each of these paradigms are detailed in this section.

1.6.1 MPI: Message Passing Interface

First introduced in 1994, is now the de-facto standard for distributed memory parallel programming. Now at its third version of the standard, and with the fourth in the process of discussion, MPI has different implementations such as MPICH [32], MVAPICH2 [33], OpenMPI [34], IntelMPI [35] and others.
It achieves parallelism through the SPMD (Single Program Multiple Data) technique, in which tasks run simultaneously the same program on multiple processors but on different data. Usually the data input and the different execution of the program is controlled by the task ID, called rank.
As its name suggest, communication between processes in the MPI paradigm is based on message exchange through the network.
An MPI program must begin with a call to `MPI_Init` or `MPI_Init_thread` for multithreaded programs. These functions initialise the MPI environment, in particular

they create the `MPI_COMM_WORLD` communicator. An MPI communicator is an entity that encapsulate a *group* of ordered process identifiers (ranks) and a *context* that allows the partitioning of the communication space [36]. Communications can only be performed inside a communicator. The `MPI_COMM_WORLD` communicator contains all the processes who have called `MPI_Init(_thread)`. To be compliant, an MPI program must end with `MPI_Finalize`, which ensures that all communications have terminated and destroys the MPI environment.
MPI communications can be characterised as *blocking* or *non-blocking*. Blocking communications require a synchronisation between the processes involved, while non-blocking ones simply make a request to start such communication and then a later verification is issued on that request to ensure that the communication has terminated.
Communication can also be divided in three categories: *two-sided*, *collective* and *one-sided*, respectively detailed in 1.6.1.1, 1.6.1.2 and 1.6.1.3.

1.6.1.1 Two-sided point-to-point communications

Two-sided are the most basic form of MPI communications. They involve only two processes, a sender and a receiver, exchanging a message. In its blocking version, the sender calls an `MPI_Send`, which needs the address of the data to be sent, the rank of the receiver, a tag identifying the communication and the communicator in which the operation is performed.
On the other hand, the receiver calls `MPI_Recv`, specifying the address of where the received data must be stored, the rank of the sender, a tag matching the one from the sender and the communicator. The receiver waits until the data is received from the sender. The main downside of this approach is that the processes have to synchronise and wait one another in order to be able to exchange the message, as represented in Figures 1.5 and 1.6. To avoid this problem and to allow for communication/computation overlap, non-blocking send and receive functions have been introduced.
In the non-blocking two-sided approach, the sender calls `MPI_ISend` which takes the same arguments as its blocking version plus a request ID, whose value is returned by the function and identifies the communication. In the same way, the receiver can call a `MPI_IRecv` which also returns a request ID. Both these operations do not perform the actual transfer, but returns immediately once the information about the communication is given to the network controller. Most MPI libraries are in fact implemented on top of the IBVerbs library [37], which provides an interface between the software and the Infiniband network controllers. Thank to this library, MPI is able to provide to the network controller enough information (source address, destination address, etc.) to let it take care of the data transfer through the network without the need of the CPU to be further involved. The CPU can then be used for computation while data is transferred. Both processes later verify that the transfer has been completed calling `MPI_Wait`, giving the respective request ID, as shown in Figure 1.8. Blocking and non-blocking send and receive functions can be mixed

together, to create a non-blocking sender with a blocking receiver or vice-versa, like in Figure 1.7. In this figures, the dashed lines represent the data transfer. For non-blocking communication the line is drawn with diagonal and vertical components to represent respectively the actual data transfer and the delay by the receiver in checking that the communication actually took place. The additional diagonal arrow in Figure 1.8 going from the receiver to the sender represents the notification issued to confirm that the data transfer was completed.

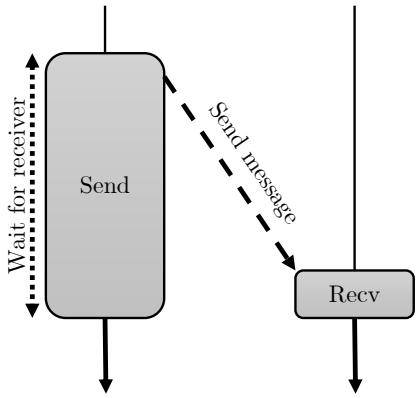


Figure 1.5: Blocking communication: late receiver

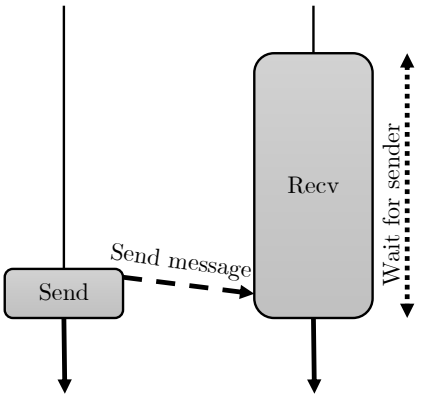


Figure 1.6: Blocking communication: late sender

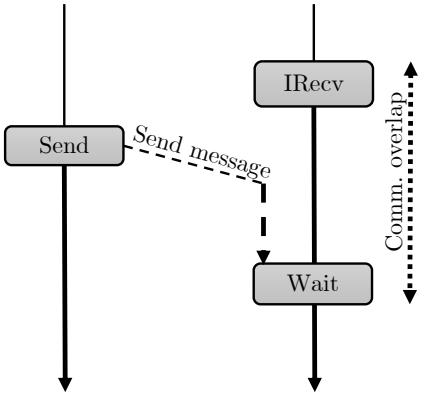


Figure 1.7: Blocking send and non-blocking receive

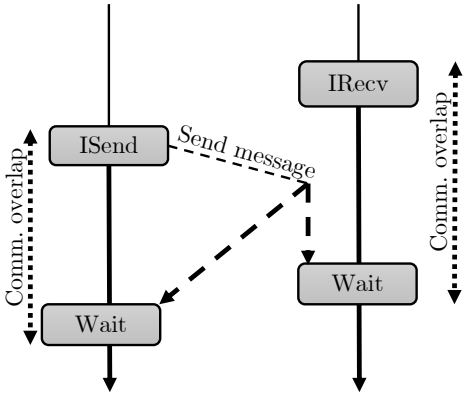


Figure 1.8: Full non-blocking communication

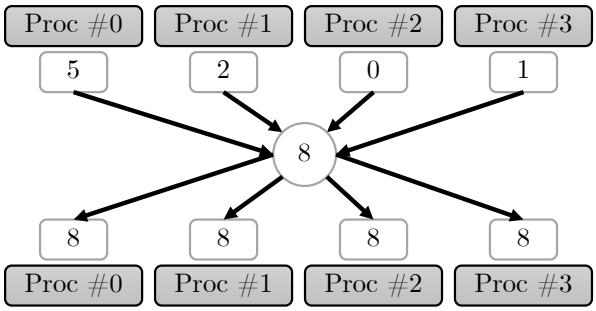


Figure 1.9: Schematic representation of an `MPI_Allreduce` (sum) on 4 processes. The actual implementation in the various MPI libraries might be different [38]

1.6.1.2 Collective communications

MPI collectives are communications that involve all processes in a communicator. The most important are:

`MPI_Allreduce` All processes perform a reduction operation (min, max, sum) on data from all processes, schematised in Figure 1.9.

`MPI_Reduce` One process performs a reduction operation on data from all processes.

`MPI_Bcast` One process broadcasts data to all the other processes.

`MPI_Gather` One process receives data from all the other processes and stores it in an array according to the sender's rank.

`MPI_Scatter` One process sends chunks of data to all the other processes according to their rank.

`MPI_Alltoall` All processes send and receive chunks of data from every other process according to their rank.

`MPI_Barrier` Synchronisation point. All processes wait until the last enters the barrier.

Although `MPI_Barrier` is the only function that guarantees it, in practice all collective communications are inherently a synchronisation point, as all processes must participate to the communication. This could be problematic for some algorithms which are ill balanced or asynchronous by nature. The MPI-3 specification has introduced a non-blocking version of all these functions (`MPI_IAllreduce`, ...) in order to try and avoid those synchronisation points.
Another disadvantage of collective communications is that they are not scalable, due to the fact that the more processes participate to a collective communication, the more time consuming the communication is. In order to improve the performance on modern processors, MPI libraries have implemented hardware-aware collective communications which take advantage of the shared memory to perform intra-socket communication [39].

1.6.1.3 One-sided communications

As seen in 1.6.1.1, MPI two-sided communications can negatively impact parallel performances creating some synchronisation points. Furthermore they inherently imply a memory copy of the message to some internal MPI buffers in order for the network to get access to it. MPI-2 norm tried to address these problems introducing one-sided communications. This model takes advantage of remote direct memory access (RDMA) capabilities of the network in order to transfer data from two different regions of memory. One requirement is that these regions must be pinned, which means that they have to be allocated through a special system call. This guarantees that these regions of memory are never paged out, i.e. that they are always mapped into physical memory. This is due to the fact that the network can only access pinned memory, hence the necessity of a copy in the traditional two-sided method. A scheme of the two mechanisms is shown in Figure 1.10.
A call to the collective function `MPI_Win_create` allows all processes to allocate a region of pinned memory called *window* that they could share with all the other processes. A call to `MPI_Win_free` deallocates the window.
MPI allows three types of operations, namely `MPI_Put` which transfers data from the sender's window to the receiver's, `MPI_Get` which retrieves data from a remote window and stores it in the local one, and `MPI_Accumulate` which allows for a reduction operation to be executed on the data of a remote window. All RMA operations are non blocking and must be executed inside a so-called *epoch*. The end of an epoch guarantees that the data transfer has been completed, however a call to `MPI_Win_flush` can be used to enforce the completion of an operation inside an epoch. MPI foresees two different modes for creating epochs on windows and are defined according to the role of the target, which can be active or passive. Active target means that the owner of the window on which the operation is performed opens and closes the epoch. This can be done in two ways: the first is via a call to `MPI_Win_fence` which enforces global synchronisation on all processes (the same way as a barrier) at the beginning and at the end of the epoch, the second through a Post-Start-Complete-Wait (PSCW) mechanism. With the PSCW mechanism the receiver must call `MPI_Win_post` and `MPI_Win_wait` to open and close an epoch on its window, while the sender must call respectively `MPI_Win_start` and `MPI_Win_complete` before and after the RMA operation. PSCW is similar to the fence mechanism but it involves only the subset of processes participating in the communication. The required synchronisation between sender and receiver however makes active target mechanism more similar to the two-sided rather than a true one-sided communication mechanism. Passive target communications are based on a lock mechanism. A process wanting to access data on a remote windows posts a request to acquire a lock on such window with `MPI_Win_lock`. Locks can be either shared or exclusive. A shared lock allows multiple processes to perform operations at the same time on the same window, however particular care must be taken by the user to guarantee that no concurrent accesses are made on the same memory location. Exclusive locks on the other hand guarantee that no other process can

access the window, until the lock is released with `MPI_Win_unlock`. These mechanisms have been improved by the MPI-3 standard, which have also introduced the possibility of allocating shared memory windows among processes on the same node. Data on this windows can be read and written with simple load and store operations, without the need of specific MPI calls, allowing a programming structure similar to a multithreaded environment, such as the one detailed in Subsection 1.6.2. In practice it is quite difficult to improve the performances of a two-sided communication kernel switching to MPI RMA operations. Window allocation is a global operation, hence it's not possible to change the window size dynamically without involving all processes. Furthermore, even if the standard requires that RMA operations must complete before the end of an epoch, implementations seem to enforce completion at the end of the epoch, even if the data transfer is ready before. Finally, passive target lock mechanism should allow a pure one-sided data transfer, however locks seem to be implemented in such a way that some kind of acknowledgement must be given by the remote process, de-facto creating a synchronisation with the local one. The MPI-4 standard, which is currently under evaluation, proposes some changes to the current RMA mechanism, in order to overcome all these drawbacks. The proposed model is based on a notification mechanism in which the sender notifies the receiver once the data transfer is complete. The receiver waits for the notification and if necessary sends back an acknowledgement of reception to the sender. This mechanism is already used in the GASPI [40] specification, whose approach is introduced in Subsection 1.6.3.

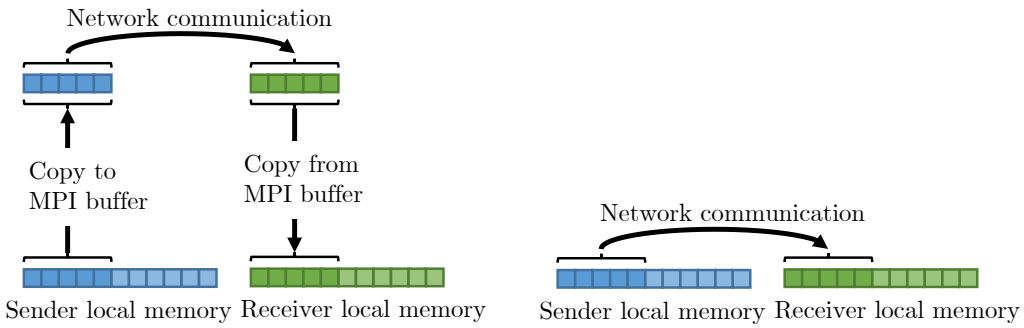


Figure 1.10: MPI two-sided (left) and one-sided (right) communication mechanisms

1.6.2 OpenMP

A thread is a segment of a process, i.e. a pipeline of instructions that are executed within a process. A process can then contain multiple threads, which can be executed in parallel if they are independent from one another. OpenMP is a shared memory parallelisation paradigm which enables the creation and management of *threads* using specific pragmas. It is based on a bulk-synchronous fork-join model in which threads are created when a parallel region is opened with the `OMP PARALLEL` pragma and they are destroyed when the parallel region is closed by

the `OMP END PARALLEL` pragma. Thread are much more lightweight than processes, and can be created and destroyed much quicker. Also they require much less memory. OpenMP is mainly used to parallelise loops with independent iterations. The `OMP PARALLEL DO` clause splits the loop in different chunks of iterations, scheduling them among the threads. Different heuristics can be used for the scheduling:

Static: The loop is divided into equal chunks, one for each thread. The load balancing among the different threads is based on the chunk size and not on the actual work inside each chunk.

Dynamic: The loop is divided into chunks of equal size, and they are dynamically distributed among the threads. The number of chunks is larger than the number of threads in order to allow for load balancing based on the actual work in each chunk, however the scheduling could imply an important overhead.

Guided: Similar to dynamic, but with the chunk size progressively decreasing in order to still have load balancing opportunities but with a reduced scheduling overhead.

Auto: The compiler automatically chooses between the previous scheduling mechanisms.

Runtime: The user specifies at runtime one of the first three mechanisms.

OpenMP has been widely used as its pragma approach allows for an easy and progressive parallelisation of the code, with minimum intervention. The scaling of the loop-based parallelisation however is clearly limited by Amdahl's law as the remaining parts of the code are executed sequentially. However, it still remains the best suited model for those applications composed of loop kernels representing the largest part of the code on a small number of cores. OpenMP also allows the possibility for a task-based worksharing construct. Inside a parallel region, a thread creates a pool of tasks which are then scheduled to all threads. Finally, another possibility is to manually take care of the scheduling inside the parallel region. This approach is much more complex than the first two as it does not allow for progressive parallelisation, and thread safety of the code must be guaranteed for each operation. It allows however a complete parallelisation of the code, reducing the limitations of the Amdahl's law of the previous approaches and gets rid of the scheduling overhead, which in certain cases might be quite important relatively to the work inside the loops or tasks.

1.6.3 PGAS

The Partitioned Global Access Space (PGAS) model consists in allocating a global memory space called segment which is partitioned among the different processes resources. Each process owns a local portion of the segment and can access the other parts of the segment with load and store operations without involving the process which owns the involved portion. Several PGAS programming languages

are currently available, such as Unified Parallel C (UPC), Co-Array Fortran or X10, GASPI and its implementation GPI-2 [40].
PGAS programming models exploit one-sided communications and are best suited on systems with Remote Direct Memory Access (RDMA) capability, which enables all process to access data on the global partitioned space directly through network interface controller, without the active involvement of the CPU or the system. One of the main advantages of the one-sided approach is the reduction in memory duplication and data movement, since data is directly written on the correct memory location without the need of an intermediate buffer as for two-sided communications. The fact that CPU does not need to get involved in the data transfer also allows for better computation-communication overlap. Two-sided communication also need a synchronisation mechanism, while in the one-sided approach requires that the receiver checks whether the data is ready for consumption when needed. The main drawback of one-sided PGAS approach is that the sender process needs to know all its receivers informations. In a two-sided approach it is the receiver responsibility to allocate the correct buffer for reception while in PGAS the sender must know the destination address of the data. This can lead to memory duplication as each process must store its own and all its neighbours information. Another drawback is that historically most applications have been designed following a two-sided MPI approach. Switching to a one-sided PGAS communication method could imply deep changes in the code structure, which might be overwhelming for large applications. Nonetheless, PGAS implementation are becoming more popular, as their benefit clearly surpass the drawbacks.

1.6.4 Hybrid OpenMP/MPI

The hardware structure of modern HPC systems have been described in Subsection 1.4.1. Clusters are constructed using computing nodes interconnected by a low-latency high-throughput network system. Each node is then composed by one or more multi-core processors. This can be seen as a hierarchical structure in which each processor is a shared-memory system, which is connected to the rest of the architecture in a distributed-memory manner. Many applications are completely oblivious of the hardware structure and simulate a fully distributed memory environment placing a MPI process on each core. This approach however does not take advantage of the shared memory capability of modern processors. In 1.6.1 it was mentioned that MPI supports threaded application, which means that it can be combined with multi-threading approaches such as OpenMP. This can be done allowing a process to run on all cores of shared-memory socket. Multiple threads are spawned inside each process and assigned one to each core of the processor. The MPI application must be initialised with `MPI_Init_thread`, and different levels of thread safety are provided by the MPI implementations. While `MPI_THREAD_SINGLE` corresponds to a non-threaded application, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE` all provide the possibility of using threads inside a MPI application, but with different restrictions. `MPI_THREAD_FUNNELED` allows only the

main thread to perform MPI calls, which might be the case when such calls are performed outside OpenMP regions. `MPI_THREAD_SERIALIZE` is less restrictive and allows every thread to perform MPI calls, but not at the same time. It is responsibility of the programmer to guarantee that the application is compliant. Finally, `MPI_THREAD_MULTIPLE` allows concurrent MPI calls by all threads, however few restriction still apply.
The interaction between OpenMP and MPI libraries is a complex matter, even if both standards consider each other aspects and try to take them into account [41]. There are a few MPI implementation that employ a thread-based approach, such as MPC-MPI [42], however these are not always available on HPC cluster and not widely used. [43] provides a test suite to evaluate performances of hybrid implementations, while [44] gives extensive analysis of hybrid OpenMP/MPI communication characteristics and [45] compares MPI, PGAS and hybrid approaches. These studies conclude that hybrid methods are far from trivial to implement and do not always perform better than the distributed-memory approaches, however they become of interest in systems with increasing amount of shared memory cores, following the trend of modern HPC clusters.

1.6.5 Alternative communication paradigms

The ever changing structure of compute cluster makes performance portability an important issue. There exists some alternatives to the more common programming models presented above, whose objective is indeed to allow the software to have the same level of performance on different architectures without adapting the code. A few of the most popular alternative models are cited here. Kokkos [46] provides a library-approach which allows the programmer to create computation kernels that performs well on both CPU-based and GPU nodes. StarPU [47] and Legion [48] try to reach the same objective via a portable task-based paradigm coupled with efficient task scheduling. In a similar manner HPX [49] offers a portable API for parallelism based on PGAS and multithreading. Finally, SYCL [50], which is based on OpenCL, allows to write kernels that can execute both on CPUs and GPUs with the same API.

1.7 Measuring code performances

A characterisation of the performances of a code is required in order to be able to improve them. This is not a trivial task due to the fact that they can depend on many factors, such as processors, memory and network on the hardware side, but also on compilers, underlying libraries, vectorisation, among others. Performance assessment is a primordial step as it allows to detect hotspots and regions of the code that might need improvement, and also can suggest to the user which kernels are worth working on and those which can be let aside. It is also a widely known notion that any measurement system causes some perturbation on the measured event and the detected quantity could be different to the one of the same unob-

served phenomenon. Consequently a good measurement system should be as less intrusive as possible in order to give a correct estimation of measurement. As for many other quantities, there are several tools available to measure the performance of a code [51, 52, 53].
Profilers can be classified according to how they collect their information, either in statistical or event based manner. Statistical profilers often rely on sampling, which consists in probing the program's call stack at regular intervals, called sampling frequency. Sampling profilers are not very intrusive and do not cause much overhead, however they can only provide a statistical approximation of the execution. The accuracy of a sampling based profiler can be increased reducing its sampling frequency, however this would increase its overhead and intrusiveness. Event based profilers collect information thanks to trigger events. In order to determine such events, the code is generally pre-instrumented via source level instrumentation adding timers or specific function calls in the source code, by the compiler or intervening directly on the executable binary. Event based profiling is generally more intrusive than sampling, and can cause a significant execution overhead. On the other hand the information obtained is much more deterministic as they do not rely on a statistical approach.
Analysis tools can also be subdivided in two other macro-categories, sequential and parallel, according to the type of code performance they focus on. In the following a brief description of the most popular tools is given. Some of these tools are complementary to each other, and some have very similar functionalities. Most of the time they are used in combination with each other, in order to cover every aspect of the code performance or to compare measurements.

1.7.1 Sequential profilers

Sequential profilers focus on the core performance of the code. Even if each tool has its specificities, the general principle is to collect some information about the flow of instruction executed by a single process.

Maqao The two main features offered by Maqao [54] (Modular Assembly Quality Analyzer and Optimizer) are a static analysis tool (CQA) and a sampling-based profiler (Lprof). They work directly on the compiled binary, hence there is no need of re-compilation and they are agnostic of the programming language. CQA is used to evaluate the quality of the code produced by a compiler, which can include vectorisation, loop optimisation, etc. Static analysis means that only the binary file is analysed, without running the code. On the other hand, Lprof can give runtime information at both loop and function level. Both tools produce a report containing hints for the programmer on how to optimise the code. Maqao analysis has allowed to improve YALES2 performances dramatically in the past [55], suggesting for example to remove double indirections from the inner loops, hard coding loop limits when possible to help the compiler with vectorisation, and enforcing memory alignment and contiguity on array allocations.

Valgrind Valgrind [56] is a framework containing several tools for code instrumentation and debugging. Most of the tools focus on memory performance and error detection, however there is also a call-graph generator and two thread error detectors. As memory debugging in large programs can be particularly difficult, the memory debugger *memcheck* is probably the most interesting tool of the Valgrind framework. During the program execution this tool verifies that all memory operations are correct, allowing to identify incorrect memory usage. Another interesting tool is *callgrind*, which produces a graph of the function calls, with some profiling information such as elapsed time, cache misses, etc. The call-graph can be visualised and explored using another tool called *kcachegrind*.

Vtune Vtune [57] provides a various set of analysis such as stack and hardware event sampling and thread profiling. The collected information can be related to the corresponding source and assembly code and it is very useful to detect code hotspots.

PAPI A detailed set of information about performance metrics can be obtained analysing hardware counters. These are often implemented as set of registers that keep track of large amount of events related to hardware performance. However, the implementation of such counters changes among different microprocessors. PAPI [58] (Performance Application Programming Interface) provides a standardised, hardware independent API that allows to write portable analysis tools based on hardware counter analysis.

1.7.2 Parallel profilers

The objective of parallel profilers is to highlight the interaction between the different processes or threads during the execution of an application. In this case sensible data to collect could be for example load imbalance or communication patterns.

Scalasca Scalasca [59] analyses the execution behaviour of applications on large-scale systems with many thousands of processors, offering an incremental performance analysis procedure. It integrates runtime summaries with in-depth studies of concurrent behaviour via event tracing, adopting a strategy of successively refined measurement configurations. It can also identify wait states in applications with very large numbers of processes and combine them with efficiently summarised local measurements. Such analysis is useful to identify potential bottlenecks, specifically those related to synchronisation and communication.

Scorep Score-P [60] is a highly scalable measurement infrastructure for profiling, event tracing and online analysis of HPC applications. The objective of the Score-P project is to provide a standardised API which could work with different profiling tools.

Extrae+Paraver Extrae [61] is an instrumentation package that capture event based information during the program execution and produces traces that can be analysed by the Paraver suite [62]. Entry and exit to the programming model runtime, hardware counters (PAPI), call stack reference, user functions and events are all examples of the possible informations collected by Extrae. Application instrumentation is obtained either via linker preload, by modification of the executable binary or by manual source code instrumentation. A sampling mechanism is also available.

TAU TAU [63] is a portable profiling and tracing tool for the analysis of parallel programs. It provides both event based and sampling profiling capabilities. The source code can be instrumented automatically with the PDT [64] parser and manually with the instrumentation API. Compiler based instrumentation is also available. The TAU tool suite comes with several applications for profile and trace visualisation [41] and scalability analysis.

1.8 Motivation and layout of the current work

HPC architecture are quickly evolving and becoming more and more heterogeneous. Multi- and many-core processors are the standard in modern clusters and software struggle to fully exploit the potential of these new supercomputers. Classical communication patterns based on two-sided and collective MPI have possibly reached the limit of their efficiency [65] and are increasingly becoming the performance pitfall in CFD codes. In order to avoid these communication bottlenecks, solutions that are more hardware-aware have long been studied, in particular the hybrid OpenMP/MPI model [41]. The aim of this hybrid model is to reduce the memory footprint and the communication cost of the MPI implementation using a lower number of processes, without reducing the resources employed in the computation phases, where processes are substituted by threads. This methodology is particularly adapted to the multi-core architecture were several threads can run on the different cores sharing the memory inside a socket, although the added complexity of mixing the two programming models does not always guarantee a performance improvement with respect to the pure MPI implementations [44]. However, several works related to the hybridisation of CFD codes such as [66],[67],[68] and [69] give some encouraging results and argue that this might be the right approach to improve code scalability, which motivates the strategies adopted in this work.
Chapter 2 presents the CFD code YALES2 [70], which is the LES solver on which the entirety of this work is based upon. A detailed description of its data structure is given, with particular focus on those structures related to the parallel exchange. Furthermore its implementation of the PCG and DPCG algorithms is analysed, and a basic performance model of the Poisson solver is given. The benchmarks and architectures used for these performance assessments are also described. Chapter 3 presents a new graph-like data structure that have been implemented in the

code in order to try and overcome the pitfalls exposed by the Poisson solver analysis. Chapter 4 describes the work done in order to implement a hybrid loop-level OpenMP/MPI model in the code and the performances obtained with such model. Chapter 5 exposes the attempt to overcome the limitations of the loop-level OpenMP approach, implementing a hybrid coarse-grain OpenMP/MPI model and an almost equivalent MPI-3/MPI model. Finally Chapter 6 summarises the entire work in a general conclusion and gives several perspectives for future work.

Chapter 2

YALES2 parallel performances

YALES2 [70, 71, 72], the code used for this work, is a massively parallel numerical toolbox that has been developed since 2009 and currently used in both academia and industry. It includes many solvers that can address various physical problems, such as the simulation of incompressible flows, combustion, two-phase flows, heat transfer, radiative transfer and much more. All the different solvers rely on a common high-performance numerical library, written in Object-Oriented Fortran. The core library includes highly-optimised linear solvers, automatic mesh refinement [73] and load balancing, high-order Finite-Volume integration, parallel I/O, etc. All these libraries are designed for computation on unstructured meshes, which are better suited to describe complex geometries. The node-centred Finite-Volume approach on which the numerical methods are based in YALES2 guarantees the conservation of the transported quantities. A newly implemented 4-th order convection scheme [74] makes YALES2 particularly well suited for LES simulations as the eddies are not artificially damped when transported over large distances. The parallelism in YALES2 is managed with a domain decomposition implemented with the MPI paradigm, in which the mesh is split among many processes. The low-Mach incompressible Navier-Stokes solver is the most widely used and most of the numerical methods implemented for this solver are used as a base for the others. This solver uses the procedure detailed in Section 1.3 to discretise Equation 1.15 on a mesh, providing a linear system in which the unknowns are the pressure at each node of the mesh, which is then solved with the numerical methods presented in 1.3.2.

The first objective of this chapter is to present the data structures on top of which the numerical methods are implemented in YALES2. Particular focus is put on those structures linked to the parallelism and to the group of elements, which can be considered the cornerstone of YALES2. Then, the details of the implementation of the DPCG Algorithm 3 in YALES2 are exposed. The aim is to provide a sufficient insight into the mechanisms of the code to be able to understand the code behaviour under different circumstances. This chapter provides an assessment of the parallel performances of the code before the beginning of this work. Several performance pitfalls are exposed, and a simplistic performance model for the DPCG algorithm is derived. The chapter is concluded with a summary of what has been shown, and some useful hints about the strategies to implement to improve the code performances.

2.1 Data structure of YALES2

In YALES2, the mesh and its connectivity can be described via 4 principal entities:

- **elements**, i.e. the cells of the mesh;
- **faces** of the elements;
- **nodes** of the mesh;
- **pairs** connecting two nodes, i.e. the edges of the mesh.

Some Finite-Volume solvers use the elements of the mesh as control volumes, however YALES2 builds its control volumes around the nodes of the mesh, as in Figure 2.1. Furthermore, YALES2 is node centred, which means that the physical quantities are stored at the nodes of the mesh. Each control volumes, also called dual cell, is bound by facets connecting the centroids of the elements to which the node belongs to the mid-point of each pair, as in Figure 2.1 in 2D, and to the centre of each face of which the node is a vertex in 3D. The mesh formed by the control volumes is called dual mesh [75, 76]. Fluxes between control volumes are computed on their external facets, each of which is associated to a pair of nodes. Indeed most of the physical quantities are stored at the nodes of the mesh, while the differential operators are created and stored on the pairs to facilitate the computation of such quantities.

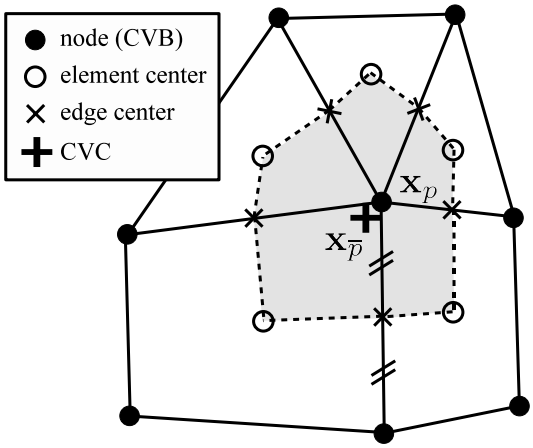


Figure 2.1: Control Volume in YALES2: x_p is the point around which it is built and $\bar{x}_p$ is its barycentre.

2.1.1 Single Domain Decomposition

In order to be able to perform massively parallel computations, many CFD solvers adopts a parallelisation strategy called single domain decomposition, or SDD. This means that, when multiple processes are used to tackle a big problem, the mesh is

split among those processes, and each one computes only its portion of the entire domain. However, the sub-problems that each process treats are not independent and data must be exchanged at the frontier between the processes. Figure 2.2 represents this domain decomposition, with the region highlighted in dark grey being the interface between processes where communication is needed. An efficient method for this data exchange is paramount to obtain strong parallel performances from the solver. The strategy used in YALES2 will be explained later in 2.1.3.
Beside communication efficiency, another important aspect of parallel performance is the load balancing between processes. Usually communication comes as a synchronisation point for the different processes, as they have to wait on one another to exchange data, so it is of the utmost importance that the computation work in-between communication phases is as homogeneous as possible amongst all processes. During a simulation, an initially good mesh partitioning can become ill balanced. This can be the case when a mesh is locally refined or coarsened, or if lagrangian particles are injected at a particular point in the mesh. Load imbalance can come also from adaptive numerical schemes (p-refinement), resolution of chemistry in reaction zones, etc.
One of the main drawbacks of the SSD is that in such case, the entire mesh graph must be re-constructed and its partitioning completely re-computed. Furthermore, if the interface between processes changes, the entire communication structure between such processes has to be built again to take into account such changes.
SSD is also particularly inefficient for memory accesses. Often, especially for differential operators, in order to compute the gradient of a quantity on a node, the values at all its neighbouring nodes are needed as well. In general, even when a mesh is split among different processes, each process can work on a subdomain of several hundreds of thousands of nodes. This means that a node's neighbour can be stored very far from it in memory, hence memory access patterns can become quite inconsistent. Accessing data from the RAM is extremely expensive in terms of processor cycles, consequently guaranteeing that all memory accesses are done into the processor cache would be extremely beneficial. To deal with this problem there is a technique called cache blocking, in which data is split in contiguous blocks of relatively small size that fit in the cache memory of the processor. YALES2 performs cache blocking through a double domain decomposition (DDD).

2.1.2 Double Domain Decomposition

In YALES2, on top of the single domain decomposition explained above, each process splits its own subdomain in several groups of elements (ElGrps), which are small enough to fit into L3 or even L2 cache. For this technique to be efficient, the computation on each group must be totally independent from the others, consequently there must be a data exchange between groups. The groups are indeed built similarly as the subdomains in the SDD, which means that the domain of each process is split in different parts, but in this case such parts still belong to each process on which they are generated. Again, when computing differential operators, each node

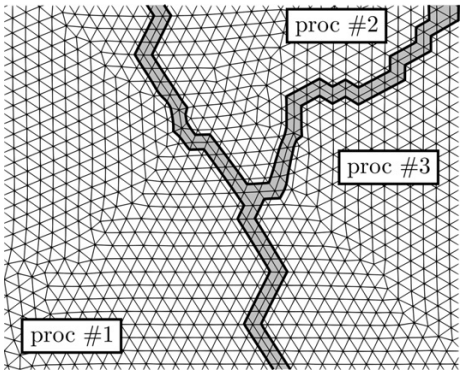


Figure 2.2: Single domain decomposition

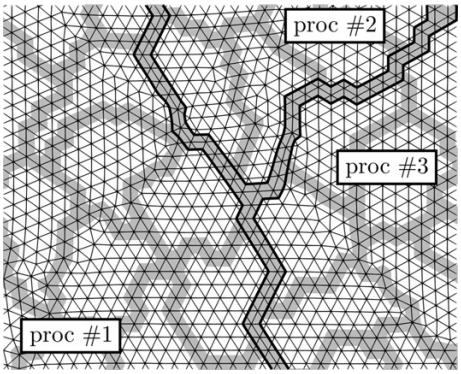
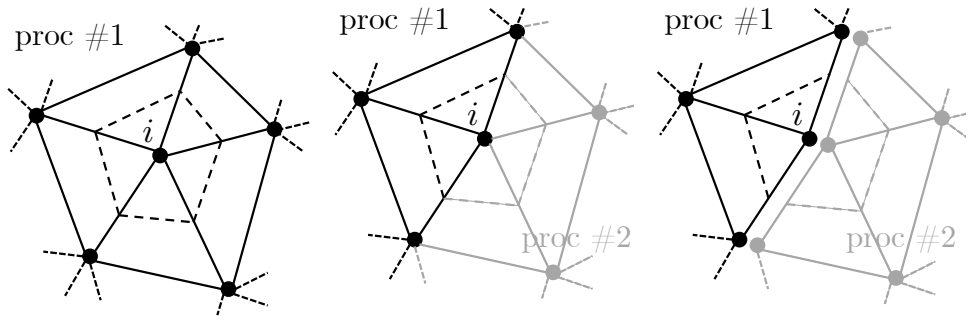


Figure 2.3: Double domain decomposition

needs the contribution of all its neighbours, even those on another group if such node lies on the frontier of the ElGrp. Logically, there is still the need of a parallel exchange for those nodes that are on the interface between different processes.
A mesh partitioned with DDD is represented in Figure 2.3, where the dark grey area is the interface between the processes, and the nodes highlighted in lighter grey are those on the interface between the different groups of elements.
Besides the cache awareness, DDD is also beneficial in those cases where a dynamic load balancing is needed: instead of re-constructing the entire graph of the elements, the load balancing can be done on a graph whose vertices are the groups of elements, and the edges the faces of those elements that belong to different groups. Depending on the size of ElGrps, this graph can be much coarser, hence faster to build and to partition. Furthermore, if the communication structure between processes is built on top of the group of elements, if some ElGrps are migrated between processes or their interface with a neighbour process changes, only that part of the communication structure needs to be rebuilt. One drawback could be that the load balancing has the granularity of the group of elements, but often their size is small enough that this is not a real problem.
YALES2 uses the groups of elements also as nodes of the deflation grid used to solve the Poisson's equation with the DPCG algorithm. With a slight abuse of language but for the sake of clarity, for the remainder of the manuscript, the grid composed by the control volumes and the deflation grid are referred to respectively as fine and coarse grid.

2.1.3 Data exchange

One particularity of YALES2 is that nodes, pairs and faces at the interface between processes are duplicated, as represented in Figure 2.4c. This means that each process can compute independently on the entirety of its sub-domain, but the duplicated nodes only account for the partial contribution coming from the inside of the domain. In order to have the complete, correct value, the process must receive from



(a) Node inside the domain (b) Node on the interface between two processes (c) Duplicated nodes at the interface between process

its neighbours their partial contribution on those nodes. The exact same principle applies to the groups of elements as well.
This data exchange between ElGrps and processes is necessary for all those operations involving the neighbouring nodes such as the computation of gradients, divergence, etc., but also when computing mask values on the nodes based on the rank number or the ElGrp colour or other quantities that might have different values on the various duplications of the node.
The data structure that takes care of the data exchange between groups is called internal communicator (IC). The IC is an array whose elements represent unique nodes forming the interface between the different groups. Each group can access the position of its interface nodes in the IC, but the vice-versa is not possible.
In order to perform communication between processes, another data structure called external communicator (EC) is employed. There is an external communicator for each of the neighbouring process, plus one called self communicator for periodic boundary conditions. An EC consists of two buffers, one to send (SendEC) and one to receive (RecvEC) data. These buffers have access to those nodes in the internal communicators who are also part of the interface with the neighbour process associated with the EC.
Boundary conditions are taken into account in a similar way as the groups of elements through the internal communicator.
The data structure of a process can be synthesised as in Figure 2.5. Algorithm 4 illustrates a typical computation and communication phase in YALES2. All its different steps will be detailed in the following.

Step 1: initialisation of the internal communicator. The internal communicator (IC) is an array on which a reduction operation $\oplus$ is performed for all nodes on the interface between ElGrps. Its value needs to be set to an initial value, which depend on $\oplus$ before such reduction operation can begin. Typical examples are an initial value of 0 if $\oplus$ is a sum, 1 for a product, the largest negative value for a *max* and the largest positive value for a *min* operation.

Algorithm 4: Computation and parallel data update for an operation $\oplus$ (max, min, sum, ...)

```
// Step 1:
Set initialisation value for all IC nodes according to ⊕ ;
// Step 2:
foreach ElGrp do
    foreach node do
        compute ElGrp[node];
    end
    // Figure 2.6
    foreach interface node do
        index = ElGrp[interface node] to IC[node];
        IC[index] = IC[index] ⊕ ElGrp[interface node];
    end
end
// Step 3:  (Figure 2.7)
Compute and update boundary conditions;
// Step 4:
foreach EC do
    foreach node do
        index = EC[node] to IC[node];
        SendEC[node] = IC[index];
    end
    Send data in SendEC;
end
// Step 5:  (Figure 2.8)
foreach EC do
    Receive data in RecvEC;
    foreach node do
        index = EC[node] to IC[node];
        IC[index] = IC[index] ⊕ RecvEC[node] ;
    end
end
// Step 6:  (Figure 2.9)
foreach ElGrp do
    foreach interface node do
        index = ElGrp[interface node] to IC[node];
        ElGrp[interface node] = IC[index];
    end
end
```

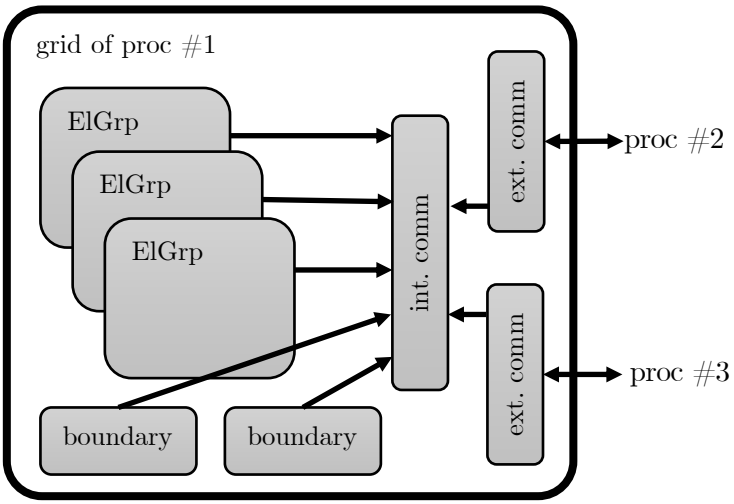


Figure 2.5: Data structure of YALES2

Step 2: computation and update of the internal communicator. As stated above, computation in YALES2 is performed with a double loop, the first on the ElGrps, and the second on the necessary entity (nodes, pairs, elements, faces). For example, to compute a gradient $\nabla\phi$ of a generic scalar ϕ on a generic node i, the values of ϕ on all the nodes j connected to (neighbours of) i are necessary, and some metrics values M which depend on each of the pairs $i \to j$, as in Equation 2.1.

$$\nabla\phi = \sum_{j \in N_i} \mathcal{F}\left(\phi_i, \phi_j, M_{i\to j}\right) . \tag{2.1}$$

Two different situation are possible:

1. Internal node as in Figure 2.4a: all $j \in N_i$ can be accessed directly in the process, or in the group, hence no data update is necessary for the node i.

2. Interface node as in Figure 2.4b: some of the neighbouring nodes belong to another process (or ElGrp). Only part of the gradient can be computed in the current group, and likewise for the other group sharing the node. Consequently a data exchange is necessary between the groups (and processes) to have the correct and complete value of $\nabla\phi_i$.

Then, the reduction for all nodes on the frontier of the group is performed on the internal communication. Two indirection arrays allow to easily loop on all such nodes and find the corresponding IC item on which to compute $\oplus$, which in the case of Equation 2.1 is a sum.

Step 3: compute the boundary conditions and finish the update of the internal communicator. The last contribution needed in order to have the complete value on each node is that of the boundary conditions. Boundary conditions

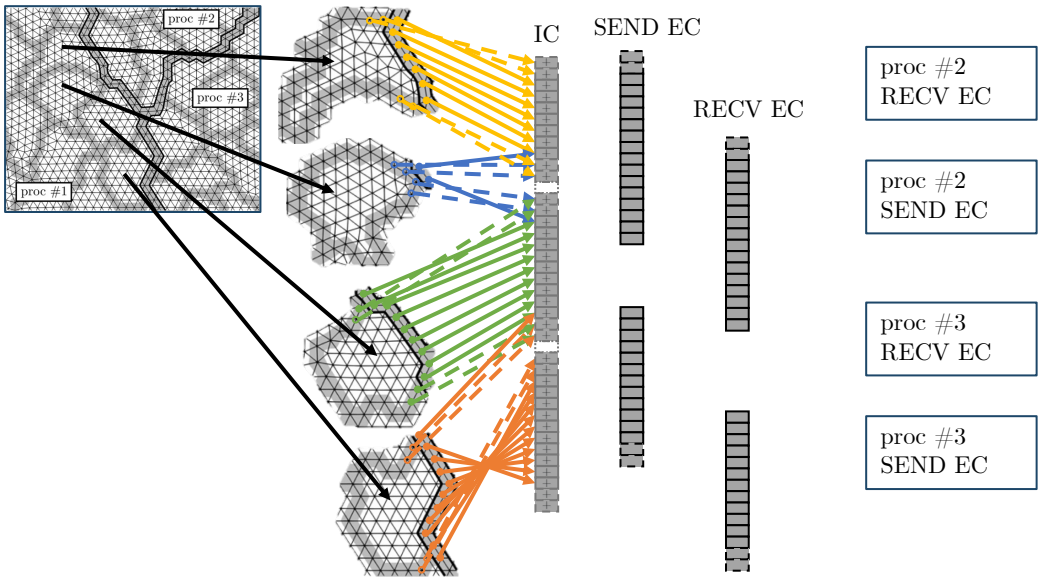


Figure 2.6: Step 2: update the internal communicator with values on the frontier of each ElGrp. Continuous lines indicates nodes that belongs to the external communicator as well, while dashed ones are for nodes only on the internal communicator. Only a few are indicated as example.

(BC) are treated separately from the rest of the data in YALES2, but the data exchange remains the same. Each process has complete awareness of the BC that intervene on its subdomain, hence these do not need a parallel update. Due to their separate treatment this step can be moved anywhere after Step 1 and before Step 6. Nonetheless their contribution has to be taken into account to obtain a correct value on the IC. As for the data on the ElGrps in Step 2, a computation phase with a double loop on each BC and all its nodes is followed by a reduction in the IC. Naturally, each and every node on the BC needs to contribute to the internal communicator.

Step 4: fill the Send external communicators and send the data to neighbour processes. After the update of the internal communicator, those nodes that lie on the interface between different processes still have incomplete values. As for the IC update, two indirection arrays are used to copy the data from the IC to the Send external communicator (SendEC). The latter is just a buffer used for the MPI two-sided exchange and does not need to be initialised as the data is simply copied on it and no other operation is performed on it. Once a SendEC has been filled with the right values, data is sent to the corresponding process, as schematised in Figure 2.7.

Step 5: receive the data in the Receive external communicator and update the internal communicator. As each the neighbour processes have performed Step 4, the current process receives the data into the Receive external com-

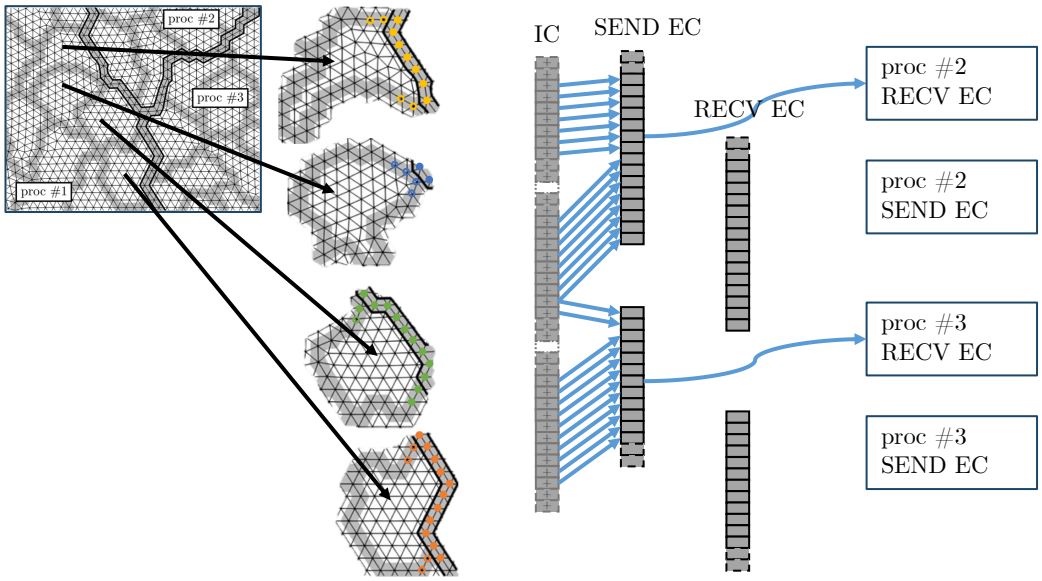


Figure 2.7: Step 4: fill the Send external communicator and send the data to each neighbour.

municator (RecvEC). As for the SendEC, this is a buffer in which data is copied and does not need initialisation. Once the MPI two-sided exchange has been completed, the same indirection arrays used to fill the SendEC are used to get data from the RecvEC and complete the IC update with the same operation $\oplus$. This mechanism is represented in Figure 2.8.

Step 6: copy the internal communicator into the data. After all the previous steps, the IC holds the complete and correct values of $\nabla\phi$ for all the nodes on the interface between the groups. This information however needs to be transmitted back to the ElGrps, as the IC is meant only for data exchange and not for computation. Data is then copied back from the IC to the respective nodes on all the groups using the same indirection arrays used in Step 2. Again, since data is copied, hence the previous value is overwritten, there is no need to initialise such values.

2.1.4 Computation of the matrix-vector multiplication in YALES2

The matrix-vector multiplication $b = Ax$ in YALES2 deserves to be explained in more detail as it is implemented in a peculiar way. For node-centred data, each non-zero value A_{ij} represents a connection between the nodes i and j. Rather than store A in a matrix form, this is stored as a list of coefficients, one for each oriented pair $i \rightarrow j$. Each pair represents the facet dividing the control volumes i and j.
In this case, the vector b and x represent data stored on the control volumes (nodes), which is the most common case in the code, but the same principle is applied to

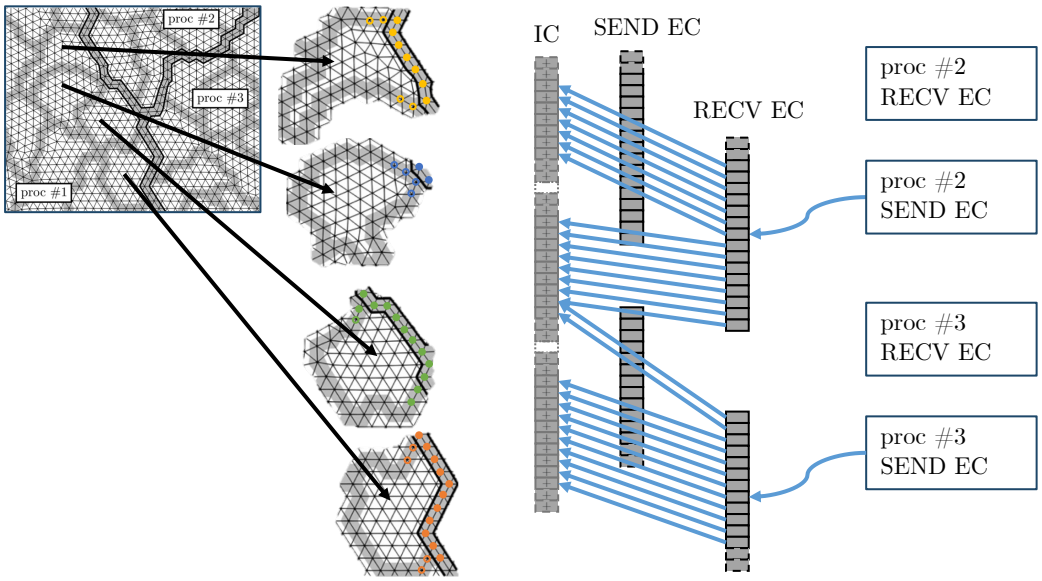


Figure 2.8: Step 5: receive data on the Recv external communicator and add it in the internal communicator.

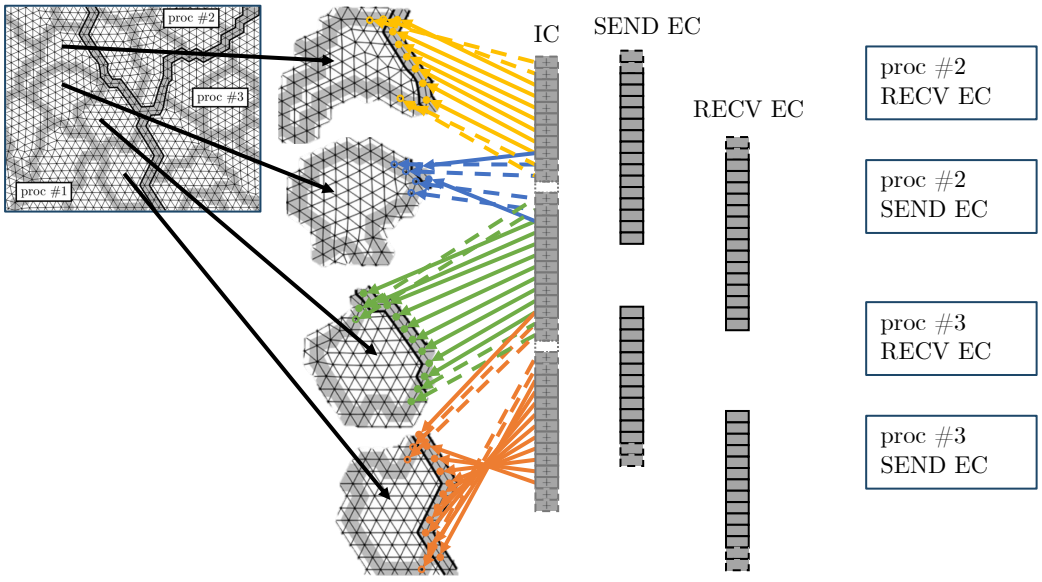


Figure 2.9: Step 6: Copy back the internal communicator to the ElGrps. Continuous lines indicates nodes that belongs to the external communicator as well, while dashed ones are for nodes only on the internal communicator. Only a few are indicated as example.

data on the elements (connected by their faces instead of pairs) or on the ElGrps, as it will be explained in 2.1.5.
As for all other data, A is decomposed in ElGrps, which is to say that each ElGrp holds and has access only to that portion of A which corresponds to all the pairs connecting all the nodes of the group. As a consequence, as for all the other examples listed above, the matrix-vector multiplication gives incomplete values on the frontier of the different ElGrps and b needs to go through all the steps of Algorithm 4 in order to be correct.
As it cans be seen in Algorithm 5, the computation is performed via a double loop, the first on the ElGrps and the second on all its pairs. The second loop requires two indirection arrays to point to the correct couple of nodes i and j in b and x, while the value in A is directly determined by the pair.
Furthermore, the matrix A is decomposed into three different matrices: A^D, A^S and A^A, which hold respectively the diagonal, the symmetrical and anti-symmetrical contributions in A.
As the pairs are oriented, the contribution of the pair has opposite sign on each node, consequently the coefficients of the symmetric and antisymmetric matrices are such that $A^S_{ij} = A^S_{ji}$ and $A^A_{ij} = -A^A_{ji}$. The diagonal contribution is taken into account with an additional loop on the nodes. A^D is stored on the nodes since it influences only the local value and does not depend on neighbouring nodes.
Finally, since the contribution of each pair is added onto $b[i]$ and $b[j]$, the entire vector b must be initialised with a null value before the computation. Algorithm 5 is written for the general case in which all three A^D, A^S and A^A exists and are not null. However if one or two of such contributions are absent, a simplified version can easily be derived.

Algorithm 5: Matrix-vector multiplication in YALES2

foreach *ElGrp* **do**
  $b = 0$ // Initalisation of b
  foreach *pair* **do**
    i = pair to node_1[*pair*];
    j = pair to node_2[*pair*];
    $c^S = A^S_{pair} \times (x[j] - x[i])$// Symmetric contribution
    $c^A = A^A_{pair} \times (x[j] + x[i])$// Anti-symmetric contribution
    $c = c^S + c^A$;
    $b[i] = b[i] + c$;
    $b[j] = b[j] - c$;
  end
  foreach *node* **do**
    $b[node] = b[node] + A^D[node] \times x[node]$// Diagonal contribution
  end
end

2.1.5 Implementation of the DPCG algorithm in YALES2

The implementation of the DPCG Algorithm 3 in YALES2 is quite straightforward. The most important aspect to underline is that, as for all other parts of the code, computation is broken down in loops over the groups of elements and the computation and communications necessary to perform the matrix-vector multiplications are done respectively as in Algorithm 5 and Algorithm 4. The convergence, based on the norm of the residual, is verified via a reduce-broadcast scheme. This is deemed necessary in order to avoid potential differences in the evaluation of $\|r\|$ due to numerical rounding errors in the `MPI_ALLREDUCE` collective operation. Performing the reduction on the master process and broadcasting the convergence condition ensures that all process would either exit the loop or continue the computation. Three other collective operations (`MPI_ALLREDUCE`) are performed in this algorithm. Two of them evaluate the dot-products to compute the values of ρ and σ. The last one is not really necessary for the DPCG per-se but provides an adaptive criterion for the convergence optimisation of the deflation step [77].
The development and implementation of the deflation in YALES2 have been the subject of the thesis of Malandain. A detailed description of the implementation of the deflation algorithm, i.e. the line *Solve* $\hat{A}d_{k+1} = W^T \left(AK^{-1} - I\right) r_{k+1}$ of Algorithm 3, together with the study of different techniques to improve the DPCG algorithm performances, such has the recycling of previous residual to have a better estimation of the first solution vector and an adaptive convergence criterion, can be found in his work [72].
The groups of elements have a primordial role in the implementation of the deflation step of Algorithm 3 in YALES2 solvers. The deflation matrix W is in fact created from the groups of control volumes derived form the ElGrps. However, thanks to the code data structure, the matrix W is never explicitly declared: the left-multiplication of a vector on the fine mesh by W^T is computed summing its components in each group to create the resulting vector; in a similar way, the left-multiplication of a vector on the coarse mesh by W is performed assigning to each control volume of a group the corresponding value on the original vector. In practice this means that the solution of the Poisson's Equation on the coarse mesh produces a constant piecewise solution on the fine mesh.
Less trivial is the construction of the operator associated to the matrix $\hat{A}$. As a logical implication of the fact that the Laplacian operator A on the fine mesh is based on the pair of control volumes, $\hat{A}$ is constructed on pair of groups of control volumes. The result of $\hat{y} = W^T A W \hat{x}$ can be of aid to build $\hat{A}$. To multiply $\hat{x}$ by W means that values of $\hat{x}$ are copied in a vector on the fine mesh, in such a way that the $i-th$ component of such vector will get the value of $\hat{x}_{g_i}$, where g_i is the group to which the $i-th$ control volume belongs. The result of the multiplication by A is the addition and subtraction of a coefficient $a_{ij}\left(\hat{x}_{g_i} - \hat{x}_{g_j}\right)$ to each of the i and j components of the result vector for each pair $i \leftrightarrow j$ of control volumes on the mesh. A direct consequence is that if two control volumes belong to the same group, the resulting contribution is null, hence only pair of nodes across the frontier of different

groups contribute to the operator. The multiplication by W^T sums the values on each group. Consequently each component $\hat{y}_{g_i}$ of the $\hat{y}$ vector can be written as the sum of the contribution of the neighbouring groups as in Equation 2.2:

$$\hat{y}_{g_i} = \sum_{g_i \leftrightarrow g_j} \left(\sum_{j \in g_j} a_{ij} \right) \left(\hat{x}_{g_i} - \hat{x}_{g_j} \right) . \tag{2.2}$$

To summarise, the contribution of a pair of groups to the $\hat{A}$ operator is the sum of the contribution to A of those pairs of control volumes that are across the boundary between groups.
The deflation step itself can be solved with a PCG algorithm. YALES2 indeed adopts an optimised version of Algorithm 2 to solve the resulting Poisson equation on the coarse grid. The optimised version, detailed in Algorithm 6, allows to perform only one point-to-point and one collective communication, for the operations respectively in green and red. In addition to the two dot-products and the norm of the residual, another quantity is added to the collective communication. This value is used to estimate the need to reset the algorithm, and for the sake of clarity has not been added to the algorithm. In order to avoid the afore-mentioned problems deriving from the rounding errors in the collective communications, each process evaluates the convergence locally. If it estimates that $\|r_{k+1}\| < \epsilon$ then it sends a 1, otherwise a 0. The collective communication consists then on a sum of 4 values across all processes. All processes would exit the loop if the result of the `MPI_ALLREDUCE` is equal to the number of processes, otherwise they will continue. This optimisation however could produce some differences on the value of the residual across the different workers. To avoid an accumulation of these numerical errors, an exact evaluation of the residual $r_{k+1} = b - Ax_{k+1}$ is performed with a certain frequency (every 10 iterations) before the computation of w_{k+1}. This part as well has been removed form Algorithm 6, as it was deemed as an unnecessary complication.

2.1.5.1 Node ownership, group connectivity and half-halo communications

An important concept arising from the node and pair duplication among groups is the ownership of such entities. As practical as it is to have a node duplicated multiple times in order to allow this efficient data structure, there must be one of this multiple copies which is considered to be the original one. Since the group of elements are the cornerstone of the data structure of YALES2 and all computation is done inside each group, the indexing of the elements, nodes, etc is never global, but each group has its own. The groups however have a universal numbering, called colour, which is incremental across all processes, i.e. process P_0 owns the groups with colour from 0 to G_0, process P_1 owns those with colour from $G_0 + 1$ to G_1 and so on. YALES2 adopts a convention for which the duplicated entity (node, pair or face) always belongs to the group with the smallest colour among those who share such entity. This naturally implies that if a node, for example, is at the interface

Algorithm 6: Optimised implementation of the PCG method on deflation grid

$K = diag(A)$;
$r_0 = b - Ax_0$, $k = 0$;
$w_0 = K^{-1} r_0$;
$p_0 = w_0$;
$q_0 = Ap_0$;
$\rho_0 = r_0^T w_0$;
$\sigma_0 = p_0^T q_0$;
$\alpha_0 = \frac{\rho_0}{\sigma_0}$;
while *not done* **do**
  $x_{k+1} = x_k + \alpha_k p_k$;
  $r_{k+1} = r_k - \alpha_k q_k$;
  $w_{k+1} = K^{-1} r_{k+1}$;
  $s_{k+1} = Aw_{k+1}$;
  $\rho_{k+1} = r_{k+1}^T w_{k+1}$;
  $\delta_{k+1} = s_{k+1}^T w_{k+1}$;
  if $\|r_{k+1}\| < \epsilon$ **then**
    exit the loop
  end
  $\beta_{k+1} = \frac{\rho_{k+1}}{\rho_k}$;
  $p_{k+1} = w_{k+1} + \beta_k p_k$;
  $q_{k+1} = s_{k+1} + \beta_k q_k$;
  $\sigma_{k+1} = \delta_{k+1} - \beta_{k+1}^2 \sigma_k$;
  $\alpha_{k+1} = \frac{\rho_{k+1}}{\sigma_{k+1}}$;
  $k = k + 1$;
end
The result is x_{k+1}.

between processes, it always belong to the smallest process, due to the colouring convention of the groups. This has particularly important consequences in how the operators and algorithms are constructed, especially for the deflation.
Subsection 2.1.5 gives a brief description of the implementation of the DPCG algorithm in YALES2. In particular, Equation 2.2 describes the construction of the $\hat{A}$ operator. As the $\hat{A}$ operator, the pair of groups themselves are built by finding those pairs that link nodes belonging to different groups. Such connectivity can be created by each process locally thanks to the colouring convention described above. In doing so however, a group is able to detect a connectivity only with groups of lower colour. For the same principle, a process will find a connectivity only with a neighbour of lower rank. As a consequence, the process with the highest rank will have a possibly much larger number of pair of groups than the process with rank 0, as the latter can only detect those pairs created by its own groups, while the former sees those with all its neighbour processes as well. Even if the number of groups is perfectly balanced among processes, the number of pair of groups is not. On this premise, a

Algorithm 7: Half-ghost mechanism implemented in Algorithm 6

```
...;
w_{k+1} = K^{-1} r_{k+1};
Copy w in neighbours ghost; low→high colour;
s_{k+1} = A w_{k+1};
Send s contribution to neighbours; high→low colour;
ρ_{k+1} = r^T_{k+1} w_{k+1};
δ_{k+1} = s^T_{k+1} w_{k+1};
if ||r_{k+1}|| < ε then
    C = 1
else
    C = 0
end
ALLREDUCE({ρ_{k+1},δ_{k+1},...,C},SUM);
if C == N_workers then
    exit the loop
end
...;
```

half-halo system for parallel point to point communication is created. Each process allocates a halo structure to receive data from the neighbour processes with whom it has created a connectivity. This means that, for a couple of processes, the halo structure is allocated only on the process with higher rank, which was able to detect and build the pair of groups with its neighbour. A schematic representation of this process is given in Figure 2.10. Using the terminology defined for the optimised PCG in Algorithm 6, each process can compute $w = K^{-1}r$ on its own groups independently. A first communication phase is then needed to fill the halo of the highest rank processes with the values of w coming from their lower rank neighbours. After

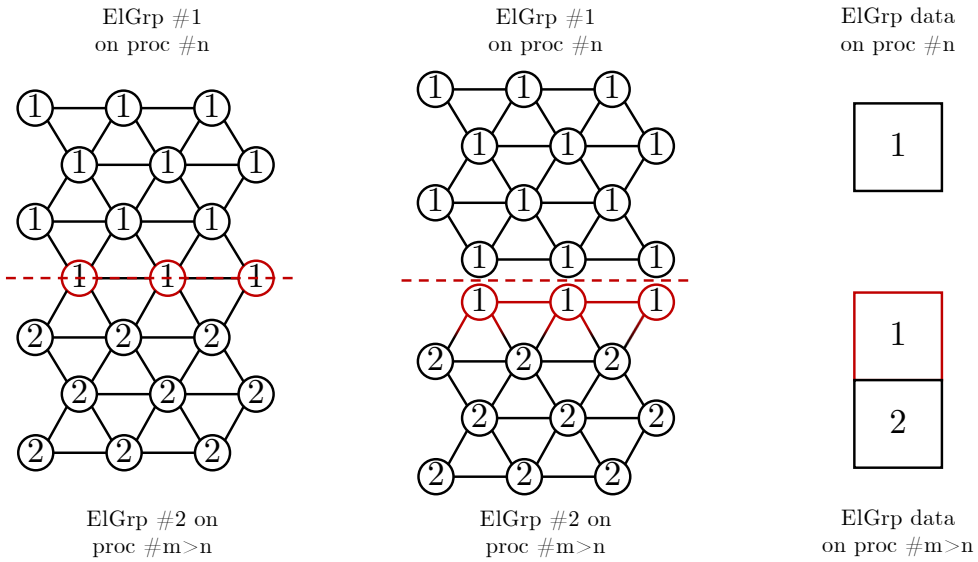


Figure 2.10: Schematic representation of the construction of the ElGrp data on different processes from coloured nodes

the application of the operator defined by Equation 2.2, which corresponds to the computation of $s = Aw$ on both internal and halo groups, the highest rank processes send the s value on the halo back to the lower rank neighbours, who can add this contribution directly to the values computed for their internal groups, without the need of a halo to receive the data. Algorithm 7 is an excerpt of Algorithm 6 that details this particular communication scheme. A more detailed explanation about the ghost mechanism is given in Chapter 3 (see e.g. Figure 3.2).

2.2 Performance assessment

Amongst all the algorithms implemented in YALES2 to solve the Poisson's equation, the DPCG is the one that gives best performances. However, [72] shows that the time spent by the processes on communication can be quite important, even more than 15% of the total time, for those cases in which the Poisson solver is preponderant, like non-reactive flows.
Strong and weak scalability have always been a preferred display of a code parallel performances. The speedup $\mathcal{S}$ is also a good index to compare code performances. The speedup simply is the ratio between two runtimes t_a and t_b, normalised or not, as shown in Equation 2.3. It measures the improvement of the case t_b over the case t_a. In addition to the wall-clock times (WCT), another indicator, often used in the CFD community, is the Reduced Computation Time (RCT). The RCT is computed as in Equation 2.4, where WCT is the execution time for n_p processes computing on a mesh of n_{cv} control volumes for n_{it} iterations, or time steps. The RCT is a measure of the scalability itself as its supposed to remain constant in

both weak and strong ideal scalability curves. RCT is widely used as it allows to compare the performances of different codes. This is due to the intrinsic definition of the parameter itself, as its normalises the execution time by the number of control volumes independently of how they are defined.

$$\mathcal{S}_{a\to b} = \frac{t_a}{t_b}\,. \tag{2.3}$$

$$RCT = \frac{WCT \times n_p}{n_{cv} \times n_{it}}\,. \tag{2.4}$$

The objective of this work is the improvement of the parallel performances of the YALES2 solver, however first of all an assessment of the code performance must be obtained. This section presents the main benchmarks used for this performance study, along with the platforms on which these measurements are made. Before showing the actual performance assessment, an improvement of the initial YALES2 timer system is described. Finally an assessment of the performances of YALES2 at the beginning of this work is presented.

2.2.1 Test cases

In this subsection the two main benchmarks used in this work are presented.

2D and 3D_cylinder

This benchmark case, whose mesh is shown in Figure 2.11, is used to simulate the flow around a cylinder. The mesh consists of 491′000 tetrahedral elements, and the boundary conditions on the side walls could be set to be walls with either slip or no-slip conditions or periodic, to simulate an infinite domain. This small mesh is particularly useful to perform small studies on a single or few processes. Its 2D version, which consists of a mesh of 25′000 triangular elements, is used mainly for visualisation purposes and quick debugging.

Preccinsta

The main benchmark used in this study is named Preccinsta. It consists of the constant temperature simulation of the PRECCINSTA burner, which stands for *PREdiction and Control of Combustion INSTAbilities for gas turbines*. The geometry, whose mesh is shown in Figure 2.12, consists of a plenum, a swirler injector and a rectangular combustion chamber with a conical convergent towards a cylindrical exit pipe. This burner has been extensively studied both experimentally [78] and numerically [79, 80], and it has been used as a validation benchmark for combustion model and numerical methods for Navier-Stokes equations, including with YALES2. The air is pumped towards the plenum with a flow rate of 734.2g/min. In the experimental configurations methane is injected by 12 tubes of 1mm diameter with a total flow rate of 35.9g/min, while for numerical simulations the gases are considered as perfectly premixed. The mixture goes across the swirler, which is composed

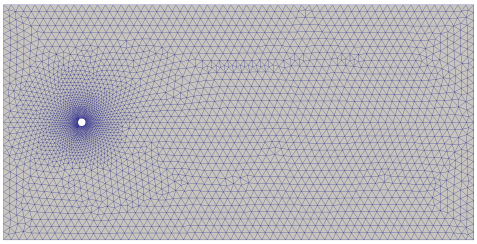


Figure 2.11: Mesh of the 3D_cylinder benchmark

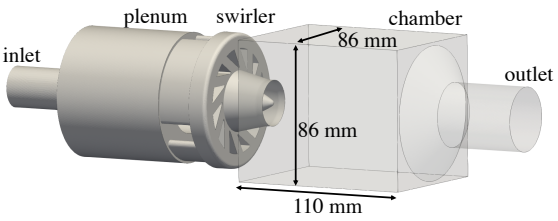


Figure 2.12: PRECCINSTA burner

by 12 slots inclined at 40 degrees before entering the combustion chamber. The burned gases then exit the chamber through the convergent cone. The simulations performed in this work consider only non-reactive flows at constant temperature, for which the incompressibility and constant density approximations are valid. Different meshes are used, with sizes of 1.7, 14, 110, and 878 million elements,

2.2.2 Platforms

The YALES2 results provided in this work have been obtained on the different clusters presented in this Subsection.

MYRIA

The MYRIA cluster is operated by the CRIANN (Centre Régional Informatique et d'Application Numériques de Normandie). It consists of several partition of different nodes, including GPUs and Intel Xeon Phi KNL accelerators. The partition used for this work consists of 332 Broadwell Xeon E5-2680 v4 [81] bi-socket nodes (9296 compute cores). Each node comes with 128GB of DDR4-2400MHz memory. The nodes are connected via a low latency, high bandwidth (100Gb/s) Intel Omni Path network. Each processor has 14 cores that run at a base frequency of 2.40GHz, with access to a dedicated 32+32KB L1 and 256KB L2 cache, while they all share a 35MB L3 cache. The processor comes with 4 memory channels, providing a maximum bandwidth of 76.8GB/s. The technology supports Intel AVX2 instructions, allowing each core to perform 2 FMA operations on a 256 bits vector per clock cycle. While Hyper-Threading and turbo boost are supported, these options were not activated.

IRENE-SKL

IRENE-SKL identifies the Intel Skylake partition of the Juliot-Curie supercomputer operated by the CEA (Commissariat à l'Energie Atomique) and hosted by the TGCC (Très Grand Centre de Calcul). This partition consists of 1656 Skylake-8168 bi-sockets nodes (79,488 compute cores), with 192GB memory each. The underlying network is a Fat-tree topology 4x EDR (4x 25Gbps) InfiniBand Mellanox, with a

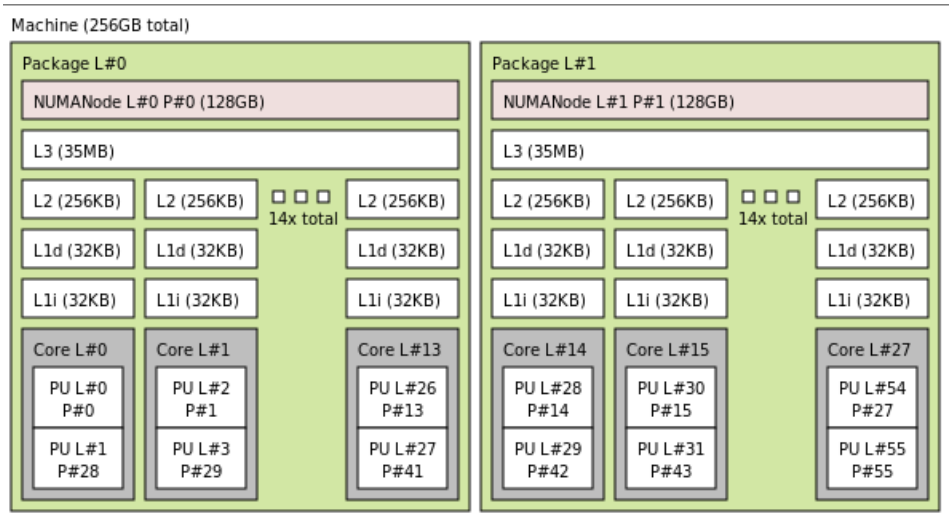


Figure 2.13: Intel Broadwell Xeon E5-2680 v4 topology

6.86PF peak performance.
In spite of being designed on the same 14nm technology process as the Intel Broadwell, the Intel Xeon "Skylake" processor provides higher performances due to a higher core count, a higher memory bandwidth and AVX-512 ISA support. The Intel Skylake 8168, which is the one deployed on the Joliot-Curie machine, has 24 cores, with a 2.7GHZ base non-AVX Core frequency. They are provided with a shared 33MB LLC and a 205W nominal TDP, and it can deliver a 2 TFLOP/s DP Rpeak. It supports advanced RAS features including SMT2, 3 UPI links and 2 AVX512 execution units. This allows each core to execute 2 FMA (Fused Multiply Add) on 8 DP (Double Precision) per cycle using a 512 bits vector width [82]. Each node is composed by two 24-cores 8168 Skylake processors, resulting in a 48 cores node. Although the Skylake processors supports Turbo boost to increase its clock frequency and Intel Hyper-Threading technology, these have not been activated for the cases presented here.

IRENE-AMD

IRENE-AMD is another partition of the Juliot-Curie cluster, which contains 2292 AMD EPYC Rome 7H12 bi-sockets node (293,376 compute cores). Each node has a 256 GB DDR4 memory, and they are are interconnected through a Dragonfly+ topology based on 4x HDR (4x 50Gbps) InfiniBand Mellanox network at switch level. This partition has a 12.2 PFLOP/s peak performance.
The 7H12 processor [83] can deliver up to 2.66 TFLOP/s DP peak performance (Rpeak). It has 64 Zen2 cores, with a 2.6GHz Base clock Speed and up to 3.3 GHz when Turbo boost is enabled, with a maximum of 280W TDP. It supports 8 memory channels and up to 2 DIMMs per channel running at 3200 MT/s with 256 MB L3 shared cache and 64 KB (32+32) L1 and 512KB L2 dedicated cache

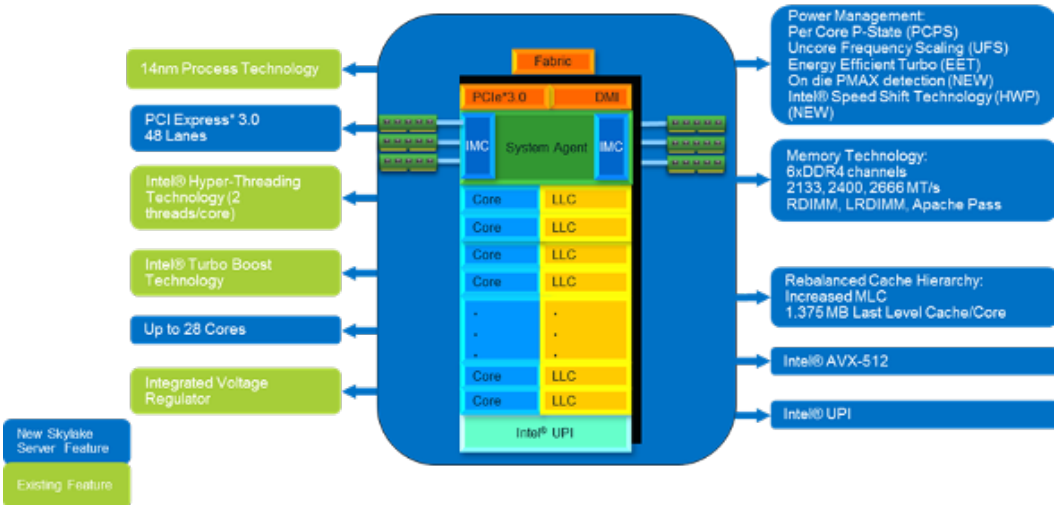


Figure 2.14: Intel Skylake Architecture

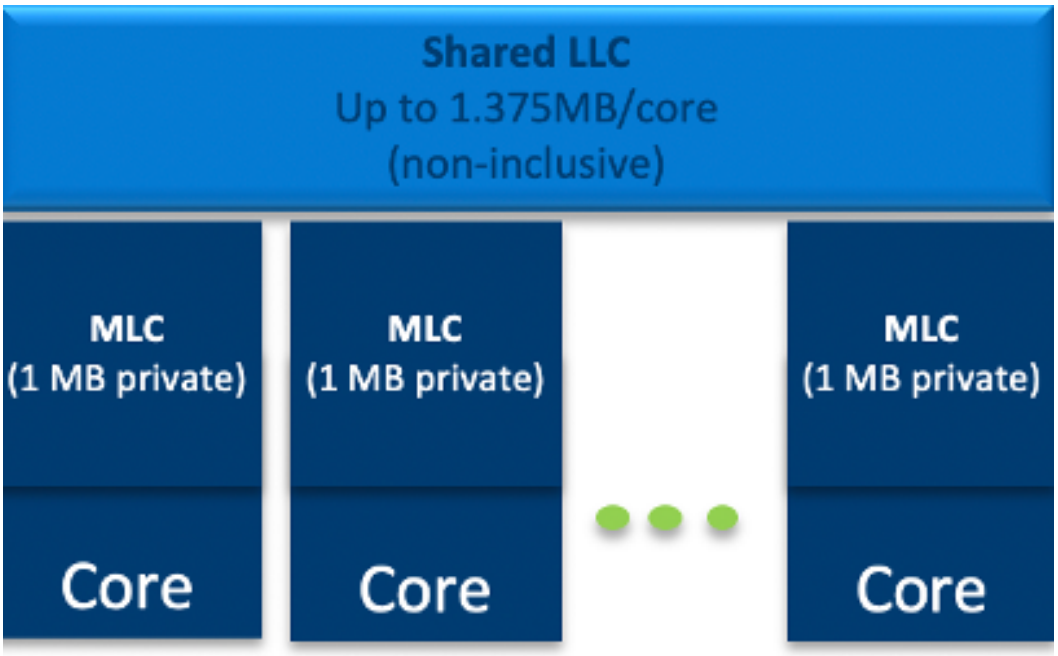


Figure 2.15: Intel Skylake cache structure

Cluster	MYRIA	IRENE-SKL	IRENE-AMD
Processor	Broadwell E5-2680 v4	Skylake 8168	Epyc Rome 7H12
Cores/Socket	14	24	32
Clock Freq. $[GHz]$	2.4	2.7	2.6
L1/Core $[KB]$	32(i)+32(d)	32(i)+32(d)	32(i)+32(d)
L2/Core $[KB]$	256	1024	512
L3/Core $[MB]$	2.5	1.375	8
Bandwidth $[GB/s]$	76.8	250	288

Table 2.1: Characteristics of the processors used for this work.

per core. It supports AVX2 (AVX256) which means that it can perform 2 FMA operations on a 256 bits vector per clock cycle, reaching 16 FLOPs per core. The two sockets are connected through xGMI2 links providing up to 288 GB/s Peak bidirectional bandwidth maintaining 128 lanes PCIe Gen4 I/O and up to 256 GB/s bidirectional bandwidth for I/O capabilities. Each processor is partitioned into four logical quadrants, each being a NUMA domain. Each of the two sockets is organised in 4 logical units, which are also Numa domains, called CCD MCM (Compute Core Die Micro-Chips Module) as shown in Figure 2.16, each formed by 2 CCX (CPU Complex). Each CCX contains 4 cores that share the L3 cache memory and all cores on the same CCD have access to memory through the same memory controller.

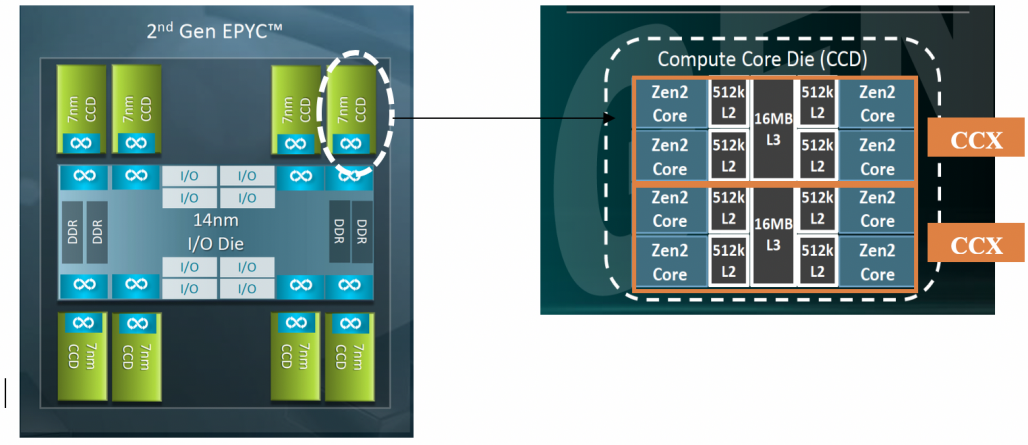


Figure 2.16: AMD EPYC Rome CCD and CCX Block Diagram

2.2.3 A new timing structure for YALES2

A brief presentation to several existing profilers and instrumentation tools have been done in Section 1.7. Although there is no doubt of the utility and versatility of such tools, it can be difficult to use them on different clusters for portability issues

or simply because they are not available on a particular infrastructure. Moreover, in an industrial context, a user might want to monitor the code performances in "production" cases, where the cost implied by the overhead of the instrumentation is not acceptable. Furthermore the typical user is probably not familiar with this particular niche of tools. These and other arguments motivate the choice of having an integrated performance monitoring system in YALES2.
At the beginning of this work the performance monitoring system was based on a timer object organised in chained lists. Each chained list corresponded to a level of the timer, 0 being the highest and 3 the deepest. The level of the timer was arbitrarily decided and hard coded for each timer. A schematic example of this data structure can be found in Figure 2.17. A function called `start_timer` would be called, taking as input the timer name and the timer level. The corresponding timer would then be searched on the chained list corresponding to the timer level. If not found a new object would be added to the chained list, otherwise a clock measure would simply be registered on the corresponding object. Another function called `stop_timer` with the same arguments would be called to end the measurement of that timer. It would look for the timer object in the same way and the difference between the clock measurement previously registered and the current time would be accumulated to the timer value.
The timer values would then be displayed during execution after a user-specified period of n_{it} iterations. The user could also select up to which timer level display the information via the input file.
This system, which looks quite rudimentary, gives a sufficiently detailed performance monitoring in most cases. In particular, high level timers show how the code is performing in the different macro-regions and having a specific timer for the communication routines allows to quickly compute the global communication to computation ratio.
During the development of this thesis and from other users experience some flaws however became evident. All timers measure their inclusive time, but there is no explicit relation between them, consequently it is not always clear, without looking at the source code, which timers are nested into each other and which are independent. Furthermore, and this is especially true for the communication timer, they give a global measure of the elapsed time but they are completely oblivious to the region of the code in which the time is being accumulated, making it impossible to even compute basic informations such as computation/communication ratios for the different macro-regions. Finally, the values that are communicated to the user are only those of the master process, hence it is more difficult to know the behaviour of the other processes and load balancing characteristics. All these aspects proved to be a major problem especially for those timers measuring collective communications. As most of the sensible regions of the code contains some sort of synchronisation mechanism, the macro-timers almost always show balanced performance across processes, but the actual imbalance is shown by the timing of such synchronisation mechanism, which almost exclusively is a collective communication or a barrier.
Taking into account the experience with different profiling tools and the aforemen-

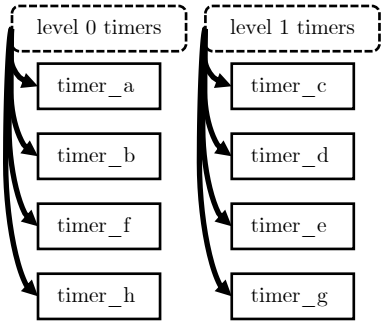


Figure 2.17: Schematic representation of the old YALES2 timers structure

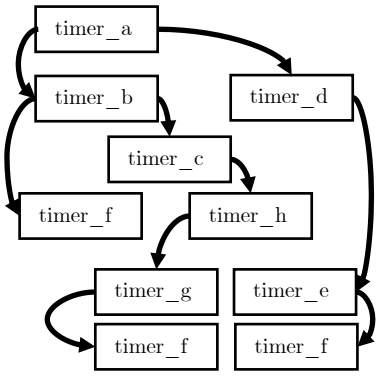


Figure 2.18: Schematic representation of the new YALES2 timers structure

tioned defects of the YALES2 timers, a new performance monitoring system was developed. This new mechanism was still based on timer object accumulating inclusive time, however, instead of arbitrarily imposed levels, timers were organised hierarchically in a tree-like data structure. Every time the `start_timer` function is called, it looks for the timer name in the chained list of "children" of the *current_timer*. The *current_timer* is simply a pointer that identifies the timer in use. The timer level does not need to be specified as an input to the function; it is automatically determined as the nesting level of the timer itself relatively to first timer that have been created. The newly started timer then becomes the *current_timer*. At a call of `stop_timer`, the *current_timer* pointer is given back to the parent of the one that just ended its measurement. Figure 2.18 gives a schematic example of this new data structure.

As for the previous system, timing information is displayed every n_{it} iterations. The user can still specify the level up to which information is to be visualised, however that now correspond to the timer nesting level. Global information for selected "notable" timers is still printed. In order to reduce the timing overhead, if the timer level is below the one that needs to be printed and it is not notable, its measurement is not performed.

In order to quicken the research of the correct timer in the chained list, a hash is generated from the timer name, and hence a hash comparison is performed to exclude wrong timers. Once a corresponding hash is found, a string comparison is performed to confirm that timer is actually the correct one. String comparison is computationally expensive to perform, accounting for more than 50% of the whole timing routine overhead. In order to reduce such overhead, the hashing algorithm has been changed to improve the uniqueness of the resulting hash [84]. This allows to have enough confidence in the hashing system to be able to skip the string comparison and save some overhead time. The previous system allowed for a pointer system to be used to immediately find the correct timer after the first use, skipping any testing other than the associativity of that pointer. This however was not pos-

sible to keep implemented as the new call-path based structure means that a timer exists multiple times in different call paths.
Another improvement consists in the fact that, for each timer, the standard deviation of the measured times across all processes is added to the printed information. This entails a few global communications to compute the standard deviation, however this additional cost is not too detrimental to the code global performance as they are performed only before displaying the data every n_{it} iterations. This information allows to quickly spot ill balanced regions.
This new timing structure has also allowed the creation of a rudimentary profiling mechanism that can be activated at runtime by the user from the input file. Several time measurements (iteration time, global time, RCT, ...) and statistics (min, max, standard deviation, average time) for each timer are stored for each call during execution and saved in a dedicated file in *json* [85] format to be later exploited in Python for example.
One of the flaws of this timing structure is that it does not deal well with recursive calls. In that case a child would be created for each recursive call, resulting in an possibly large and memory consuming structure. A possible solution to this problem could be the application of techniques such as the one proposed by [86]: keeping track of the context, i.e. the active set of subroutines at the time of the timer call, allows to create a finite number of such context even for recursive programs. Recursivity however, at the time of this work, in not widely used in YALES2, with very few exception in routines with such low execution time that their individual timing is not indicated anyway, due to the relatively high timing overhead compared to the routine time.

2.2.4 Code characterisation

The peculiar characteristics of the AMD Epyc 7H12 processor allow, through process placement, to characterise to some extent the dependency of the code to memory bandwidth, cache size and clock frequency. With a constant number of processes it is indeed possible, changing their placement to change the memory bandwidth and the quantity of cache memory available to each process independently. The following measurement tries to characterise the YALES2 DPCG solver by performing measurements of the 14M elements Preccinsta benchmark on 32 cores with different process placements, as shown in Table 2.2. The decision of removing the MPI contribution to this characterisation is motivated by the fact that it is extremely dependent on the number of processes, hence the characterisation itself wouldn't be useful for another core count. A process placement G/S means that a contiguous group of G processes is bound over a stride of S cores, and such configuration is replicated until the total number of requested processes is placed on the node. Regarding the cases exposed in Table 2.2 this means that for Case C1, the 32 processes are bound to the first 32 cores of the node, leaving the other 32 empty, as in Figure 2.20. In Case C2, out of the 8 cores of each CCD, only 4 of them are used to process. As the memory bandwidth is the same for each CCD, using only

Case	Placement	Effect	Tot. time	MPI time	Comp. time $\mathcal{T}$
C1	32/64	Standard	103.0 [s]	24.72 [s]	78.28 [s]
C2	4/8	2 x BW	80.20 [s]	20.85 [s]	59.35 [s]
C3	2/4	2 x Cache & BW	69.20 [s]	19.72 [s]	49.48 [s]

Table 2.2: Effect of process placement on cache, bandwidth and execution time.

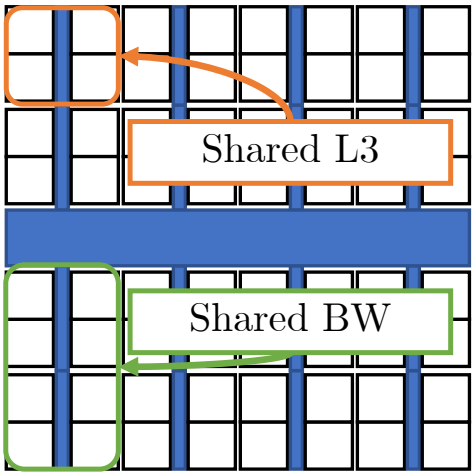


Figure 2.19: Schematic representation of the the AMD Epyc node.

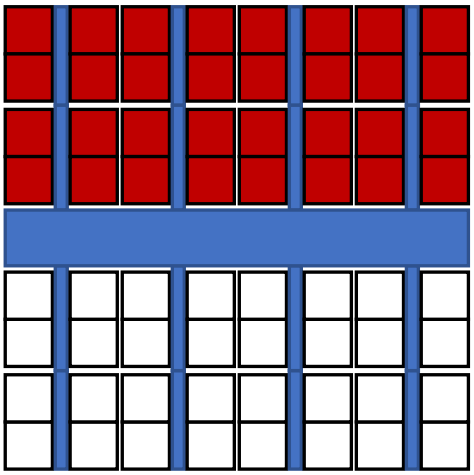


Figure 2.20: Schematic representation of the C1 case. Active cores in red.

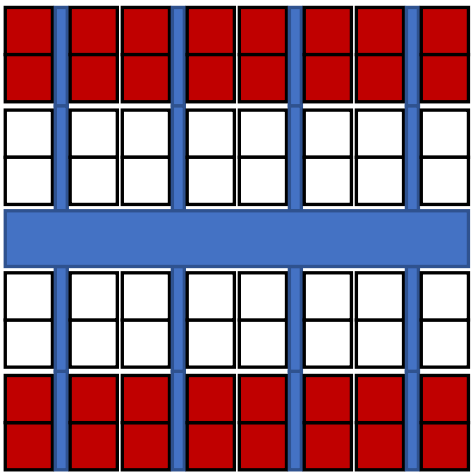


Figure 2.21: Schematic representation of the C2 case. Active cores in red.

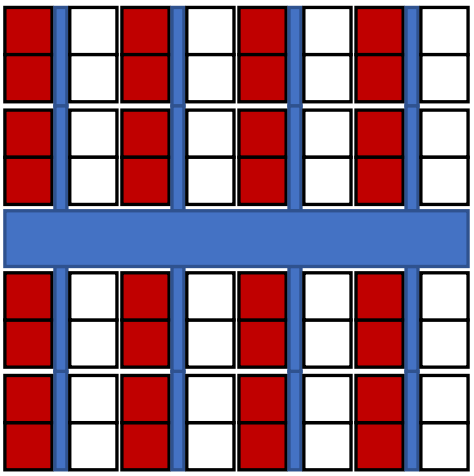


Figure 2.22: Schematic representation of the C3 case. Active cores in red.

Characteristic	Dependency [%]
Bandwidth	48.4
Cache	33.3
Frequency	18.3

Table 2.3: YALES2 code characterisation on the AMD Epyc 7H12 processor.

half of the cores allows to double the memory bandwidth available to each core. It is important however that these cores belong to the same CCX, as to keep the available L3 memory constant with respect to Case C1, as depicted in Figure 2.21. Finally, Case C3 uses the same amount of core per CCD, however processes are bound to only 2 of the 4 cores of each CCX, meaning that the L3 memory available to each core has doubled with respect to Case C1 and C2, while keeping the same bandwidth as C2. This last case is shown in Figure 2.22.
Using these three measures, the dependency of the code to cache memory, memory bandwidth and processor frequency can be computed as Equations 2.5:

$$\begin{aligned} Bandwidth &= 2 \times \frac{(\mathcal{T}_{C1} - \mathcal{T}_{C2})}{\mathcal{T}_{C1}}, \\ Cache &= 2 \times \frac{(\mathcal{T}_{C2} - \mathcal{T}_{C3})}{\mathcal{T}_{C2}}, \\ Frequency &= 1 - (Bandwidth + Cache)\,. \end{aligned} \tag{2.5}$$

The expressions in Equations 2.5 can be explained as follows. The difference in computation time $\mathcal{T}_{C1} - \mathcal{T}_{C2}$ from case $C1$ to case $C2$ is supposed to be solely due to the doubling of the available bandwidth per core, as all other parameters remained unchanged between the two simulations. The fraction of the computation time limited by the memory bandwidth can be computed as the ratio of the difference between the two cases and the original computation time, corrected by a factor equal to the bandwidth increment (in this case 2). A similar reasoning is applied for the increase of cache memory between $C2$ and $C3$. Finally, it is supposed the the only remaining factor influencing the execution time is the clock frequency of the CPU. The results for the YALES2 code are synthesised in Table 2.3. This characterisation shows that the code performance is highly dependent on memory ($\sim$80%) and much less on the computational power. This is in line with the expectations as the linear operators on which the code is based have a particularly low arithmetic intensity. This might have important consequences as reducing the computational power in order to increase cache and bandwidth available per process can possibly improve performance and scalability and even reduce the execution time.

2.2.5 Performance measurements

As a primary step for the present work it is paramount to analyse in detail the performances of the YALES2 code and more in particular of the Poisson solver.

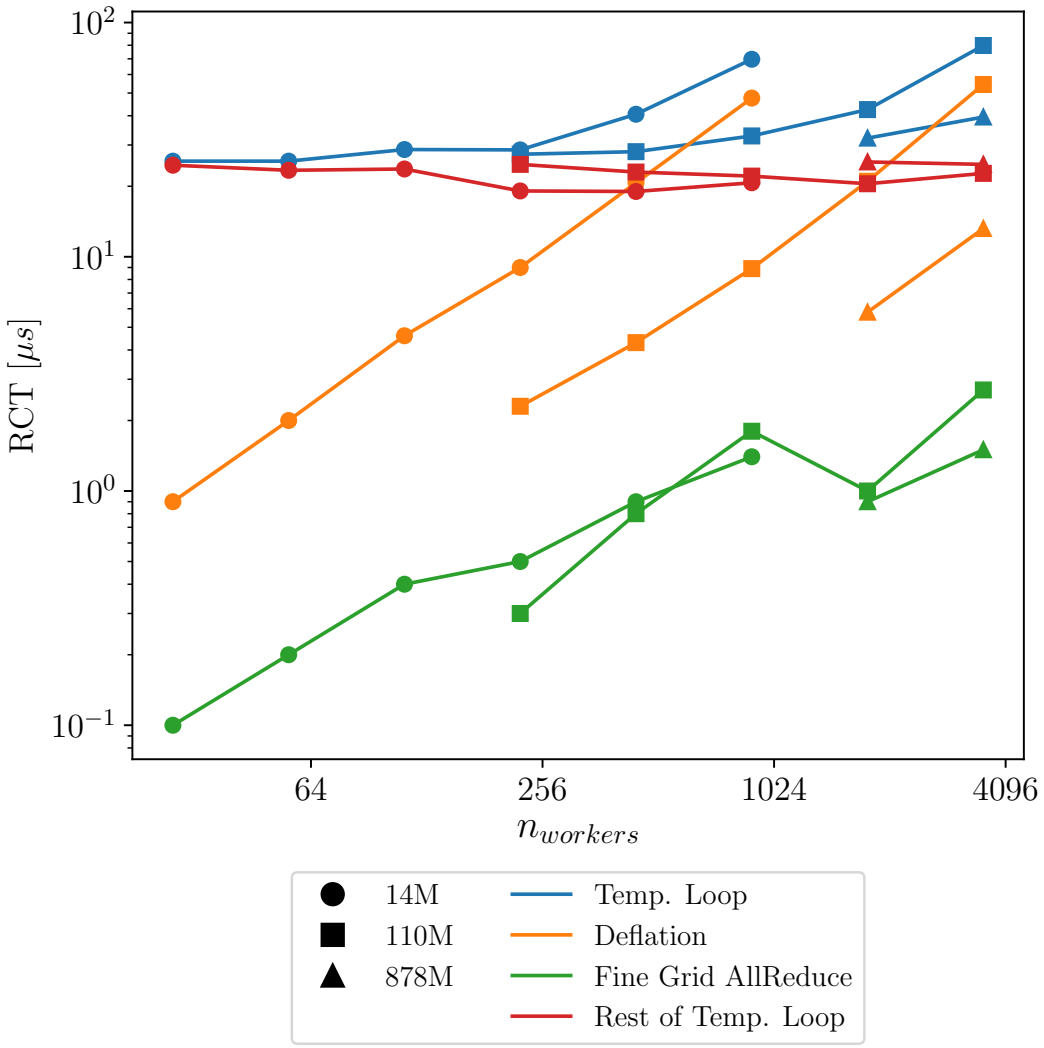


Figure 2.23: Preccinsta scalability breakdown on the MYRIA platform

The particular focus of this work on the cost of collective communications and in the deflation part of the DPCG algorithm can easily be explained by Figure 2.23. It shows a scalability study for the Preccinsta burner in YALES2 on the MYRIA platform. Together with the total execution time of the temporal loop, represented in blue, there is also its breakdown in three main parts. The time spent in the collective communication of the fine grid solution and on the deflation part of the DPCG algorithm are represented respectively in green and orange, while the rest of the temporal loop is shown in red. It is clear than that the loss in performance of the entire solver is due to the deflation and, in a more marginal extent, to the collective communication in the PCG algorithm. The rest of the code scales quite well, even at the most extreme cases, where the domain size per worker is quite small and its contribution to the total time becomes smaller than the deflation alone.
In order to better understand how to improve the performances of the DPCG al-

gorithm, several measurements have been performed, varying some of the most influential parameters for this implementation. Up to a certain extent, this was done already in [72] for the convergence criterion and the number of recycled solution to be used for the initial guess of the residual. These two parameters have not been investigated in this work, since an optimum value has already been found. In order to simplify the model, the usage of recycled solutions was deactivated. This leads to sub-optimal performances during the study, however there is no reason to think that its beneficial effects would somehow disappear after other improvements are put in place.
Arguably the most influential parameter on the code overall performances is the size of the group of elements. This is determined by setting the NELEMENTPERGROUP parameter in the input file. In the remainder of the manuscript, NELEMENTPERGROUP can be shortened as NEPG for the sake of brevity. As seen in Subsection 2.1.5, YALES2 has the group of elements resulting from the DDD as the cornerstone of its entire data structure. Furthermore, it uses them in order to construct the deflation grid. The main advantages of this approach are that there is no need to create a new coarse grid and the deflation operators can be easily computed. It entails however much deeper consequences, which will be analysed in the following.
The measures shown in this subsection were obtained through a strong scalability study of the 14M elements Preccinsta benchmark on the MYRIA platform. For each number of processes used, different values of the NELEMENTPERGROUP parameter were tested.
A few conventions for the terminology adopted in the rest of the analysis are defined here. The notation n_x stands for the quantity (number) of x. A certain entity x noted without any subscript indicates that it is proper to single worker and it includes all its duplications, if any. The subscript $[g]$, short for global, indicates the amount of a certain entity on a single worker, with the duplicated entities counted only once. The subscript $[t]$, short for total, indicates the amount of a certain entity on all workers, including all its duplications. Finally, the subscript $[gt]$ indicates the amount of a certain entity on all workers, but with the duplicated entities counted only once. The notation $[gt]$ is used also for those quantities that do not have duplicates such as the elements and the ElGrps.

2.2.5.1 Grid characteristics

The first immediate consequence in varying the size of the group of elements is the variation in the total number of groups $n_{ElGrp_{[gt]}}$. Clearly, on a same mesh, increasing the size of the groups implies a reduction on their number, and vice-versa, as depicted in Figure 2.24. Such variation is indeed independent of the number of workers used for the computation, as confirmed by the bottom plot of the figure. The second implication, is the variation in the number of nodes and pairs on each worker's mesh. This is due to the fact that nodes and pairs on the boundary between the different groups are duplicated. In Figure 2.25, the top graph shows that the ratio

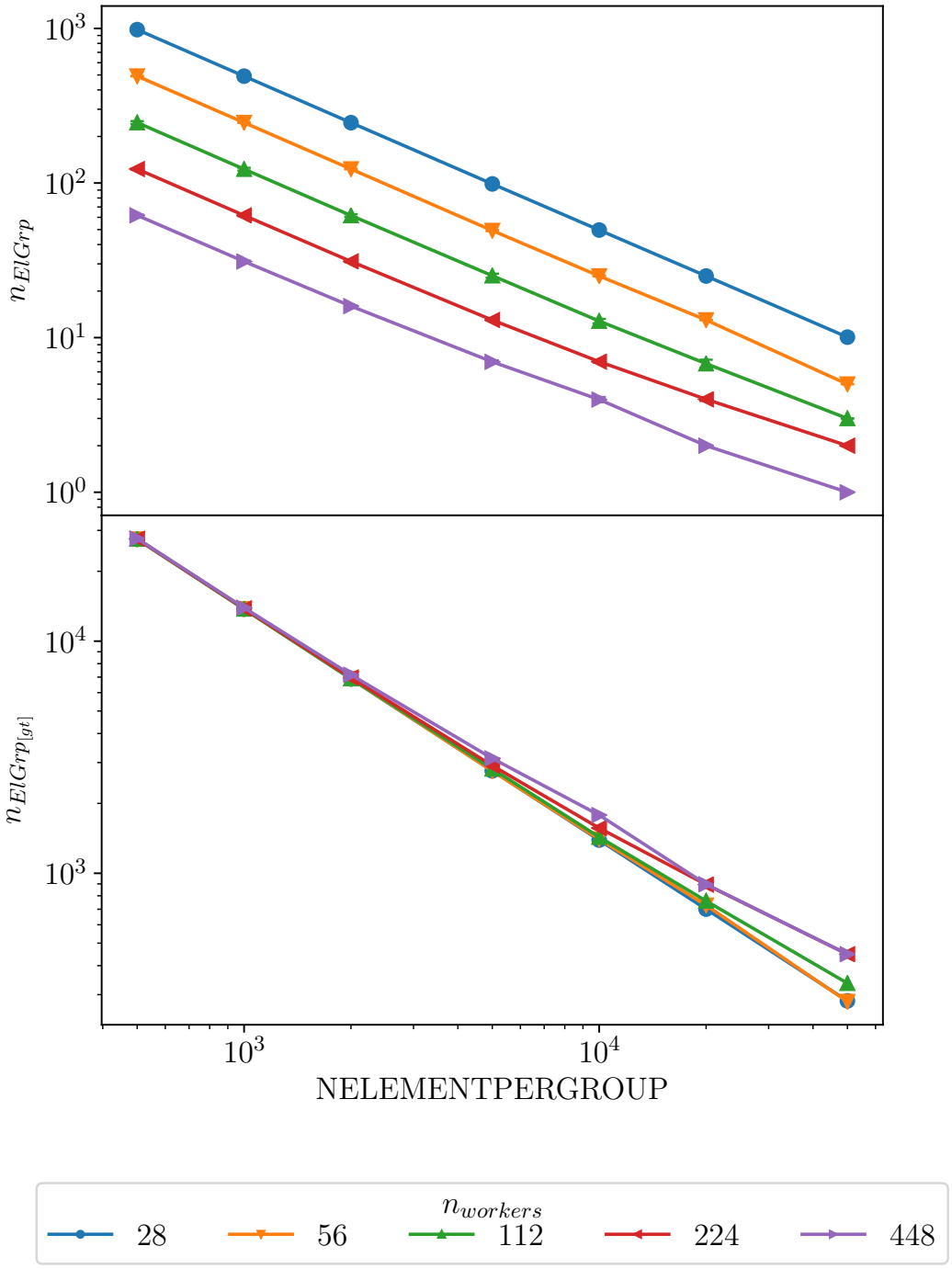


Figure 2.24: Analysis of the variation of the number of groups with NELEMENTPERGROUP on Preccinsta 14M

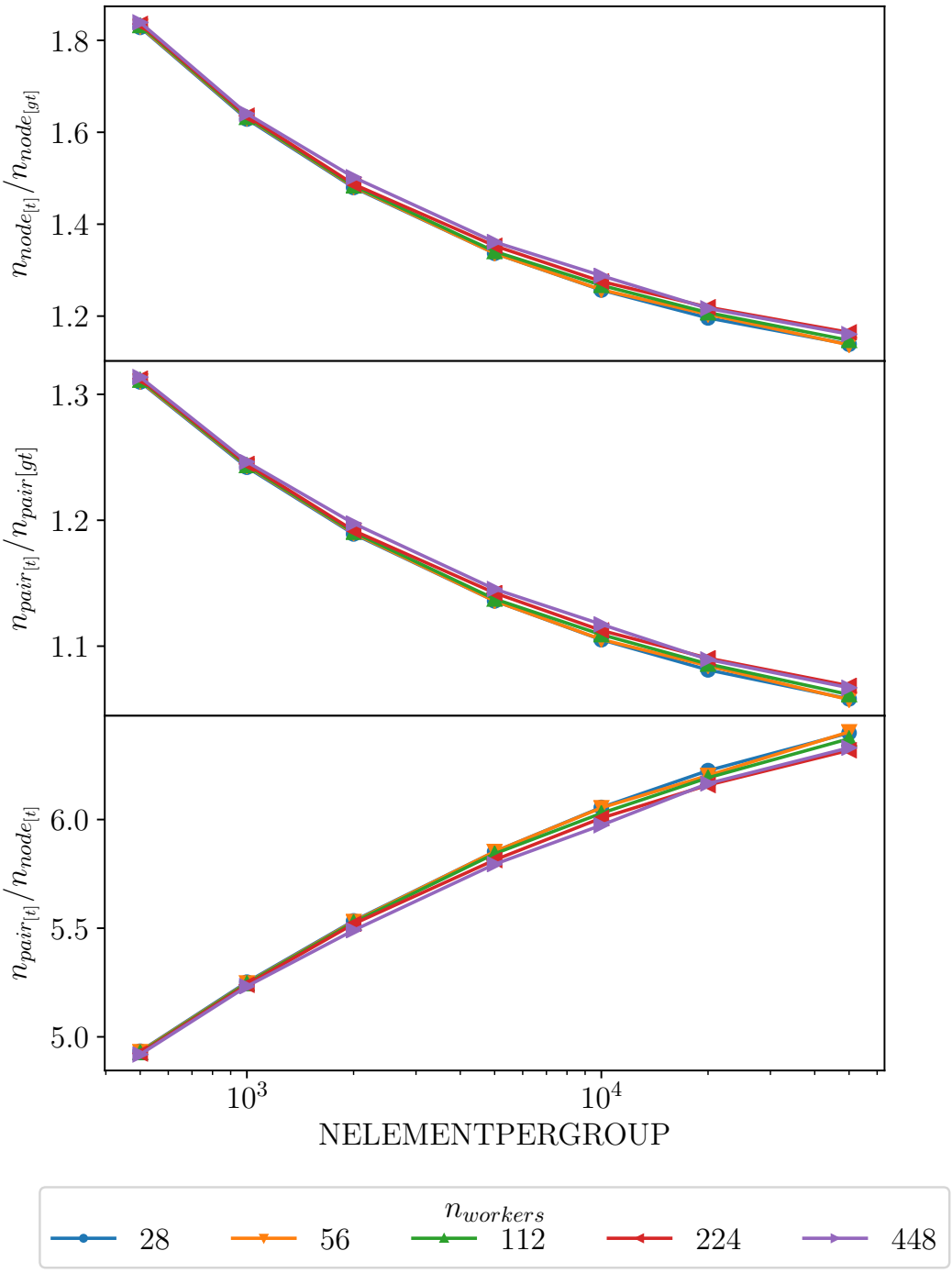


Figure 2.25: Analysis of the variation of the ratio of unique and duplicated nodes and pairs with NELEMENTPERGROUP on Preccinsta 14M

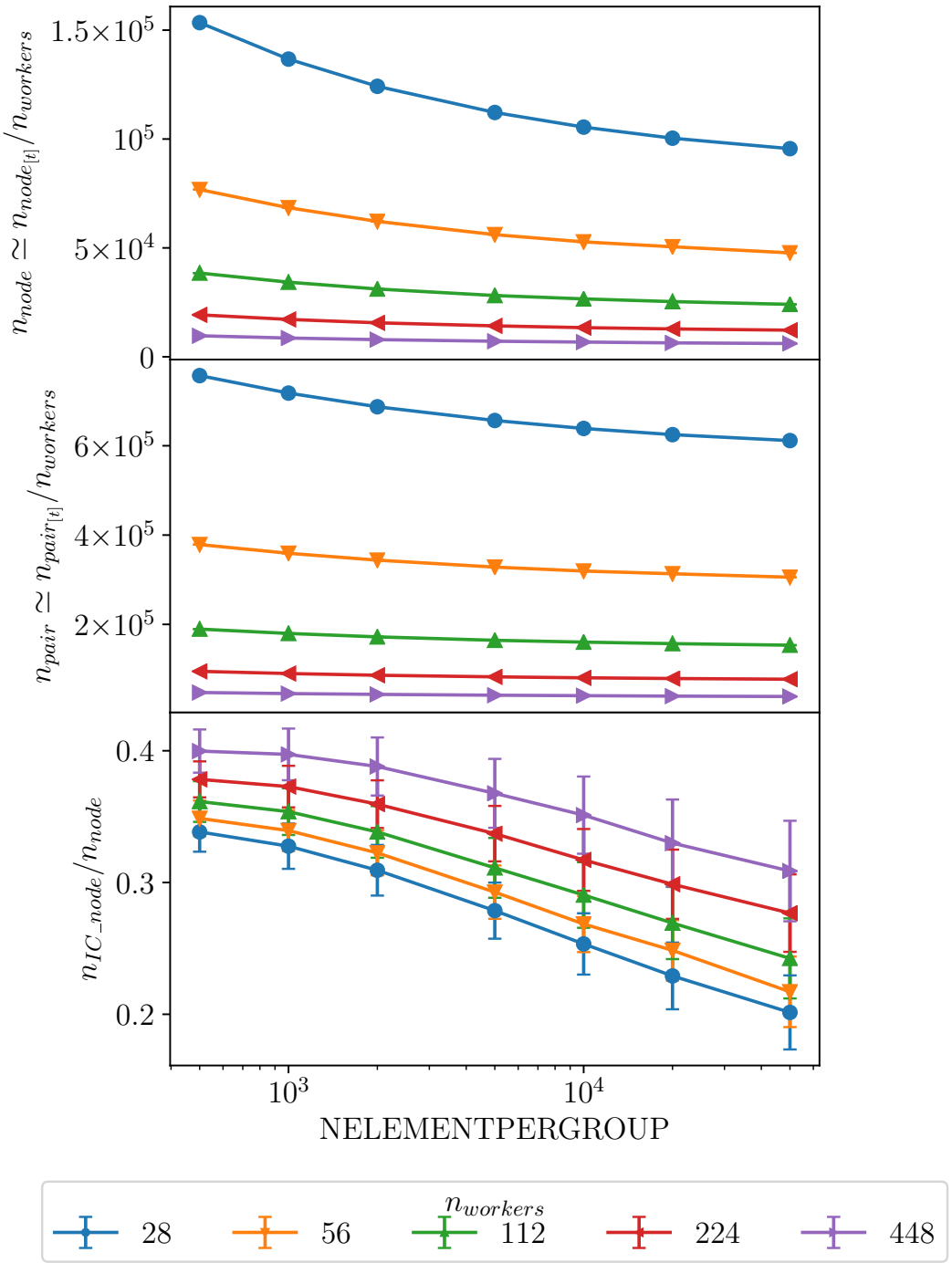


Figure 2.26: Analysis of the variation of the total number of nodes and pairs and size of the internal communicator with NELEMENTPERGROUP on Preccinsta 14M. The error bars represent the standard deviation across all workers

between the sum of the nodes of each group $n_{node_{[t]}}$ and the number of unique nodes $n_{node_{[gt]}}$ decreases dramatically when the size of the groups increases, independently from the number of worker used in the computation. The same behaviour is observed for the pairs, however, less prominently and with a different ratio. This is confirmed by the graph on the bottom of the picture, which indeed shows that the ratio between total amount of pairs and nodes increases slightly with the size of the groups. This could be a problem as the loops on nodes are vectorised, while loop on pairs are not. However the difference is so minimal that the effect is actually negligible. The two central top plots of Figure 2.26 however, show that the reduction in the total amount of nodes and pairs per process is extremely important, especially for the lower numbers of workers. In practical terms this directly translates into a considerable reduction in the required amount of computation that is performed by the different workers. The graph on the bottom of the same figure shows the relation between the size of the internal communicator for the nodes and the total amount of nodes (with duplication) on each worker. The internal communicator is a fundamental piece in the data structure of YALES2, however the update of the IC is a supplementary operation which is not necessary for computation. Consequently it could be argued that the IC update is an overhead, and as such should be reduced. It is possible to remark that increasing the size of the groups has a beneficial effect in reducing such overhead, as the ratio between the size of the IC and the total number of nodes decreases from $\approx 34\%$ to $\approx 20\%$ for $n_{workers} = 28$ and from $\approx 40\%$ to $\approx 32\%$ for $n_{workers} = 448$. The gain on the cases with small number of workers is extremely important considering that n_{node} decreases much more than for the other cases as mentioned above.

2.2.5.2 Number of iterations in the DPCG solver

The following and most important consequence is that the performance of the deflation phase and that of the rest of the DPGC algorithm are indissolubly linked by the size of the groups. Changing the groups size, hence their number, has an opposite effect on the two parts of the algorithm. Figure 2.27 shows the average time needed for the convergence of the DPCG algorithm as a function of NELEMENTPERGROUP and $n_{workers}$, while Figure 2.28 shows the percentage breakdown of this time in three phases: the time spent on the fine grid, on the deflation and initialising the DPCG. It is clear that the solver initialisation is negligible and therefore will not be analysed further. It is interesting to notice how the proportion of time spent in the deflation increases dramatically with the number of workers, confirming the poor scalability behaviour exposed in Figure 2.23. The two parts of the algorithm, namely the iteration on the fine grid and the deflation behave very differently and consequently a separate analysis is performed for each of them. Figure 2.29 shows the number of iterations on the fine grid and the ratio of number of iterations on the coarse grid and those on the fine one varying NELEMENTPERGROUP. These are computed as a mean value over 200 time-steps of the solver, with the error-bars indicating the standard deviation among the different time-steps. On the bottom

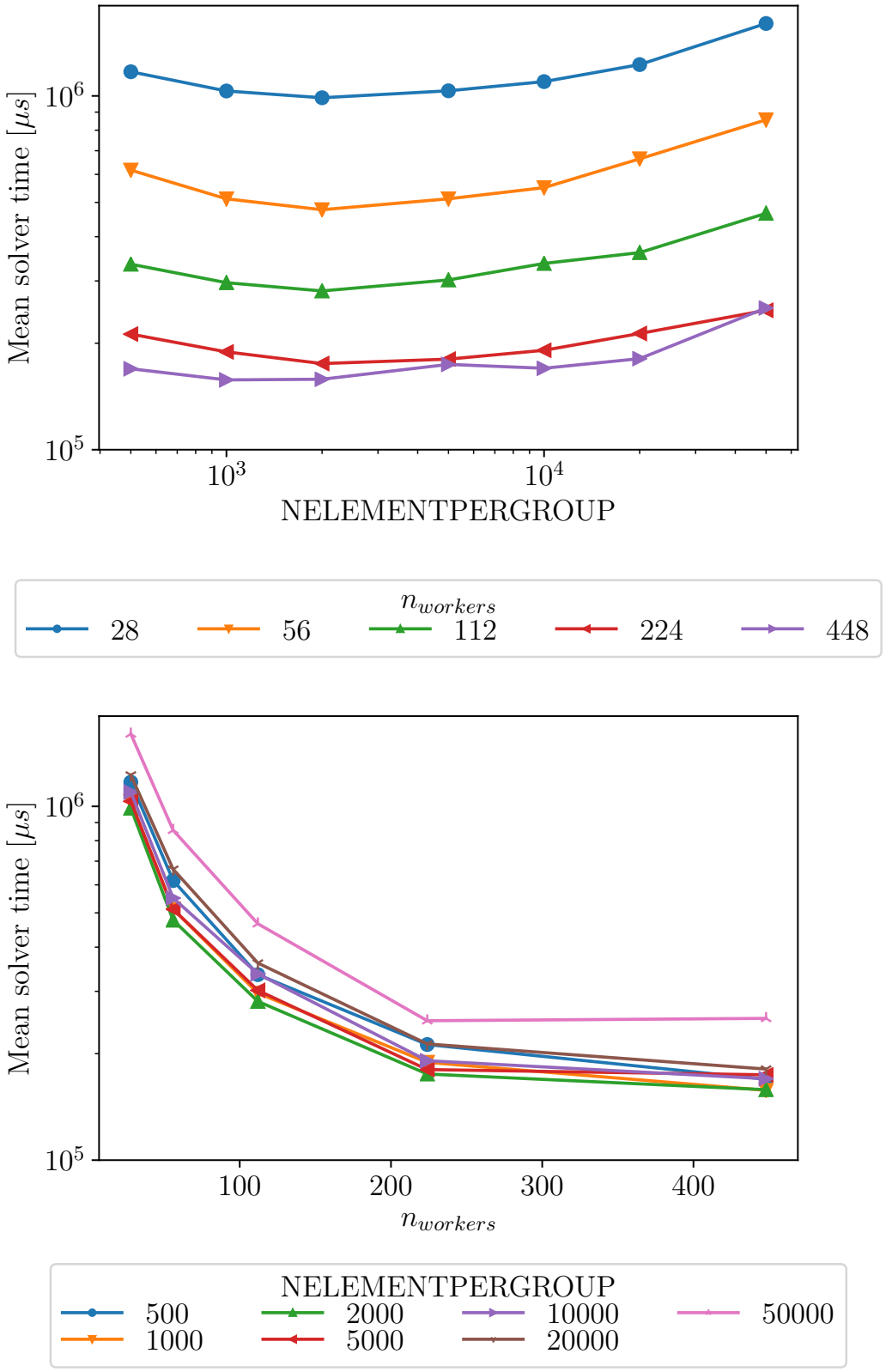


Figure 2.27: DPCG solver time as a function of NELEMENTPERGROUP and $n_{workers}$ for Preccinsta 14M on MYRIA

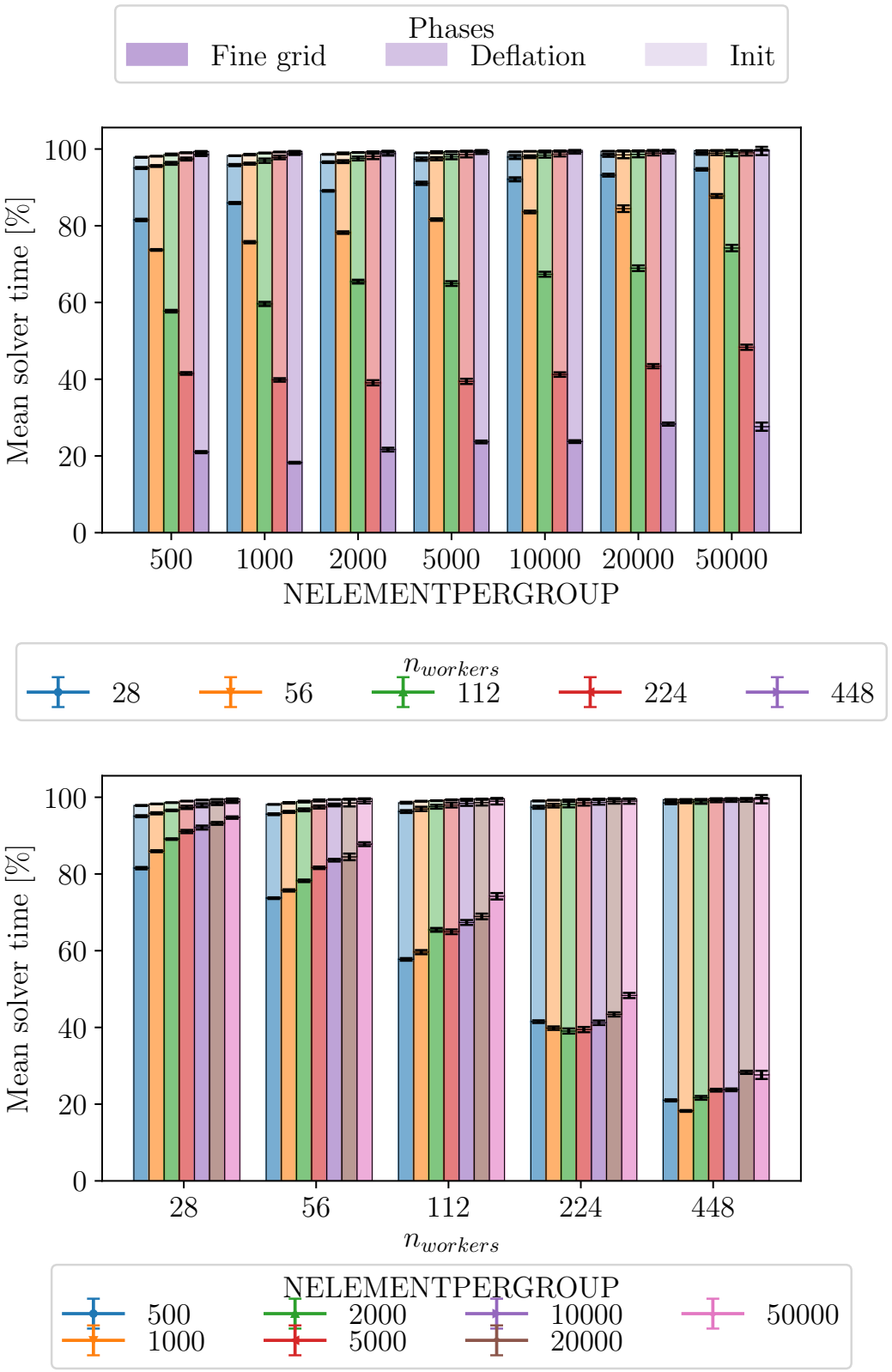


Figure 2.28: Breakdown of the DPCG solver time into fine grid, deflation and initialisation for Preccinsta 14M on MYRIA. The error bars represent the standard deviation of the measured time across the different workers

plot, the average total amount of deflation iterations is shown. This is simply computed as the product of the average amount of iterations on the fine grid and the average number of deflation iterations per fine grid iteration. Figure 2.29 shows that increasing the size of the groups, hence coarsening the deflation grid, allows for a faster convergence. However, the precision of such solution decreases, consequently more iterations are needed on the fine grid to arrive at its final converged state. It is important to remark that the number of iterations necessary to converge the DPCG solver are independent from the number of workers. The small differences are caused by numerical rounding errors mainly in operations related to inter-process communications. The total amount of deflation iterations remains approximatively constant, with a slight tendency to increase for bigger group sizes. The only exception is for 440 workers with a group size of 5′0000 elements. This case, in which each process has only one ElGrp, hence only one point in the deflation mesh is clearly an extreme situation, in which the PCG on the deflation seems to need more iterations to converge.
Experience with the code showed that the the optimal value for NELEMENTPERGROUP is between 2'000 and 5'000, and confirmation is found in Figure 2.27. In order to understand this, a more detailed analysis of the cost of a DPCG iteration is needed. As for the number of iterations, the cost analysis is split into two separate parts, considering the time spent in the deflation separated from the rest of the algorithm. As a reminder, this means that the time spent in the *Solve* $\hat{A}d_{k+1} = W^T \left(AK^{-1} - I\right) r_{k+1}$ step of Algorithm 3 is measured separately and its cost is removed from the rest of the algorithm. That deflation step is solved with the PCG algorithm 6 on the coarse grid.

2.2.5.3 Fine grid iteration

First it is interesting to look at the analysis of the average time needed to perform an iteration on the fine grid, excluding the time spent on the deflation step. The same data is represented in Figure 2.30 as a function of the group size (NELEMENTPERGROUP) and the number of processes used to perform the computation $n_{workers}$. Figure 2.31 shows a percentage breakdown of the total iteration time in various sub-categories: computation on nodes, computation on pairs, collective communication, P2P communication and the remainder of the algorithm.
The top graph of Figure 2.30 shows how, increasing the size of the groups helps reducing the total cost of each iteration, particularly for the cases with a lower number of workers. There is an evident link between this graph and those of Figure 2.26. The explanation is found looking at the percentage breakdown. It is clear that the largest contribution to the cost of an iteration is given by the two computation phases, as these make up for about 80% of the iteration time for all those cases computed with 28, 56 and 112 processes, and respectively 70% and 60% for 224 and 448 processes. On the bottom plot of Figure 2.30, scalability curves are obtained for each value of NELEMENTPERGROUP. While the runtime is higher, scalability seems to be slightly better for the lower values of the parameter. Another

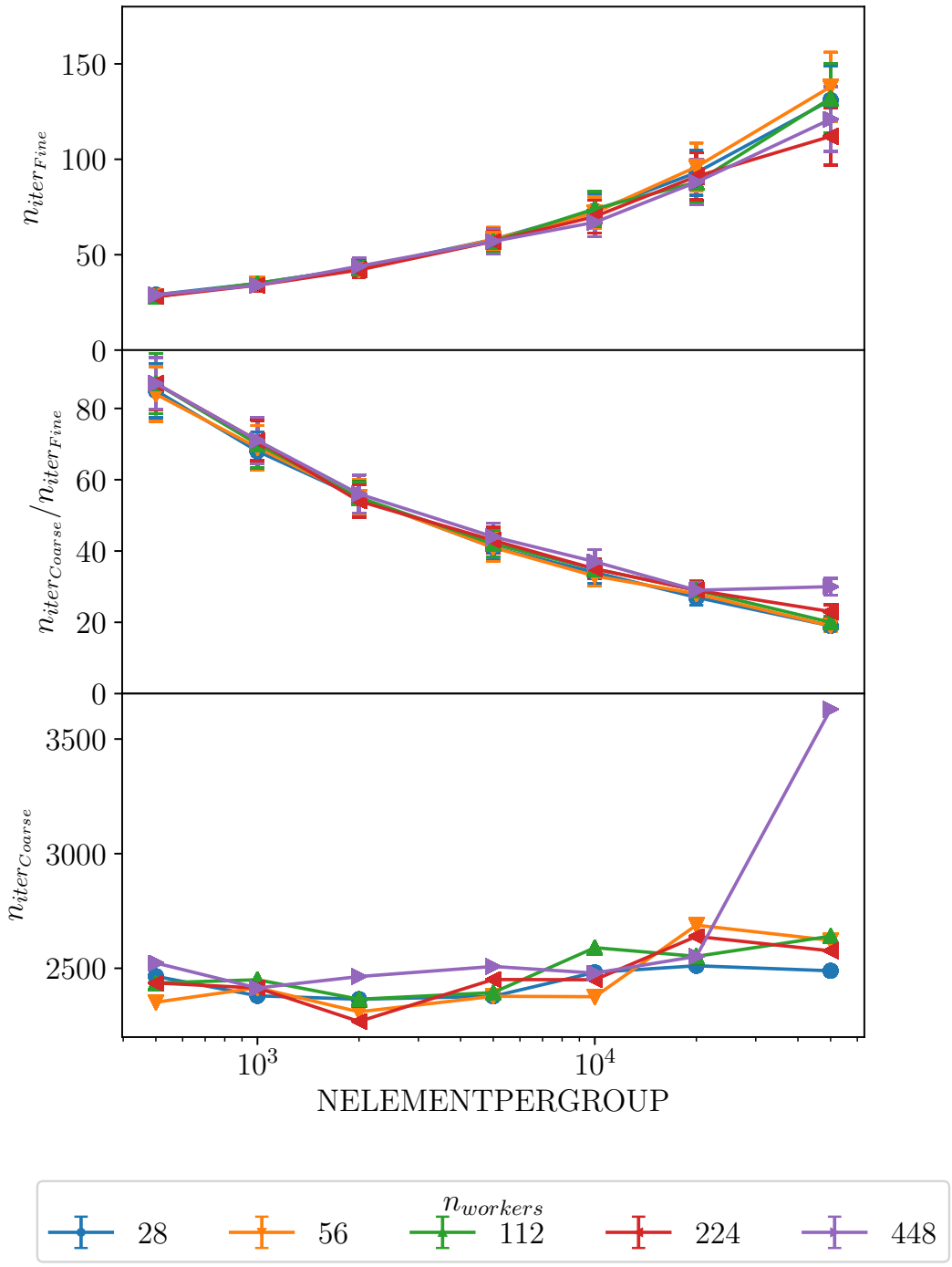


Figure 2.29: Analysis of the number of iterations for the convergence of the DPCG algorithm as a function of the size of the groups for Preccinsta 14M. The measures were collected over several time steps and the error bars represent the standard deviation on the iteration numbers across all time steps

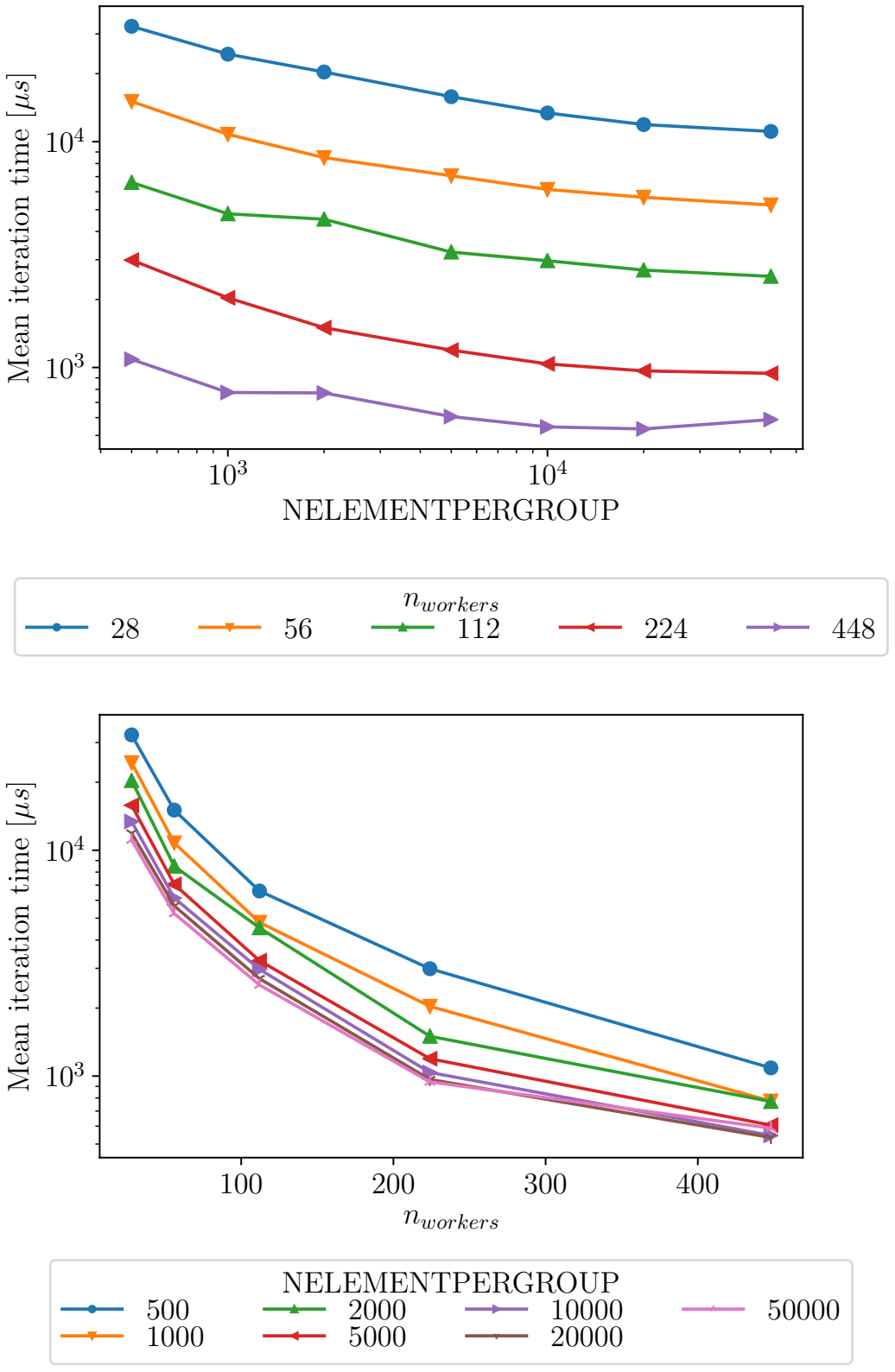


Figure 2.30: Fine grid iteration time as a function of NELEMENTPERGROUP and $n_{workers}$ for Preccinsta 14M on MYRIA

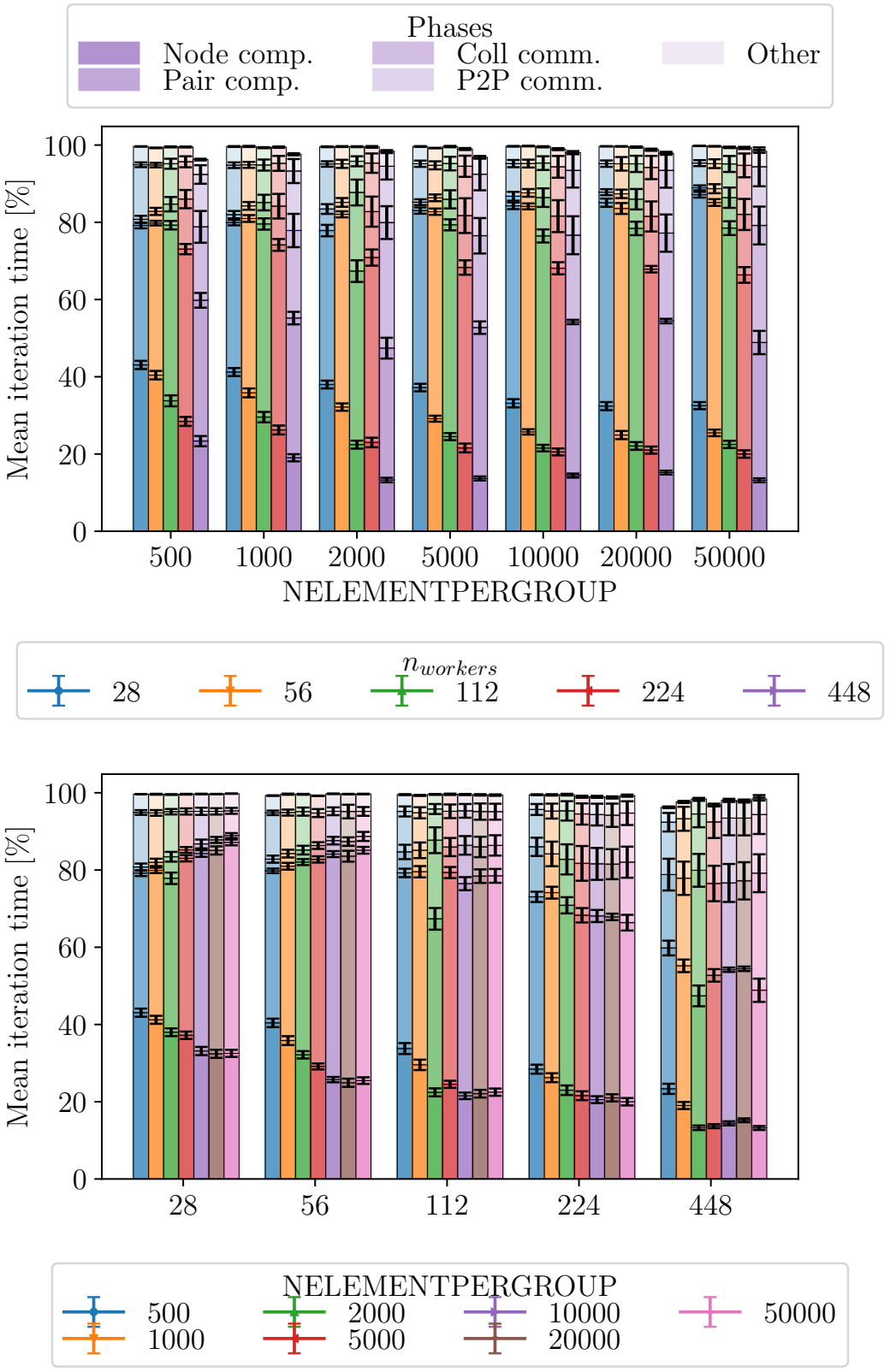


Figure 2.31: Breakdown of the fine grid iteration time into different phases for Preccinsta 14M on MYRIA. The error bars represent the standard deviation of measured time across all processes

interesting thing to notice is that the cost of point-to-point (P2P) communications remains quite limited and its percentile contribution does not increase much with the number of processes. In addition, increasing the size of the groups reduces the percentage of the time spent on these communications. On the other hand, as it is to be expected, collective communications become gradually more expensive while increasing $n_{workers}$. This observation is perfectly coherent with what previously shown in Figure 2.23. The error bars on the percentile breakdown graphs represent the standard deviation across the workers for each phase. These are useful to understand the load balancing of each phase. While the computation phases are almost perfectly balanced for all cases, a higher number of workers corresponds to an increase in the standard deviation of the communication phases. This means that the increase of communication cost, especially for collective ones, is only in part due to a larger number of processes. The increasing imbalance in other parts of the algorithm also contributes to the loss of efficiency, as they are blocking collective communications, which implicitly impose a synchronisation point for all processes.

2.2.5.4 Deflation iteration

A similar analysis is displayed in Figure 2.32 and Figure 2.33 for the deflation iteration. Here however, the computation time is extremely low, above 20% only for a couple of cases. Collective and point to point communications are the main contributors to the total iteration cost. Increasing the group size helps to reduce the cost of the deflation iteration only for those cases with 28 and 56 workers. This is due to the fact that, since the mesh is the same, in these cases each worker has a higher number of groups on which to perform computation with respect to the other. While it is reasonable to assume that the cost of communication depends mainly on the number of workers, analysing the percentage breakdown shows that increasing NELEMENTPERGROUP has the effect of reducing mainly the proportion of time spent on the computation on the pair of groups. The computation on the groups is much lower due to the fact that it's fully vectorised, while the computation on the pair of group needs an indirection operation which prevents vectorisation. Reducing the number of pairs has then a stronger influence on the computation time. Together with the percentile breakdown, the standard deviation across the different processes is also displayed for each group. It is important to remark the large values of the standard deviation for the two communication phases and the relatively large one of the edge computation phase. The explanation for these large discrepancies in time measurement among processes must be searched in the way the connectivity among groups and processes is created for the deflation grid.
As explained in 2.1.5.1, processes with higher colour build a deflation grid that also includes halo groups from their neighbours with lower colour. The matrix-vector multiplication $s = Aw$ is performed according to Equation 2.2, i.e. with a loop on the pair of groups, including those with the extra halo groups. This means that workers with higher colour have to compute on a larger number of pairs, which accounts for relatively high standard deviation in this computation phase. The communication

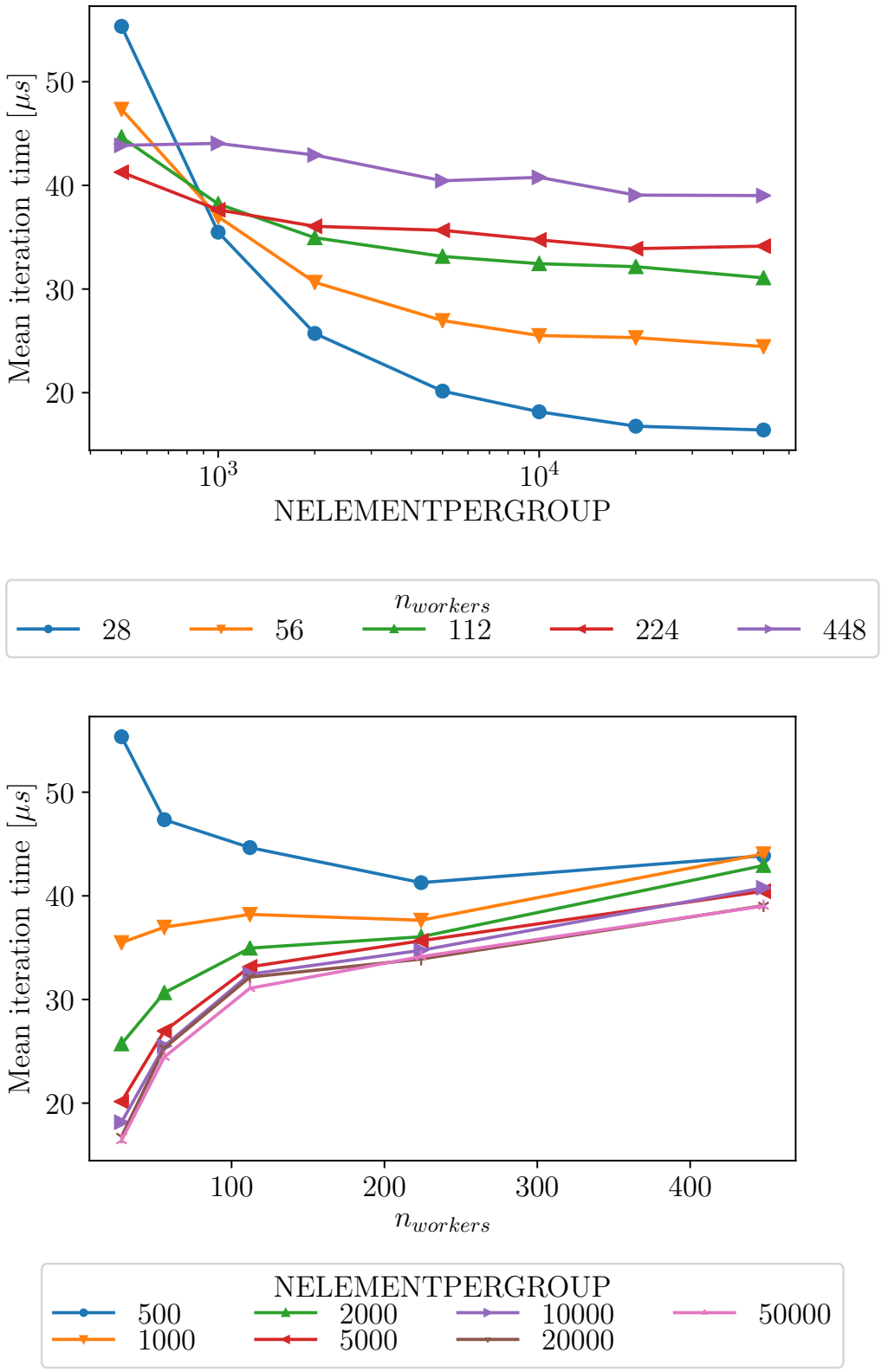


Figure 2.32: Deflation iteration time as a function of NELEMENTPERGROUP and $n_{workers}$ for Preccinsta 14M on MYRIA

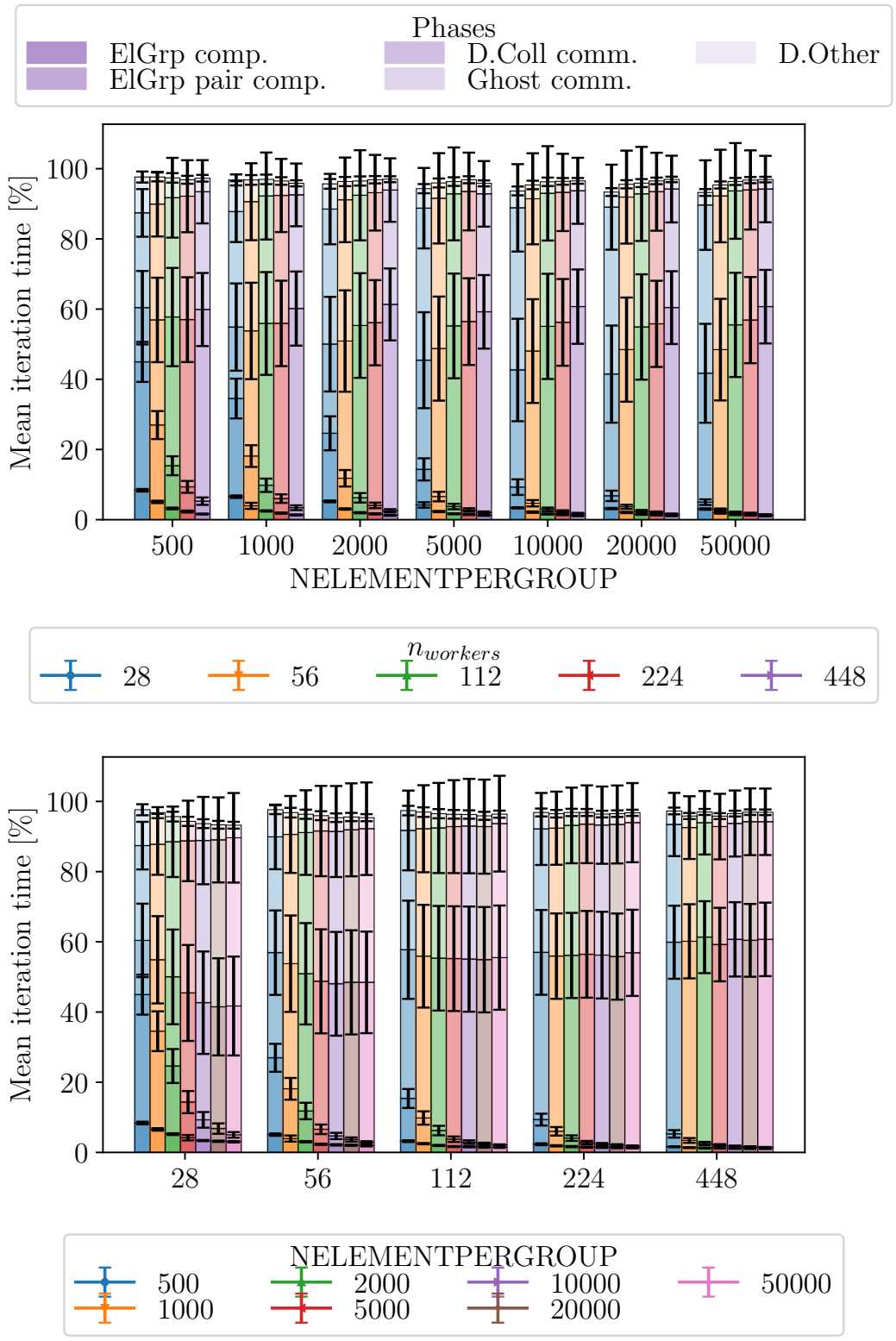


Figure 2.33: Breakdown of the deflation iteration time into several phases for Preccinsta 14M on MYRIA. The error bars represent the standard deviation of measured time across all processes

phases however show a standard deviation that is much more important. This again is possibly due to the half-halo system. It creates indeed a communication pattern that highlights possible delays and worsen the communication efficiency. With the reasonable assumption that processes are fairly synchronised after the computation of w, workers on the receiving side, must wait for their partners to pack and send the data. Once they receive the data they have to unpack it and compute s. As explained above, the amount of computation for these processes is higher than for the others. Finally they have to pack and send their data back to their neighbours, who were possibly already waiting due to their lower amount of computation. Another cause of the communication imbalance is the fact that there is a huge variance in the number of neighbours that each process has. A process that needs to communicate with many neighbours will be forcibly delayed with respect to one that has only a few. Figure 2.34 shows a trace obtained with TAU for a deflation iteration for the 14M elements Preccinsta benchmark on the MYRIA supercomputer. Although the time spent in the MPI functions is exaggerated by the instrumentation, it is clear that the P2P communication pattern and the difference in the number of neighbours play a key role on the high cost of the whole iteration. It is also possible to see the imbalance in the computation on the pair of groups between the two communication phases. Looking at Figure 2.34 it is clear how most of the cost of the collective communication is a direct consequence of the imbalance generated by the P2P communication and the computation on the pair of groups. To resume, this brief performance assessment has shown that:

- The solution of the PCG on the deflation grid and the collective communications for the solution on the fine grid do not scale at all.
- The rest of the code scales almost perfectly, at least up to 20'000 processes.
- As a consequence of the two previous points, the time spent solving on the coarse grid becomes preponderant for large numbers of processes.
- The performance of the code are driven by the value of NELEMENTPERGROUP, which has an opposite effect on the fine grid and on the deflation. To have bigger groups means to have less duplicated nodes and pairs, which reduces the cost of a single iteration on the fine grid, but at the same time it increases their number. On the other hand, the average number of iterations on the coarse grid per iteration on the fine one is reduced.
- The cost of the fine grid iteration is for the most part due to computation. The cost of the collective communications increases with the number of processes.
- The cost of the deflation iteration is mainly due to communication. The large standard deviation for the communication time across the different processes indicates that such phases are ill balanced.
- The large communication time in the collective communication is due to the bad pattern of the point-to-point exchange.

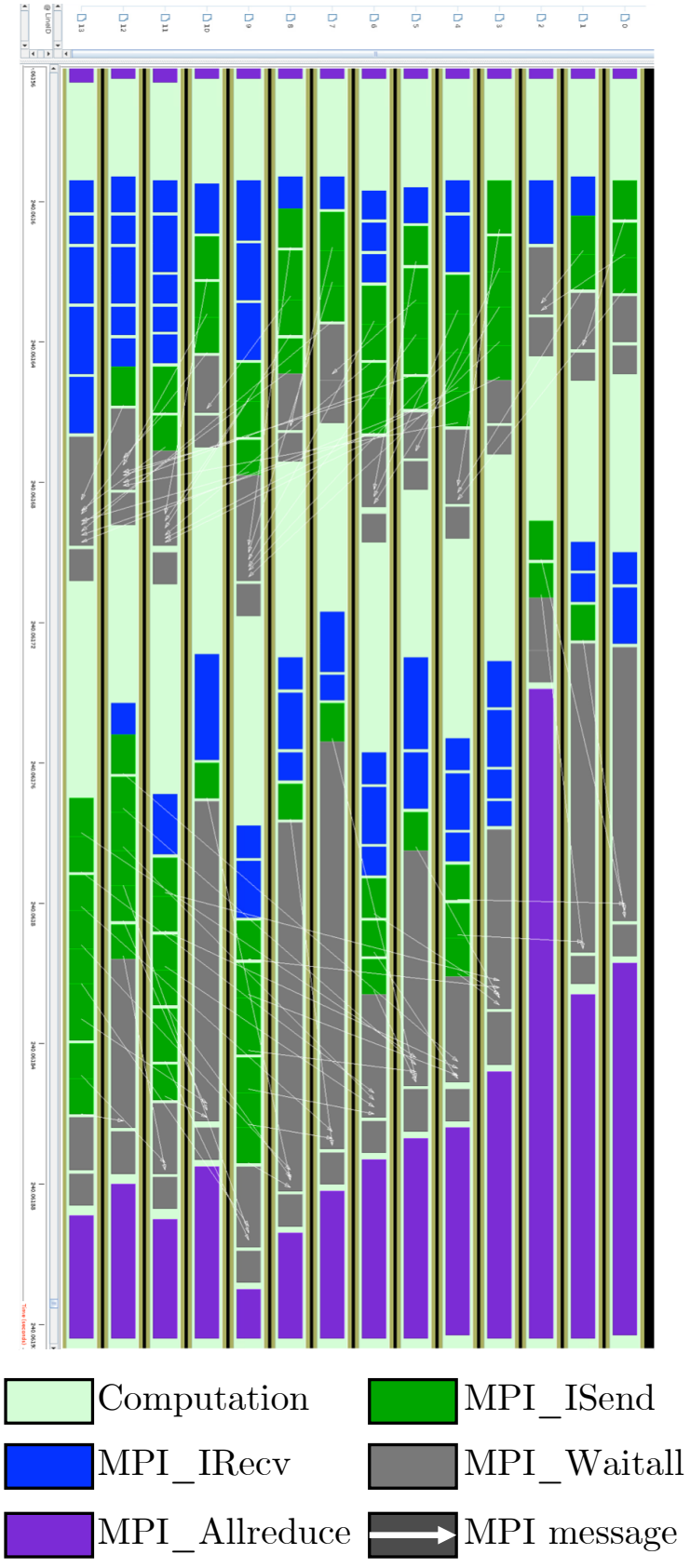


Figure 2.34: TAU trace of a deflation iteration showing the communication pattern and computation imbalance

These points will be thoroughly analysed in the following, with particular focus on the influence of the value of NELEMENTPERGROUP and $n_{workers}$ on the performance of each step of the algorithm.

2.2.6 Performance model

In this subsection a simplistic performance model is derived from the measurements exposed in Subsection 2.2.5. The performance model is voluntarily kept simple, as many variables could intervene in the performance of the solver and it would be extremely hard to account for everything. Also, the objective of this model is not to provide an exact prediction of the behaviour of the code, but only a summary indication of the influence of the most influential parameters on the final performance. For this reason, the model is almost exclusively composed by linear functions, whose coefficients are obtained performing a least-square method interpolation over the measurements presented above.

2.2.6.1 Grid characteristics

Starting from the grid characteristics, the most important parameters are the number of ElGrps per each worker n_{ElGrp}, the total amount of groups $n_{ElGrp_{[gt]}}$, the total amount of unique nodes $n_{nodes_{[gt]}}$ and pairs $n_{pairs_{[gt]}}$. All these quantities depend only on the total amount of elements $n_{elem_{[gt]}}$, the parameter NELEMENTPERGROUP, shortened NEPG and the number of workers $n_{workers}$. The number of nodes and pairs depend also on the type of elements that constitute the mesh. YALES2 is able to treat meshes with all kinds of elements: tetrahedra, hexahedra, polyhedra, etc. however all analysed cases, except if specified otherwise, are of meshes with tetrahedral elements. The two expressions for the number of groups can be derived directly with the formulations in Equation 2.6. The *ceil* expression, i.e. the rounding of the division to the upper integer, gives a more precise estimation of the number of groups because it follows the two-steps decomposition performed by the code: first the mesh is divided among the different workers, and then each of them divides its own portion of mesh in ElGrps. This distinction is important only in extreme cases where the value of NELEMENTPERGROUP is higher than the count of elements per worker. In such case, (at least) one group must be created per worker, consequently the minimum number of groups in a computation is equal to the number of workers.

$$\begin{aligned} n_{ElGrp} &= ceil\left(\frac{ceil\left(\frac{n_{elem_{[gt]}}}{n_{workers}}\right)}{NEPG}\right) \simeq \frac{n_{elem_{[gt]}}}{n_{workers} \times NEPG}\,, \\ n_{ElGrp_{[gt]}} &= n_{ElGrp} \times n_{workers}\,. \end{aligned} \tag{2.6}$$

The total number of unique nodes and pairs can be approximated as a linear function of the number of elements, as in Equation 2.7 and Equation 2.8.

$$n_{node_{[gt]}} \simeq \beta_{node_{[gt]}} \times n_{elem_{[gt]}}\,. \tag{2.7}$$

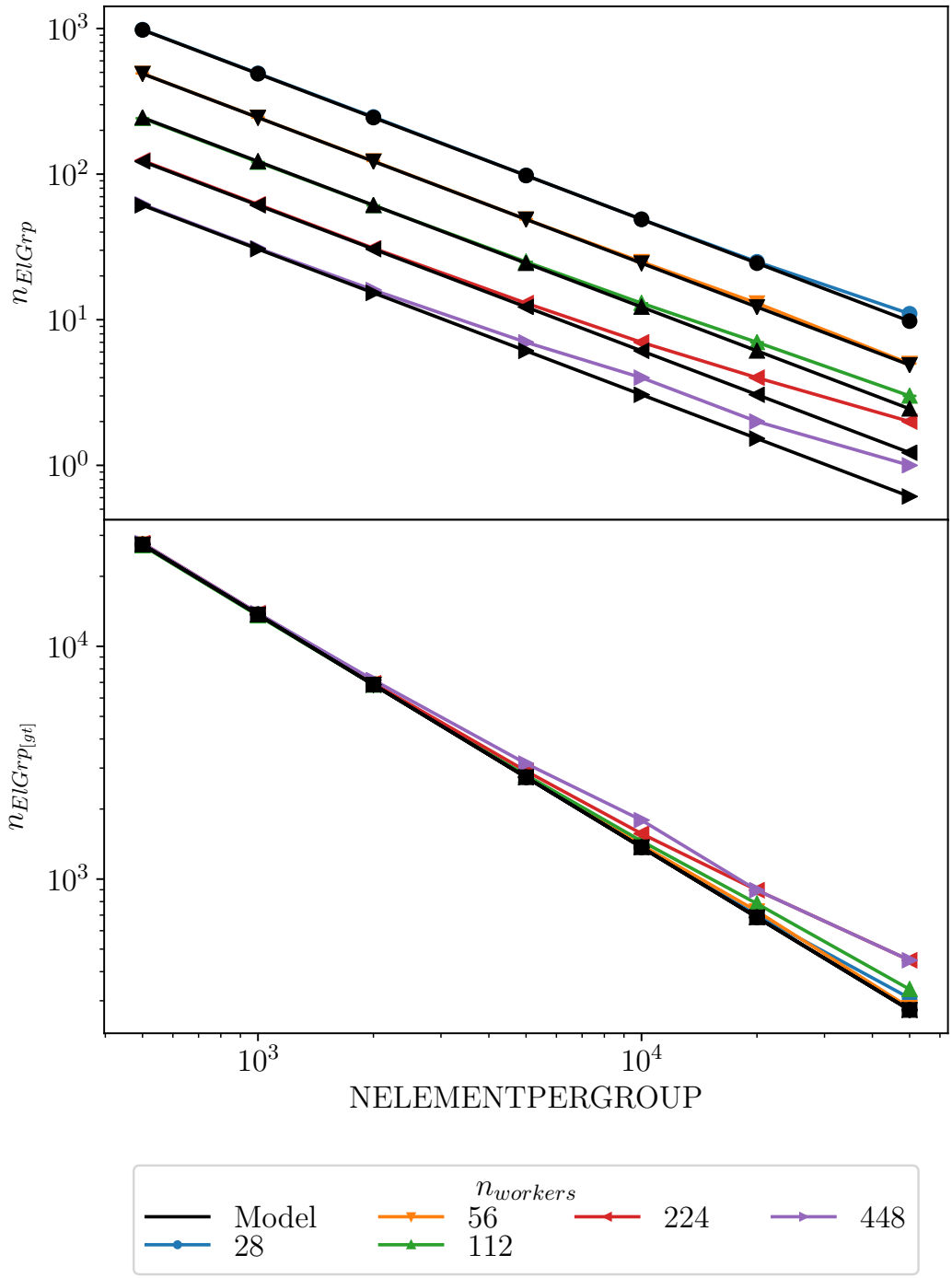


Figure 2.35: Models of the variation of the number of ElGrps with NELEMENTPERGROUP for Preccinsta 14M

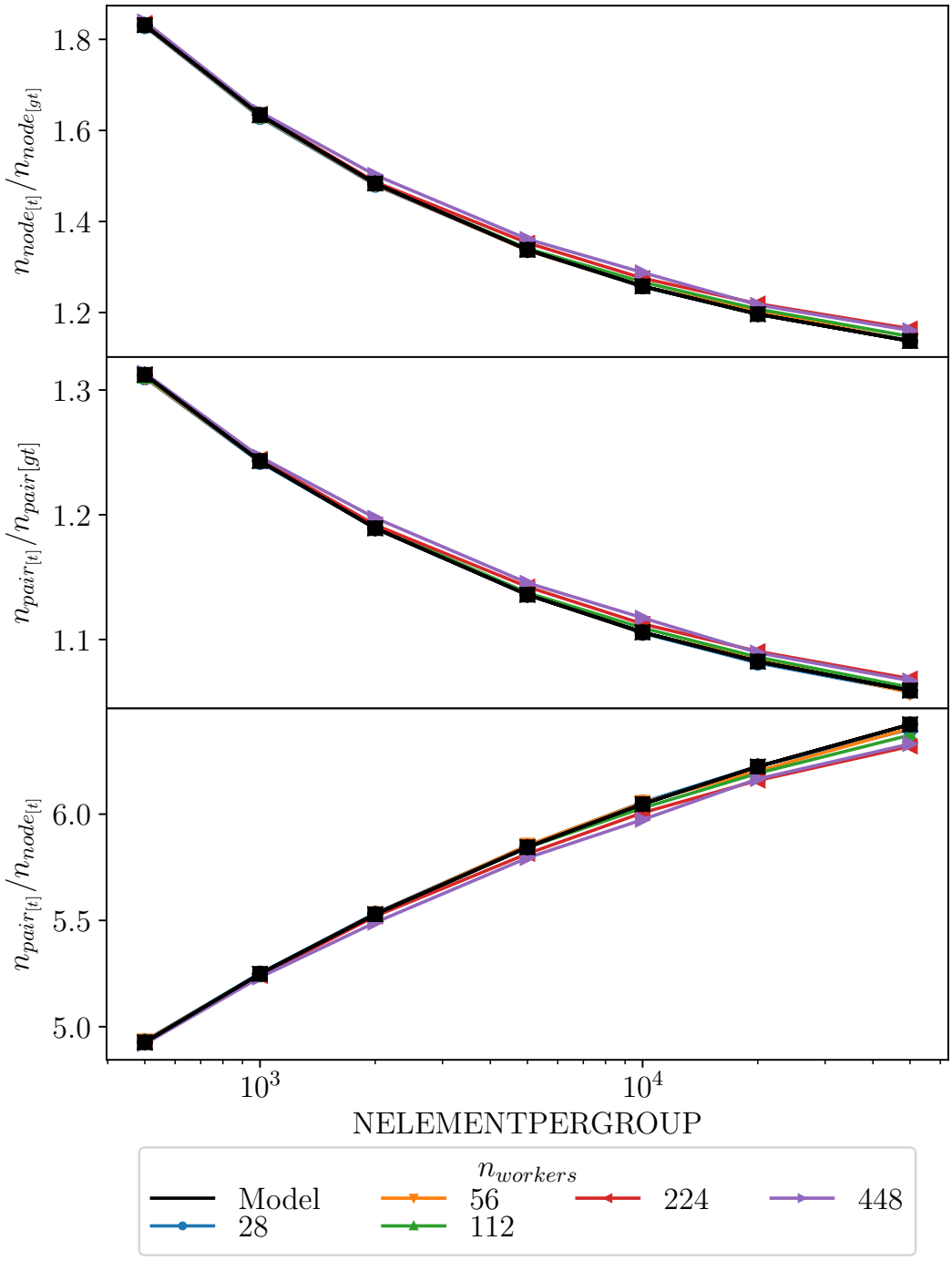


Figure 2.36: Models of the grid size variation with NELEMENTPERGROUP for Preccinsta 14M

Model	α	β	γ
$node_{[gt]}$	/	$\simeq 1/6$	/
$pair_{[gt]}$	/	$\simeq 6/5$	/
$node_{[t]}$	1	$\simeq 3/200$	$\simeq 2/5$
$pair_{[t]}$	1	$\simeq 1/125$	$\simeq 2/5$
$pair/node_{[t]}$	$\simeq 37/5$	$\simeq -2/5$	$\simeq 1/5$

Table 2.4: Coefficients obtained for the grid model

$$n_{pair_{[gt]}} \simeq \beta_{pair_{[gt]}} \times n_{elem_{[gt]}} \,. \tag{2.8}$$

The model of the ratio between the total amount of unique and duplicated nodes and pairs is more complicated. The duplicated nodes and pairs lie on the external surface of the ElGrps. It appears then intuitive to build a model that adds to the amount of unique nodes a quantity that is related to the volume to surface ratio of the groups of elements. This relation is obtained via the γ exponent in Equation 2.9 and Equation 2.10.

$$\frac{n_{node_{[t]}}}{n_{node_{[gt]}}} \simeq 1 + \beta_{node_{[t]}} \times n_{ElGrp_{[gt]}}^{\gamma_{node_{[t]}}} \,. \tag{2.9}$$

$$\frac{n_{pair_{[t]}}}{n_{pair_{[gt]}}} \simeq 1 + \beta_{pair_{[t]}} \times n_{ElGrp_{[gt]}}^{\gamma_{pair_{[t]}}} \,. \tag{2.10}$$

The surface of the ElGrps is to also representative of the size of the internal communicator. The IC size is proper to each worker, so it would be of interest to find a model for the ration between its size and the number of nodes with duplications n_{node} on the partition. Unfortunately it was not possible to find such relation with simple models. Finally, the ratio between the total number of pairs and the total number of nodes can be modelled as in Equation 2.11.

$$\frac{n_{pair_{[t]}}}{n_{node_{[t]}}} \simeq \alpha_{pair/node_{[t]}} + \beta_{pair/node_{[t]}} \times n_{ElGrp_{[gt]}}^{\left(\gamma_{pair/node_{[t]}}\right)} \,. \tag{2.11}$$

Table 2.4 summarises the values obtained for the coefficients for Equations 2.7, 2.8, 2.9, 2.10 and 2.11, while Figure 2.35 and Figure 2.36 show that such models, in black, fit quite well with the measurements.

2.2.6.2 Number of iterations in the DPCG solver

In a similar manner, it is interesting to find a model for the number of iterations on the fine grid $n_{iter_{Fine}}$ and the ratio between the number of iterations on the coarse grid and the one on the fine grid $n_{iter_{Coarse}}/n_{iter_{Fine}}$. The Poisson's equation is an elliptic equation, the convergence of the system must depend on the time needed to propagate an information throughout the domain. For a discrete 1D system this means that the number of iterations needed to solve the Poisson's equation is proportional to the number of discretisation points. This can be extended in 2D

Model	κ
$n_{iter_{Fine}}$	835
$n_{iter_{Coarse}}$	2.9

Table 2.5: Coefficients for the model of the number of iterations

and 3D, where the proportionality is respectively the square root and cubic root of the number of discretisation points, for a square or cubic domain. The model for the iteration numbers is consequently sought with Equation 2.12 and Equation 2.13, where $n_{dim} = 3$ for this 3D simulation. The coefficients for this model are reported in Table 2.5. Figure 2.37 shows the good agreement between the model and the measured data. As a confirmation for the models, on the bottom plot is shown the model of the total number of deflation iterations, which is obtained multiplying the two models for the number of iterations. This results in the multiplication of the two κ coefficients, as shown in Equation 2.14.

$$n_{iter_{Fine}} \simeq \frac{\kappa_{iter_F}}{n_{ElGrp}^{\left(\frac{1}{n_{dim}}\right)}} . \tag{2.12}$$

$$\frac{n_{iter_{Coarse}}}{n_{iter_{Fine}}} \simeq \kappa_{iter_C} \times n_{ElGrp}^{\left(\frac{1}{n_{dim}}\right)} . \tag{2.13}$$

$$n_{iter_{Coarse}} = n_{iter_{Fine}} \times \frac{n_{iter_{Coarse}}}{n_{iter_{Fine}}} \simeq \frac{\kappa_{iter_F}}{n_{ElGrp}^{\left(\frac{1}{n_{dim}}\right)}} \times \kappa_{iter_C} \times n_{ElGrp}^{\left(\frac{1}{n_{dim}}\right)} \simeq \kappa_{iter_F} \times \kappa_{iter_C} . \tag{2.14}$$

2.2.6.3 Fine grid iteration

The case of time spent in each iteration is much more complicated. Instead of trying to find a model for the entire cost of the iteration, it is better to split it in parts, and look for a sub-model for each part. The iteration on the fine grid has been divided in 5 sub-parts: the computation on the nodes, the computation on the pair of nodes, the collective communications, the point-to-point communications and the rest, the was harder to classify but, as seen above, accounts for a mere 10% of the iteration and its contribution does not seem to be particularly influenced by the size of the groups. The update of the internal communicator for the nodes and for the pairs has been included in the respective computation phases. The cost of each phase and their respective models are shown in Figure 2.38.
It is logical to think that the cost of a computation phase would be proportional to the amount of work, i.e. the number of nodes pairs on the partition. The size of the groups already has and effect on $n_{node_{[t]}}$ and $n_{pair_{[t]}}$, however it should be expected to have an additional influence also on the computation time. This is

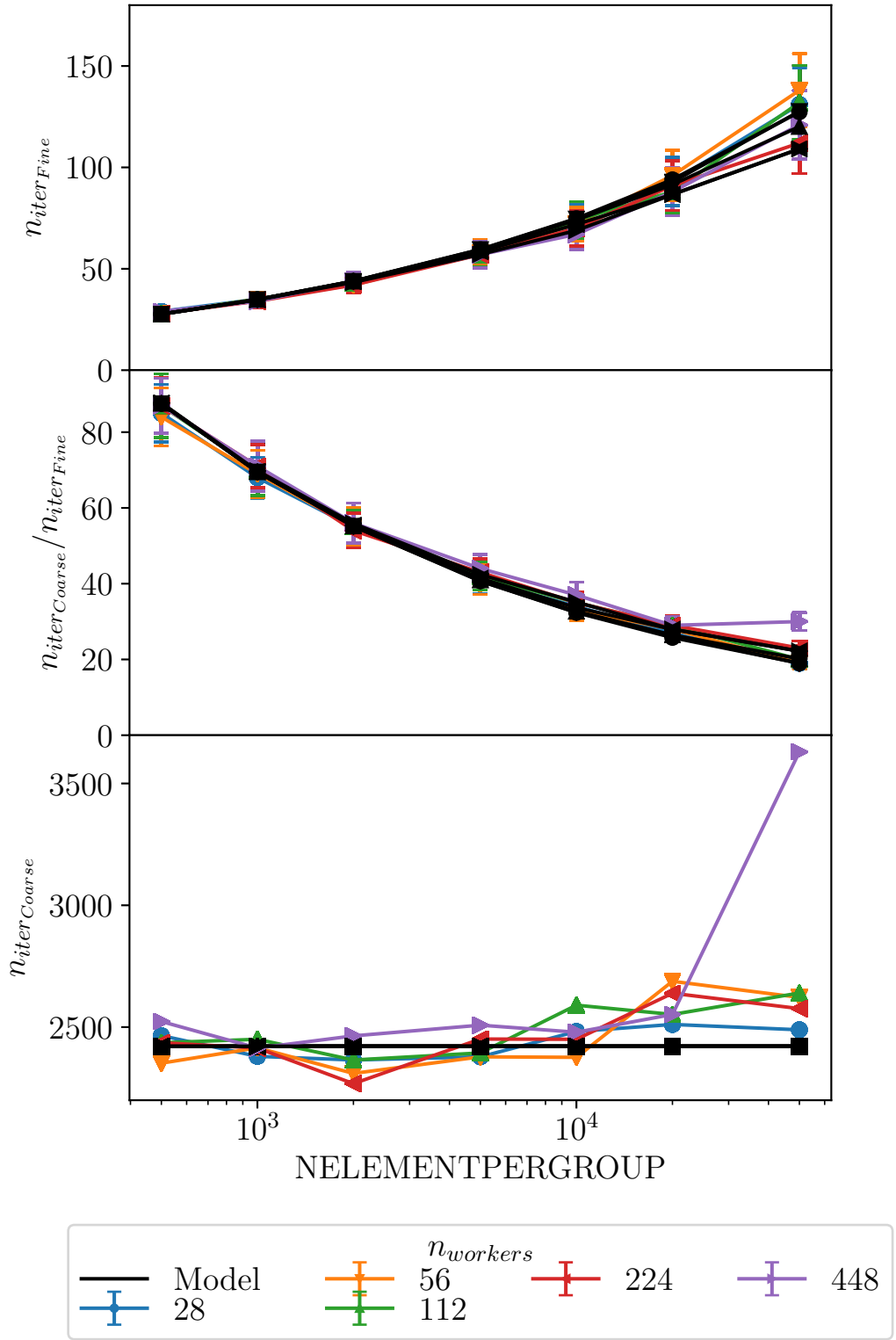


Figure 2.37: Models of the number of iterations on the fine and coarse grids for Preccinsta 14M

$n_{workers}$	$\beta_{1_{node}}$	$\beta_{2_{node}}$	$\beta_{1_{pair}}$	$\beta_{2_{pair}}$
28	0.041	8.13	0.0102	4.16
56	0.027	8.14	0.0107	3.93
112	0.022	5.55	0.0098	4.87
224	0.014	4.69	0.0057	6.65
448	0.001	2.36	0.0052	2.25

Table 2.6: Coefficients to model the cost of the computation on nodes and pairs

due to several effects: small groups should benefit from improved memory access time due to the cache blocking effect, however they are penalised by the node and pair duplication and the size of the internal communicator. The model used for the computation phases is then a simple combination of the two contributions as in Equation 2.15 and Equation 2.16. In this case it was not possible to find a unique pair of coefficients that would generate a model good enough for all values of $n_{workers}$, hence a different couple of coefficients have been taken for each case. These values are shown in Table 2.6 for both nodes and pairs. Figure 2.38 shows that this model is in good agreement with the measurements in all cases.

$$T_{node}\,[\mu s] = \beta_{1_{node}} \times n_{node} + \beta_{2_{node}} \times n_{ElGrp}\,. \tag{2.15}$$

$$T_{pair}\,[\mu s] = \beta_{1_{pair}} \times n_{pair} + \beta_{2_{pair}} \times n_{ElGrp}\,. \tag{2.16}$$

The performance prediction for the communication phases is a more complex problem. The postal model $T_{msg} = \lambda + \beta \times n_{byte}$ assumes that the time needed to send a message T_{msg} is the sum of a fixed latency λ and the product of the inverse of the bandwidth β by the size of the message n_{byte}. A simple performance model for the MPI communication is proposed by [87, 88], while [89] presents an extensive study of performance models for different versions of the PCG algorithm. In particular this paper modifies the postal model to take into account some penalties due to network congestion and other factors. Such penalisation is based on the specific hardware counters and seem too fine grained for a simpler and summary model such as the one that is looked for here. It would be however extremely interesting to try and apply such models at the YALES2 DPCG implementation to generalise the values of the coefficients for the models presented here. Figure 2.38 shows that there is no clear behaviour or dependency for the collective communications. The time spent in such phase seems to oscillate erratically around a constant value, without a clear dependence on the number of workers. Counter-intuitively as it may seem, it is on average lower for 448 than 28. This shows that a model based on a $log(n_{workers})$ type of function, which works quite well for a stand-alone `MPI_ALLREDUCE` for example, it is of no use here. This confirms that the time spent in the collective communications depends mostly on factors external to the communication itself. Without a better indication, a simple linear model as in Equation 2.17 is adopted for the collective communications. The value for the coefficients are reported in Table 2.7. Figure 2.38

shows that the model is not always in good agreement with the measured data and should probably be further improved. The worse case is $n_{workers} = 224$, whose cost is highly overestimated.

$$T_{Coll}\,[\mu s] = \alpha_{Coll} + \beta_{Coll} \times n_{workers}\,. \tag{2.17}$$

In order to model point-to-point communications it can be useful to look at how the data exchange is actually implemented in YALES2. To some extent this was already presented in Algorithm 4, however Algorithm 8 shows in better detail the sequence of operations that are necessary for the parallel update. Although it uses non-blocking MPI functions, this procedure is quite synchronous, as the `MPI_WAITALL` on the receiving messages does not allow the calling process to proceed until all the incoming messages have been received. Compared to a mechanism with blocking calls however, it allows a process to post all its outbound messages, even if a receiving process is late. In order to model the MPI exchange, a postal model as the one mentioned above could be a reasonable solution. The algorithm includes also a copy of the internal communicator back to the data on the ElGrps. The time spent on this operation is then proportional to the size of the internal communicator. This step has not been included in the computation phase like the update of the internal communicator itself, as this is inherent to the parallel communication. The final model used for the performance of the point-to-point communication in Equation 2.18 sums the contribution of the MPI messages and the copy of the IC to the data. Since each message has a slightly different size, an average of all sizes is used. Figure 2.38 shows that this model, on average, agrees quite well with the measures, except for the case on 224 workers, which is substantially overestimated. The values of the coefficients, summarised in Table 2.7 indicate however that, if the model is correct, the influence of the MPI exchange is negligible if compared to the copy of the internal communicator. This means that the P2P communication does not depend directly on the amount of data exchanged but rather on the size of the internal communicator, that in return depends on the groups size. It is worth noting that the IC includes the amount of data exchanged, as the external communicators are part of the internal communicator itself. It is then difficult to properly separate the two contributions.

$$T_{P2P}\,[\mu s] = \alpha_{P2P} + \beta_{P2P} \times n_{IC_{node}} + n_{neighbours} \times (\gamma 1_{P2P} + \gamma 2_{P2P} \times \overline{n_{item}})\,. \tag{2.18}$$

For the remainder of the algorithm, since there is no clear pattern to be mimicked, Equation 2.19 is used to obtain a model based on the number of ElGrps and workers. The coefficients for such model are reported in Table 2.7. Figure 2.38 shows that, however simplistic and without a solid motivation, this model fits the measurements.

$$T_{Other}\,[\mu s] = \alpha_{Other} \times n_{ElGrps}^{\gamma 1_{Other}} + \beta_{Other} \times n_{workers}^{\gamma 2_{Other}}\,. \tag{2.19}$$

The complete model for the iteration, obtained adding together the contributions of all the sub-models discussed above, is superimposed to the original measurements in Figure 2.39. The obtained model fits almost perfectly the measurements. The

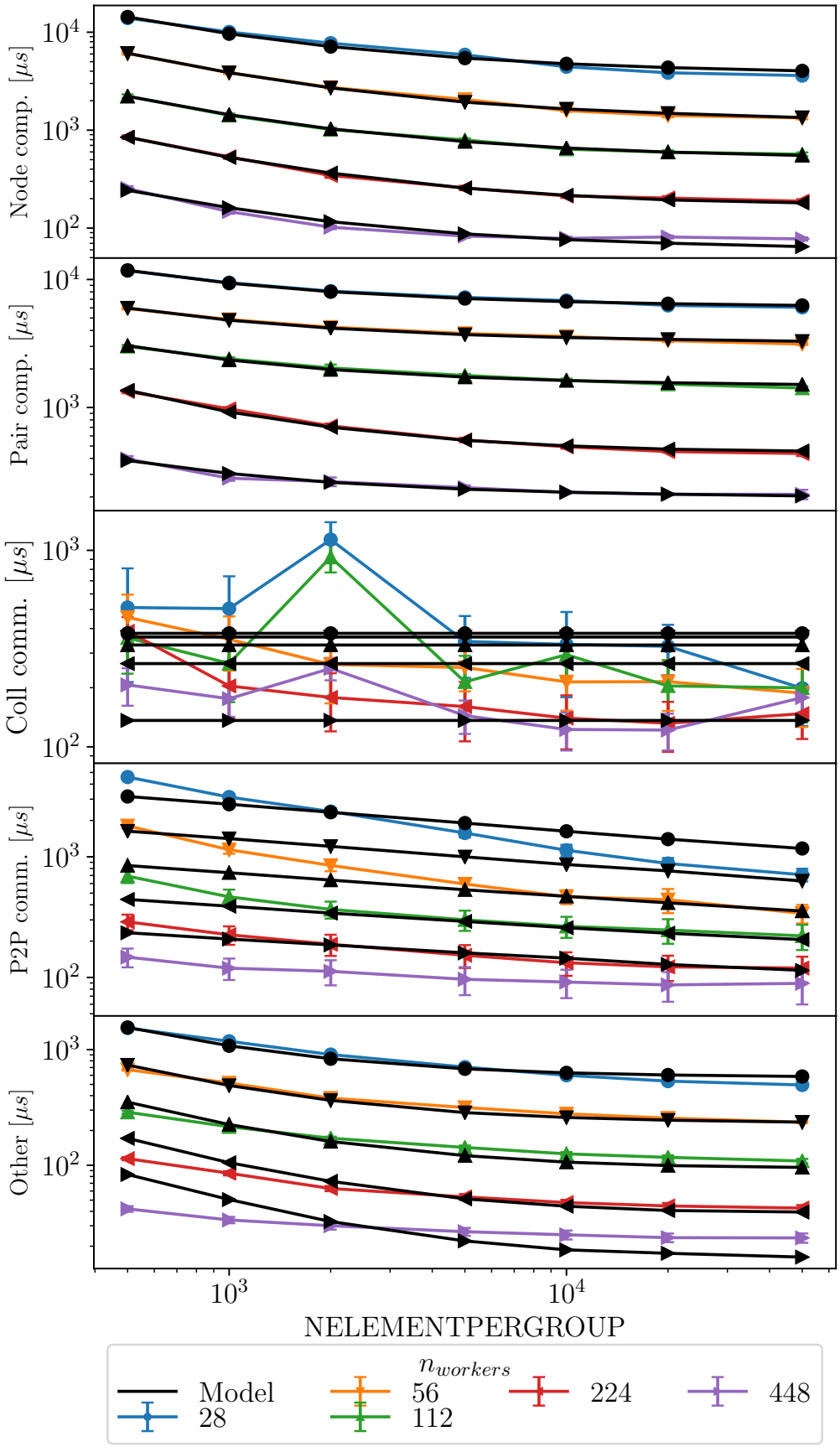


Figure 2.38: Models for each phase of a fine grid iteration for Preccista 14M on MYRIA

Algorithm 8: Parallel data update in YALES2

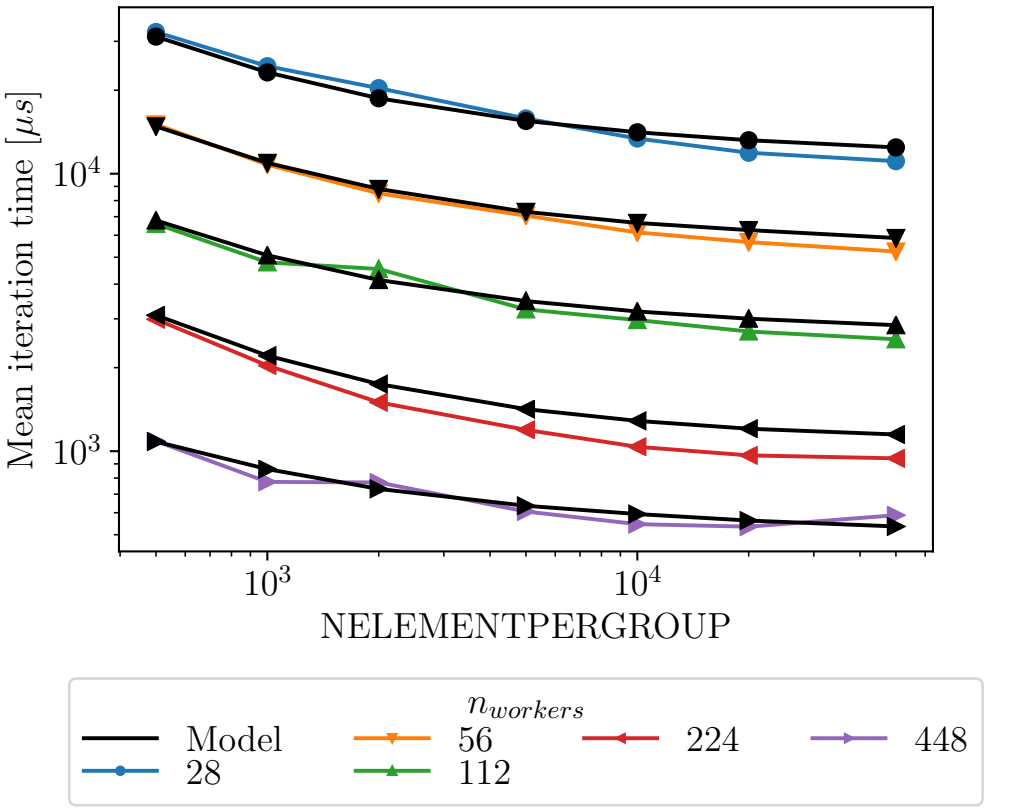
```
for i ← 1 to n_neighbours do
    MPI_IRECV(recv_buffer[i],n_item [i],neighbour [i],...,recv_request[i]);
end
for i ← 1 to n_neighbours do
    Pack: IC → (n_item [i]) → send_buffer[i];
    MPI_ISEND(send_buffer[i],n_item [i],neighbour [i],...,send_request[i]);
end
MPI_WAITALL(n_neighbours,recv_request,...);
for i ← 1 to n_neighbours do
    Unpack: recv_buffer [i] → (n_item [i]) → IC;
end
MPI_WAITALL(n_neighbours,send_request,...);
Copy IC to ElGrps
```

Model	α	β	γ_1	γ_2
Coll	394.2	-0.58	/	/
P2P	2.7E-16	0.061	2.7E-16	7.8E-16
Other	4.67E04	1.337	-1.32	0.958

Table 2.7: Coefficients to model the cost of the computation on nodes and pairs

worse case is $n_{workers} = 224$. This is due to the bad modelling of the collective communication, whose important overestimation is clearly visible in the complete model. The model tends to overestimate the cost of those cases with a larger value of NELEMENTPERGROUP.

Although all coefficients used for the various models have been determined a posteriori with a least square method regression on the different measurements, it is important to underline the fact that all the variables used in the fine grain iteration model, with the exception of the size of the internal communicator $n_{IC_{node}}$, can be obtained, thanks to the grid size models, from only 3 parameters. These are the number of elements of the grid $n_{elem_{[gt]}}$, the number of processes $n_{workers}$ and the input parameter NELEMENTPERGROUP, which are known a priori for each simulation. The number of neighbours for each process is another parameter that is not available a priori and that has not been modelled, however it has been shown to have a negligible influence on its sub-model, consequently it could be removed. This means that, if a similar model is obtained for the deflation, the runtime of the DPCG solver could be determined beforehand, with a certain accuracy.



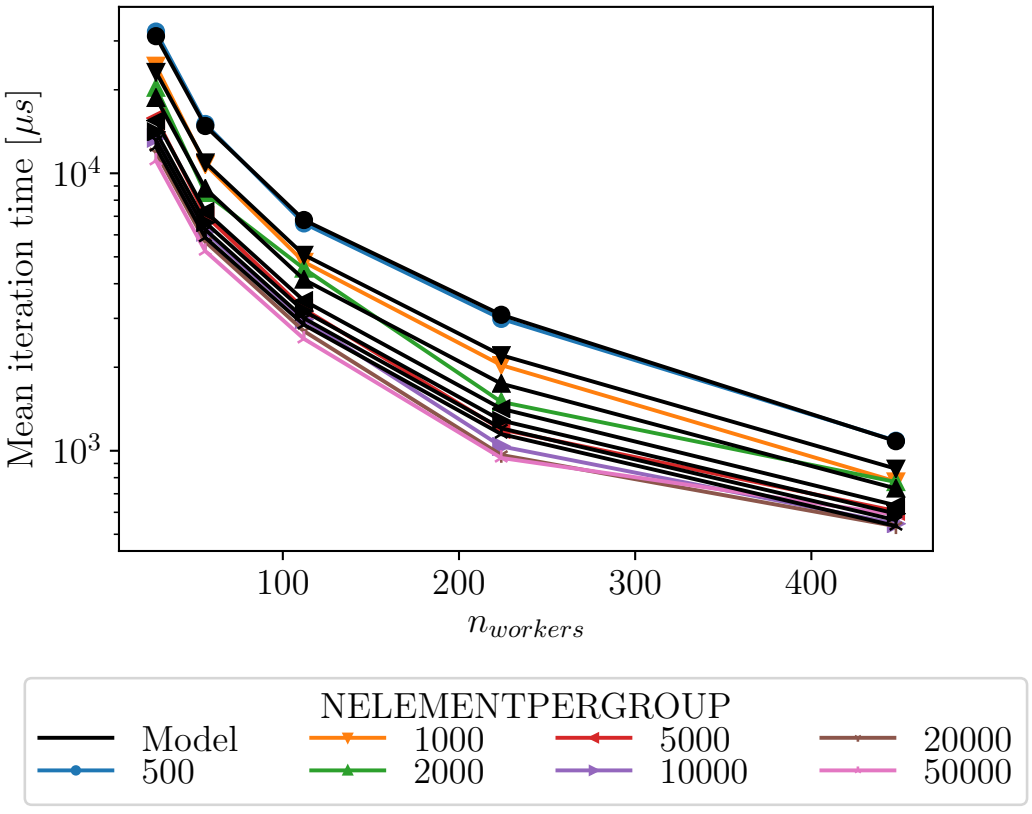


Figure 2.39: Resulting model for the iteration on the fine grid for Preccinsta 14M on MYRIA

Model	α	β
ElGrp	0.446	0.00446
ElGrpPair	0.118	0.00243

Table 2.8: Coefficients obtained for the computation phases on the deflation grid

2.2.6.4 Deflation iteration

The model for the deflation iteration is obtained in a similar manner. The different phases shown in the breakdown plots of Figure 2.33 are considered separately and the full model is obtained adding the different sub-models of each phase. The first phase is the computation on the group of control volumes. This is modelled with Equation 2.20 as a simple linear function of the number of ElGrps on each worker. The same principle is applied to the computation on the pairs of groups, as in Equation 2.21. While the number of groups does not vary much among the different workers, the same is not true for the number of ElGrp pairs. This was shown to cause some imbalance on its computation phase. The model is based on an average value of pairs computed across all workers. The values obtained for the coefficients are displayed in Table 2.8.

$$T_{ElGrp}\,[\mu s] = \alpha_{ElGrp} + \beta_{ElGrp} \times n_{ElGrp}\,. \tag{2.20}$$

$$T_{ElGrpPair}\,[\mu s] = \alpha_{ElGrpPair} + \beta_{ElGrpPair} \times n_{ElGrpPair}\,. \tag{2.21}$$

The trace in Figure 2.34 showed that the time accounted as collective communication by most processes is actually time spent waiting for other workers that were late in the previous phases. The model tries to account for this behaviour having a contribution that is proportional to the difference between the average and the maximum amount of neighbours of all processes. The number of neighbours on the deflation grid is not guaranteed to be the same that on the fine grid. In order to distinguish the two, for the coarse grid the term n_{GC} is used, which stands for the number of ghost communicators. In Subsection 2.1.5 it was explained how the deflation uses a half-halo system with ghost cells only on one of the workers exchanging data. The ghost communicators however exist on the two sides, and are used as the external communicators on the fine grid to pack and unpack the data on buffers for the MPI operations. Consequently for each worker the number of ghost communicators is equivalent to the number of neighbours on the coarse grid. A contribution proportional to the number of workers is also added, resulting in Equation 2.22. The coefficients for this model are displayed in Table 2.9. The number of neighbours on the deflation grid is not expected to change with the number of groups, as it depends only on the first grid partitioning among the different workers. What changes is the amount of data exchanged between the different workers, as less groups means less boundary data. Consequently, the model of the collective communication remains constant varying the value of NELEMENTPERGROUP. This happens to overestimate the average cost of the collective communication for $n_{worekrs} = 28$, however it

Model	β_1	β_2	β_3
D.Coll	0.025	1.109	/
D.Ghost	1.153	0.028	/
D.Other	0.092	0.004	5.02E-4

Table 2.9: Coefficients obtained for the last three phases of the deflation iteration

is still inside the error range.

$$T_{D.Coll}\,[\mu s] = \beta_{1_{D.Coll}} \times n_{workers} + \beta_{2_{D.Coll}} \times (\|n_{GC}\|_\infty - \overline{n_{GC}})\ . \tag{2.22}$$

The mechanism used to exchange data among process in point-to-point communications is similar to that used for the fine grid. In this case however the buffers are not packed from and unpacked to the internal communicator but directly on the data. The model of Equation 2.23 then only takes into account the number of ghost communicator and their size. The coefficient for such model can be found in Table 2.9.

$$T_{D.Ghost}\,[\mu s] = \overline{n_{GC}} \times (\beta_{1_{D.Ghost}} + \beta_{2_{D.Ghost}} \times \overline{n_{item_{GC}}})\ . \tag{2.23}$$

The remainder of the algorithm is the exact evaluation of the residuals which is performed every 10 iterations. Its cost represents that iteration, which is then averaged also over the 9 iterations in which it was not performed. This phase consists on a computation on the pair of groups, a ghost exchange and a computation on an ElGrps data. The ElGrp computation phase includes multiple dot-products and other operations on the groups, and it contribution is one order of magnitude lower than the communication phase. The operation on the ElGrp data is then excluded and this phase is modelled as the sum of a ghost exchange and an operation on the pairs of groups. Since its done only once every 10 iterations its coefficients will differ from those of the other two parts separately, and they are reported in Table 2.9. From the value of the coefficients it is clear that the contribution of the ElGrp pairs computation is paltry with respect to the ghost exchange. Since there is no other significant contribution other than the ghost exchange, $\beta_{1_{D.Other}} \simeq {}^{\beta_1}{}_{D.Ghost}/10$ and $\beta_{2_{D.Other}} \approx {}^{\beta_2}{}_{D.Ghost}/10$, as it could be expected.

$$\begin{aligned} T_{D.Other}\,[\mu s] =& \overline{n_{GC}} \times (\beta_{1_{D.Other}} + \beta_{2_{D.Other}} \times \overline{n_{item_{GC}}}) + \\ &+ \beta_{3_{D.Other}} \times n_{ElGrpPair}\,. \end{aligned} \tag{2.24}$$

The resulting model is shown in Figure 2.41. With the exception of $n_{workers} = 28$ whose runtime is quite largely overestimated, the model tends to slightly underestimate the cost of the deflation iteration for all group sizes and number of workers. It seems that the model obtained for the deflation iteration is not as good as the one for the fine grain iteration. It should be noted however that the time scales are much different ($\sim 1E1\,[\mu s]$ vs $\sim 1E4\,[\mu s]$) and although the relative error is larger, in absolute terms, the errors of the deflation model are much smaller.

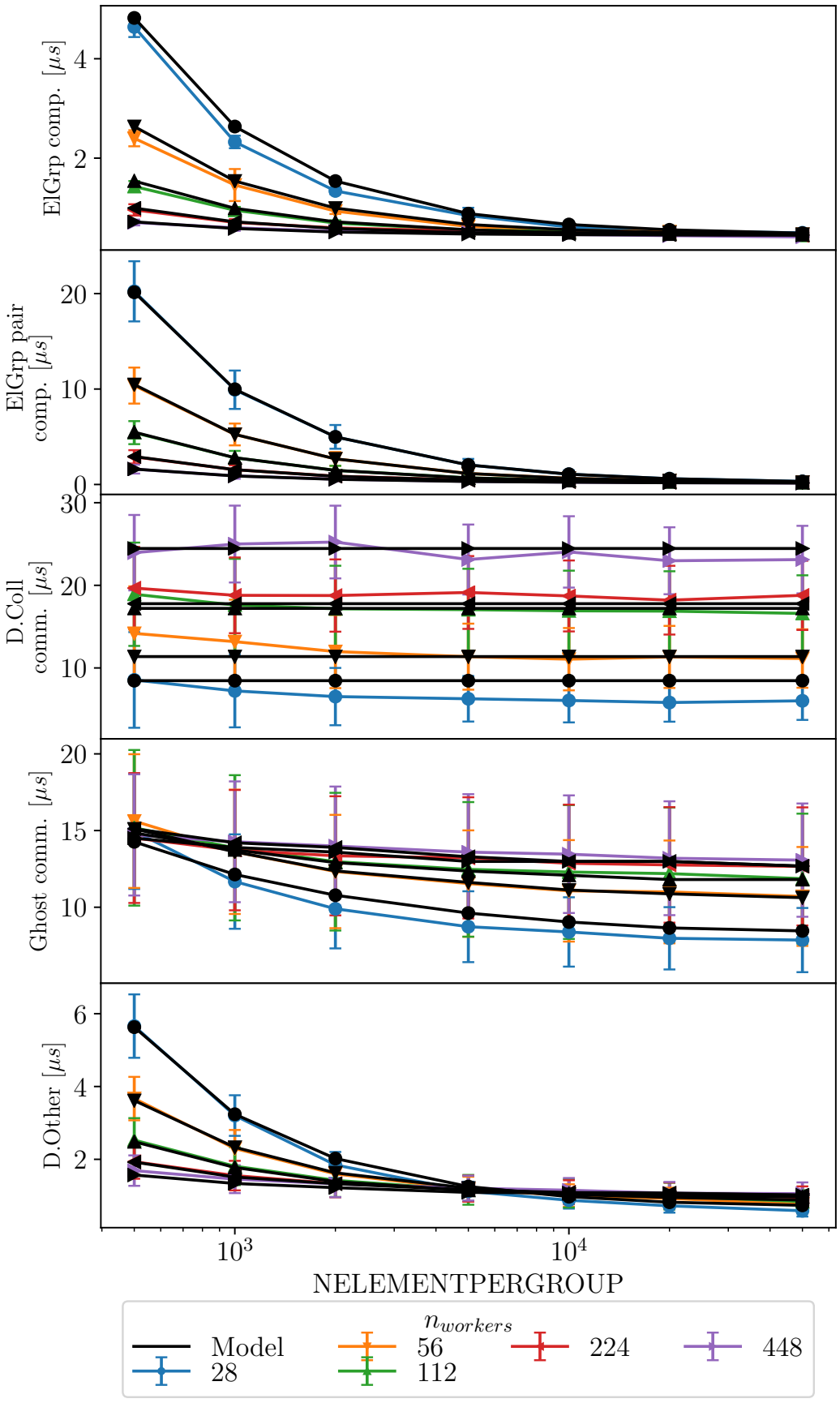


Figure 2.40: Models for each phase of a deflation iteration for Preccinsta 14M on MYRIA

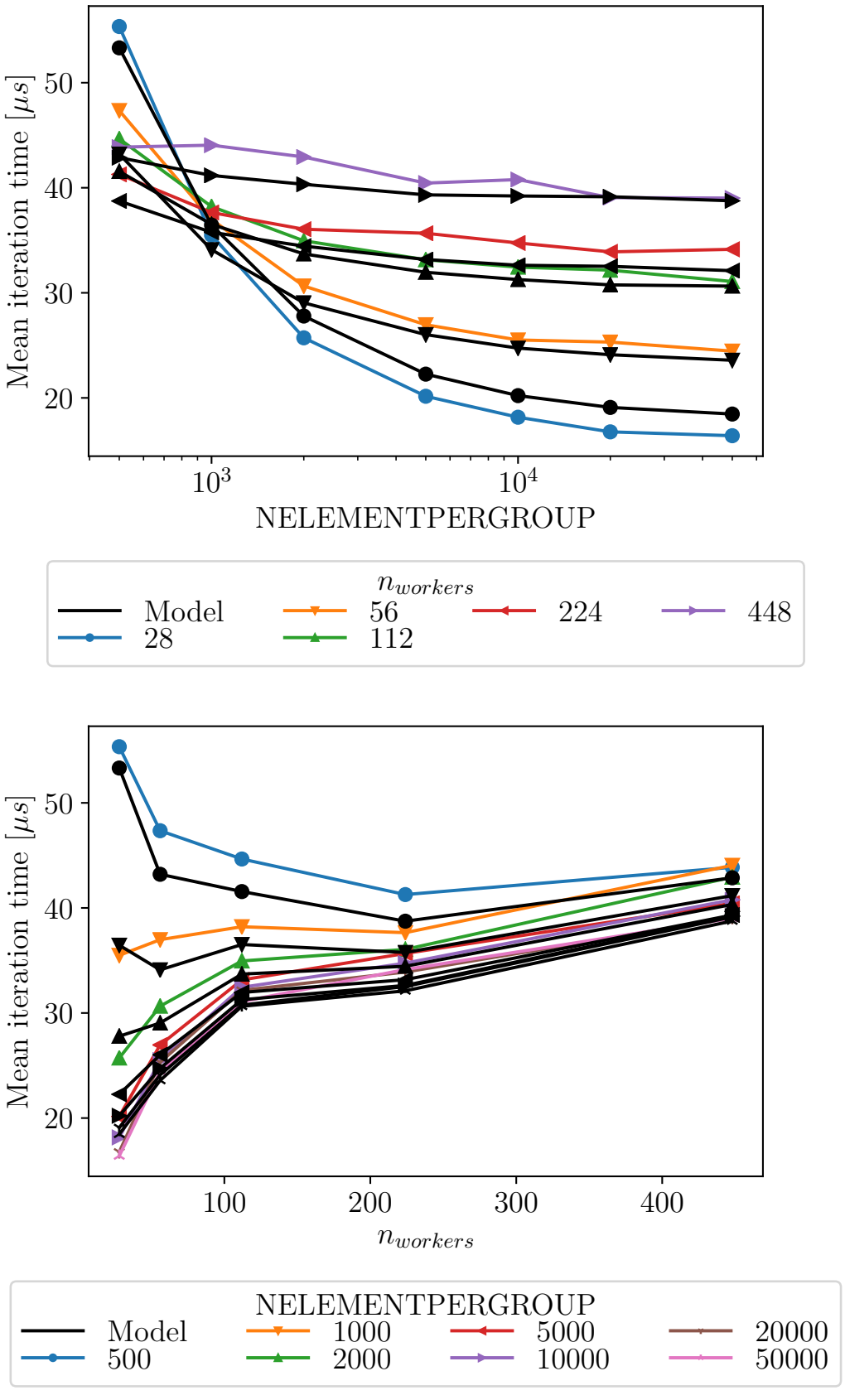


Figure 2.41: Resulting model for the deflation iteration for Preccinsta 14M on MYRIA

2.2.6.5 Scalability of the model

The same models can be applied to the Preccinsta mesh of 110 Million elements with fair accuracy. The relations modelling the grid characteristics remains unchanged, as groups are created the same way. Clearly, the value of $n_{ElGrp_{[gt]}}$, $n_{node_{[gt]}}$ and $n_{pair_{[gt]}}$ are different, as they are linked to the $n_{elem_{[gt]}}$, but the model still predicts them correctly, as represented in Figure 2.42 and Figure 2.43. However, Figure 2.44 shows that, using the same coefficients obtained for the 14M grid, the number of iterations on both fine and coarse grid are not predicted correctly. This indicates that such coefficients have a dependency on the grid size. The trends however are still well predicted, and a re-tuning of the coefficients allows to predict the correct values. It is important to remark how, as more elements mean a larger system to solve, the number of iterations on the fine grid increased. However, for each of the fine grid iterations, the deflation system converged faster, in spite of the larger number of groups. Figure 2.44 can indeed be misleading, as the number of iterations are represented as a function of NELEMENTPERGROUP, when they are in fact a function of $n_{ElGrp_{[gt]}}$. This representation could be deceptive when this Figure is compared with Figure 2.37, due to the fact that, for a same value of NELEMENTPERGROUP, $n_{ElGrp_{[gt]}}$ are different in the two cases. Both models on the two top graphs take such difference into account, as they are indeed based on $n_{ElGrp_{[gt]}}$. The bottom graph however loses such dependency as Equation 2.14 is simply a multiplication of two constant coefficients. This helps to show how the total number of deflation iterations is finally higher for the 110M elements mesh. Figure 2.44 shows as well how with small groups the system can become unstable and its convergence erratic. For the case with $n_{workers} = 896$ and NELEMENTPERGROUP=500 the standard deviation in the number of fine grid iterations computed across several time steps is extremely large, which is symptom of such instability. For this reason, such low values of NEPG are to be avoided. The model can also be applied to the time spent in the fine grid iteration and its different phases, as shown in Figure 2.45 and Figure 2.46 respectively. While for both computation phases the coefficients remains practically unchanged, collective communications are twice as expensive than on the 14M element mesh, and that is reflected on the α_{Coll} coefficient, which doubles as well. β_{Coll} is almost null, which indicates that the dependency on the number of workers is fortuitous and should probably be removed or exchanged for a factor that explains the difference between the two meshes. The large standard deviation that characterise this phase suggests that such values could be influenced by some imbalance, which however does not appear to be as strong in any of the other phases. Furthermore, in spite of being more expensive than the 14M counterpart, its contribution is still extremely low when compared to the rest of the algorithm. Concerning the P2P exchange, the value of β_{P2P} is switched with that of $\gamma 2_{P2P}$, while the other two coefficients are still extremely small. This would make the exchange completely dependent on the amount of data exchanged and totally unrelated to the size of the internal communicator, in contrast of what seen for the 14M case. As mentioned above, the two are strictly correlated and it's hard to separate the

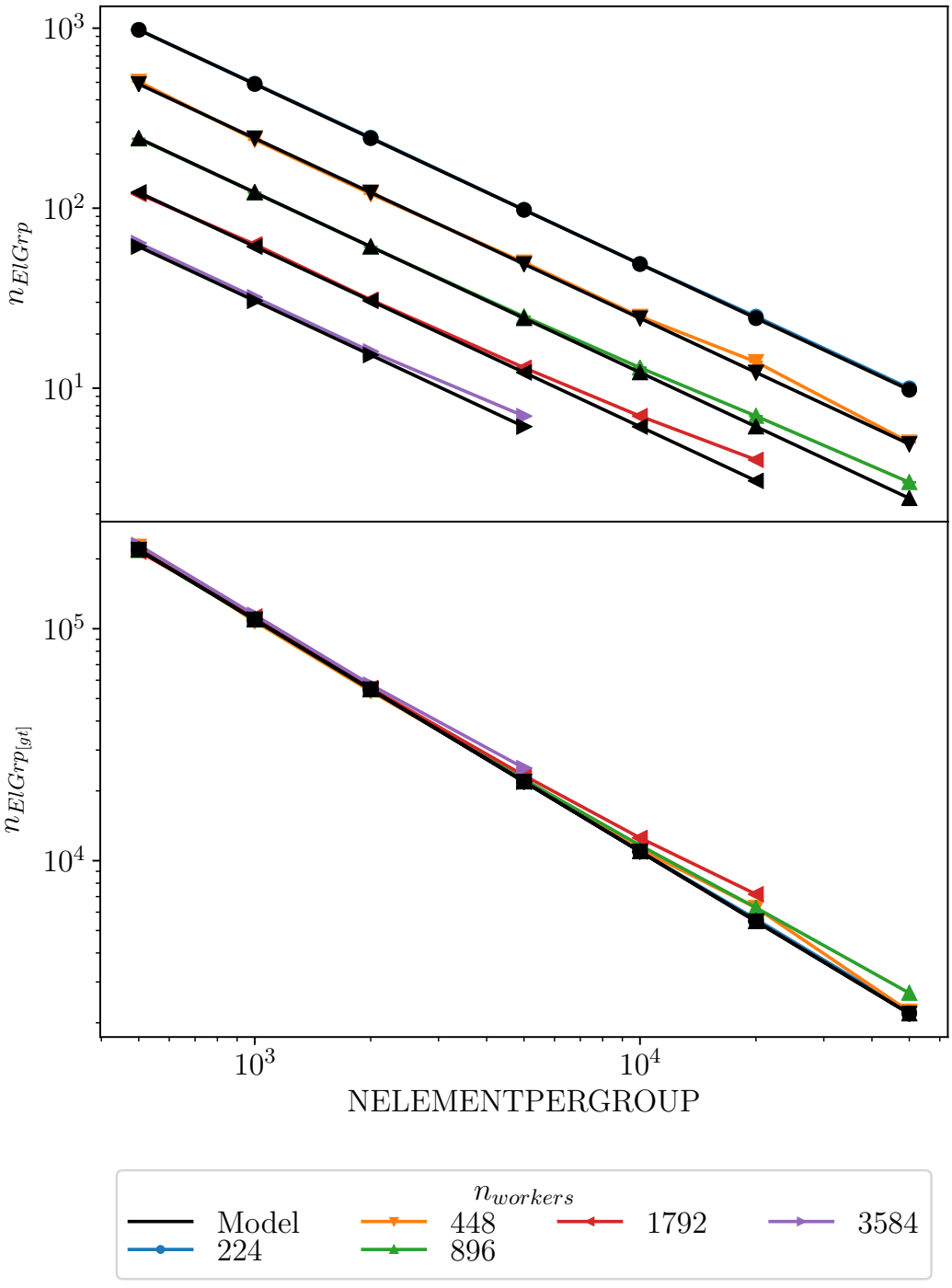


Figure 2.42: Model for the number of ElGrps applied to the Preccinsta 110M elements mesh

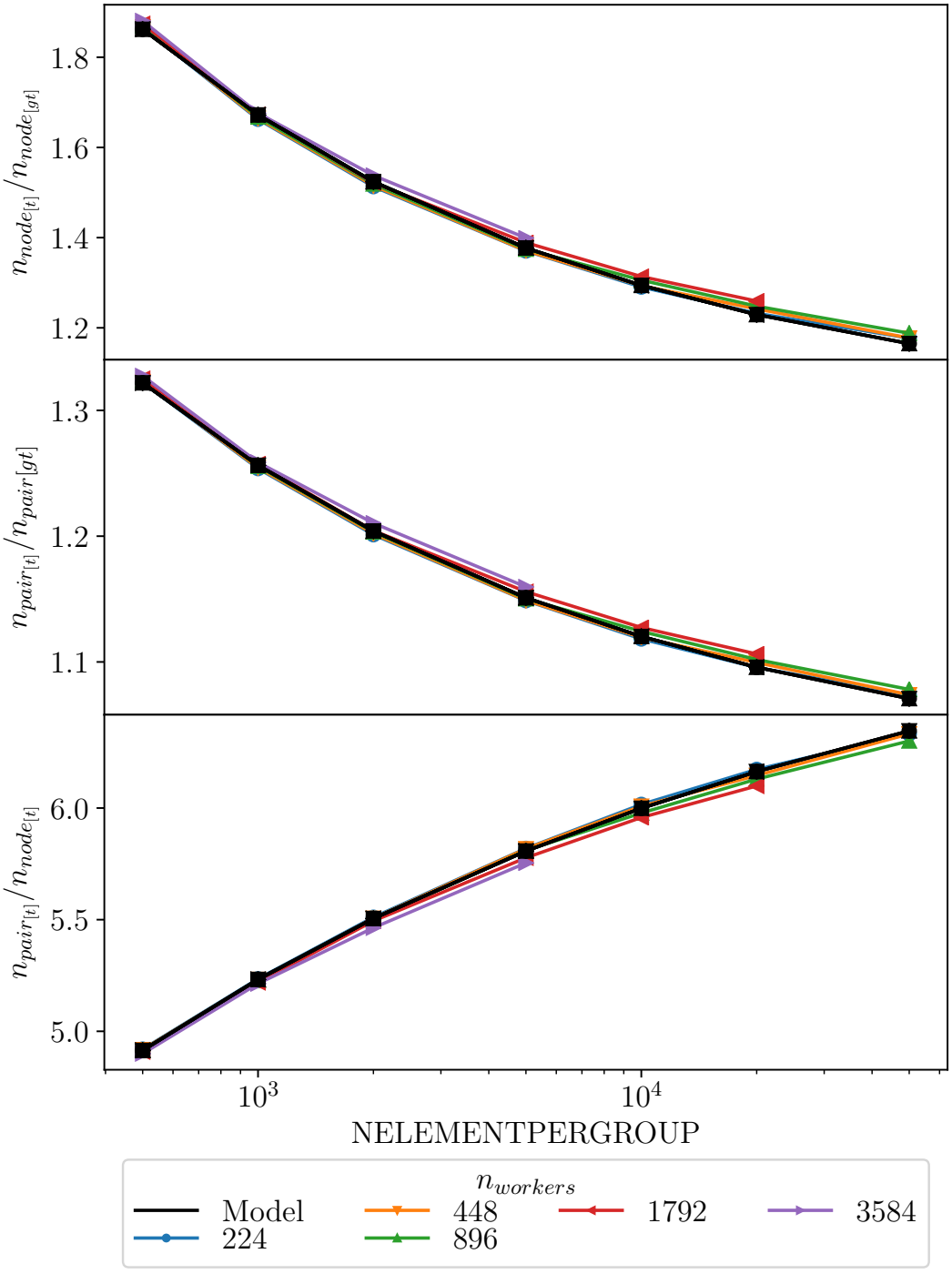


Figure 2.43: Model for the nodes and pairs duplication applied to the Preccinsta 110M elements mesh

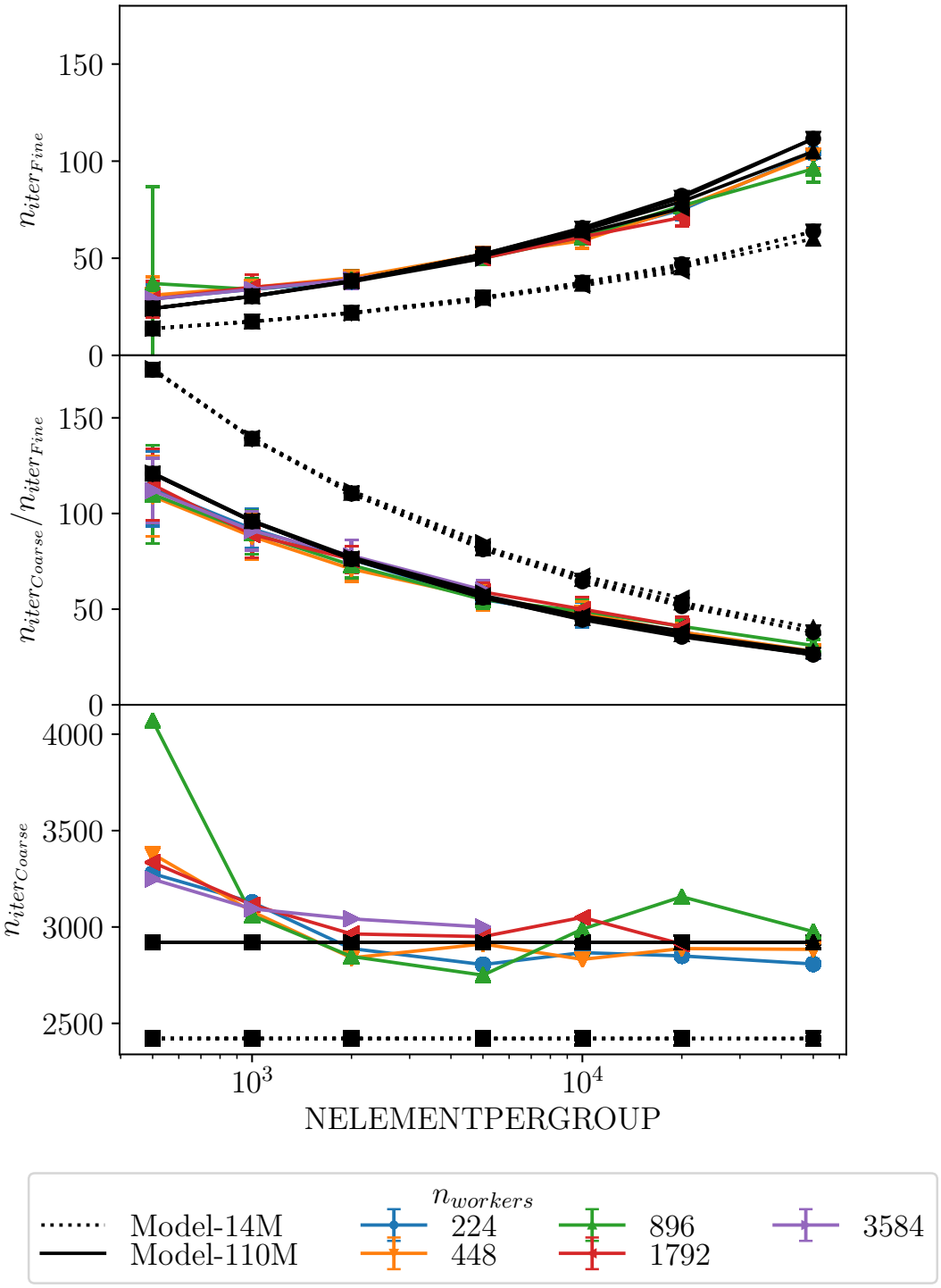


Figure 2.44: Model for the number of iterations applied to Preccinsta 110M

respective contributions. As the 110M benchmark is a weak scalability study of the 14M one, the size of the internal communicator and the amount of data exchanged by each worker should be almost the same, and this is reflected on the cost of the P2P exchange, which is extremely similar between the two cases. In spite of the different coefficients, the two models also produce similar results, which strengthen the conviction of the joint contribution of internal and external communicators on the cost of the P2P exchange. In addition, for a same value of $n_{workers}$, the size of the external communicator does not change with NELEMENTPERGROUP, hence the IC size is necessary to explain the reduction in cost for higher NEPG values. Finally, it was not possible to obtain a good agreement between model and measurements on the last phase. Although the contribution of this last phase is about the same order of magnitude of the P2P communication, which itself is much smaller than the computation, Figure 2.45 shows that the model fits quite well nonetheless. This is possibly due to the misestimation of the communication phases with respect to the measurements. In practical terms, the last phase is composed by smaller computation and P2P communication phases, which means that all possible gains obtained for the other phases will influence this one as well. Concerning the deflation iteration, it is possible to see from Figure 2.48 that the models for the computation fit quite well. The coefficients remain the same with respect to the 14M model for these two phases. Again, as it is a weak scalability study, the number of ElGrp per worker is supposed to be the same, and the number of ElGrp pairs should not differ much either. For the collective communication the coefficients had to be changed to be able to obtain such fit, mainly to balance the higher values of $n_{workers}$ which are not reflected linearly in the increment of time. For the P2P ghost exchange, a good fit is obtained with coefficients with values very close to those of the 14M case, confirming the correctness of this relation. The same is true for the last phase of the deflation iteration. Figure 2.47 shows that the model is in fair agreement with the measured performances, once again with the exception of the lower value of $n_{workers}$ which is largely overestimated.

2.2.6.6 Further considerations on the performance model

Figure 2.27 shows the average runtime of a complete DPCG iteration as a function of NELEMENTPERGROUP and $n_{workers}$. It can be seen how, as stated above, the optimal value for NELEMENTPERGROUP is around 2'000. The proposed models underline the fact that the number of iterations on the fine grid is inversely proportional to a power of the number of ElGrps, and consequently directly proportional to their size. On the other hand, the cost of a single iteration on the fine grid is inversely proportional to NELEMENTPERGROUP, but not enough to compensate the increase in their number. Concerning the deflation iterations, their total number remains practically unchanged, and the cost of each iteration is reduced at most to a third of its maximum value for very large values of NELEMENTPERGROUP. A relatively small value of NELEMENTPERGROUP such as 2'000 is consequently the best compromise as it allows a substantial reduction, although not maximal,

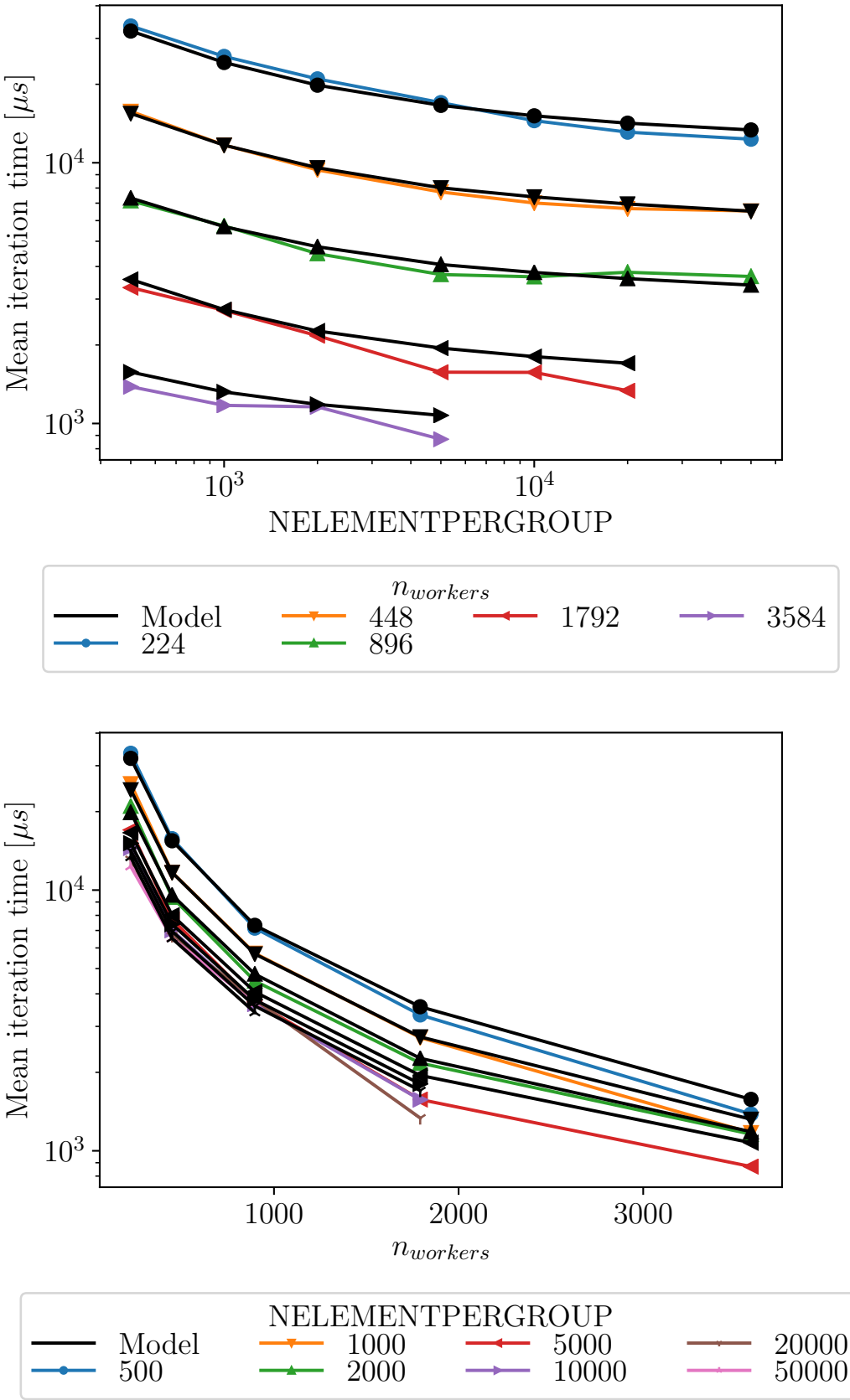


Figure 2.45: Model for the fine grid iteration cost of Preccinsta 110M on MYRIA

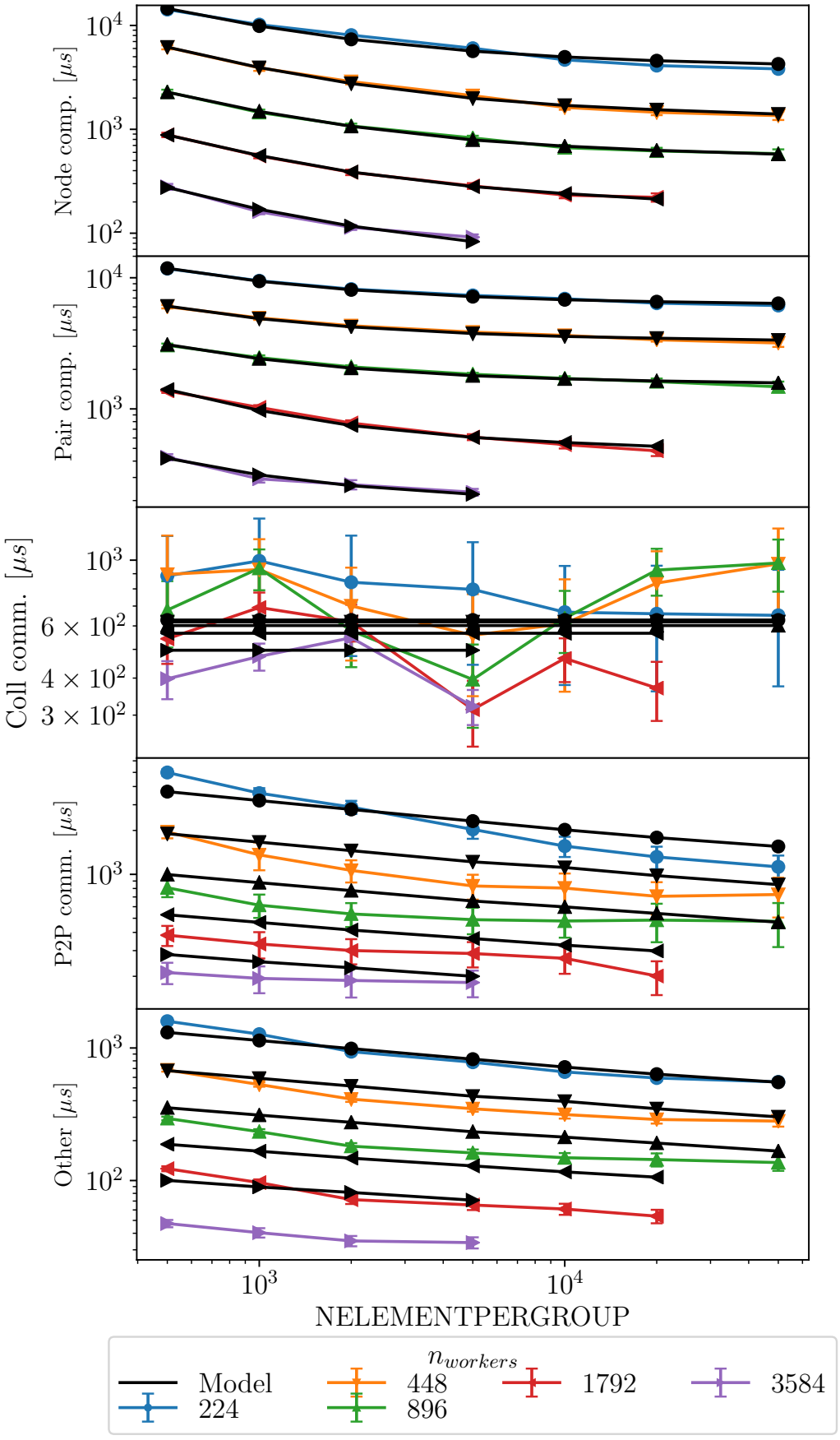


Figure 2.46: Model for cost of the different phases in the fine grid iteration for Preccinsta 110M on MYRIA

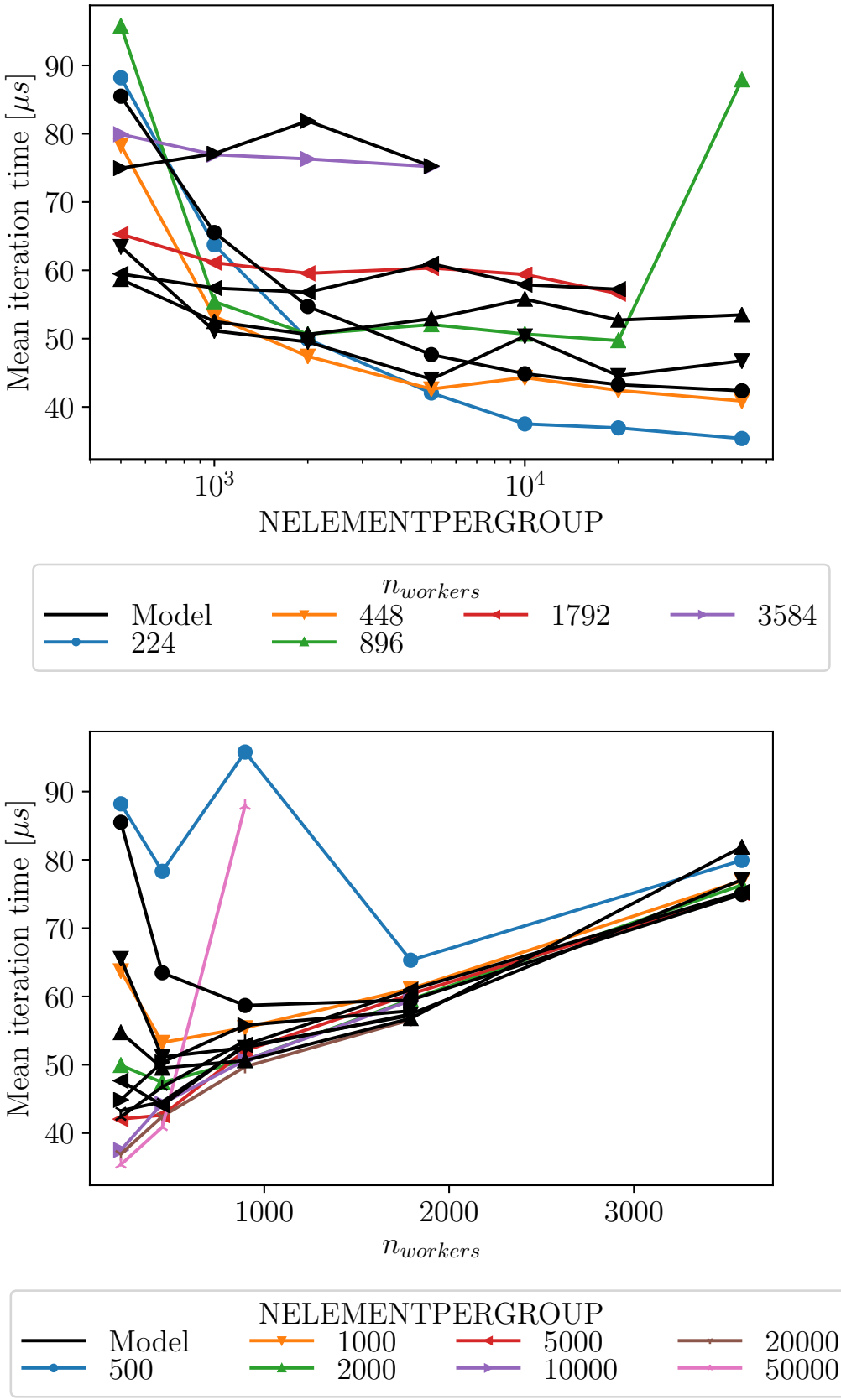


Figure 2.47: Model for the coarse grid iteration cost of Preccinsta 110M on MYRIA

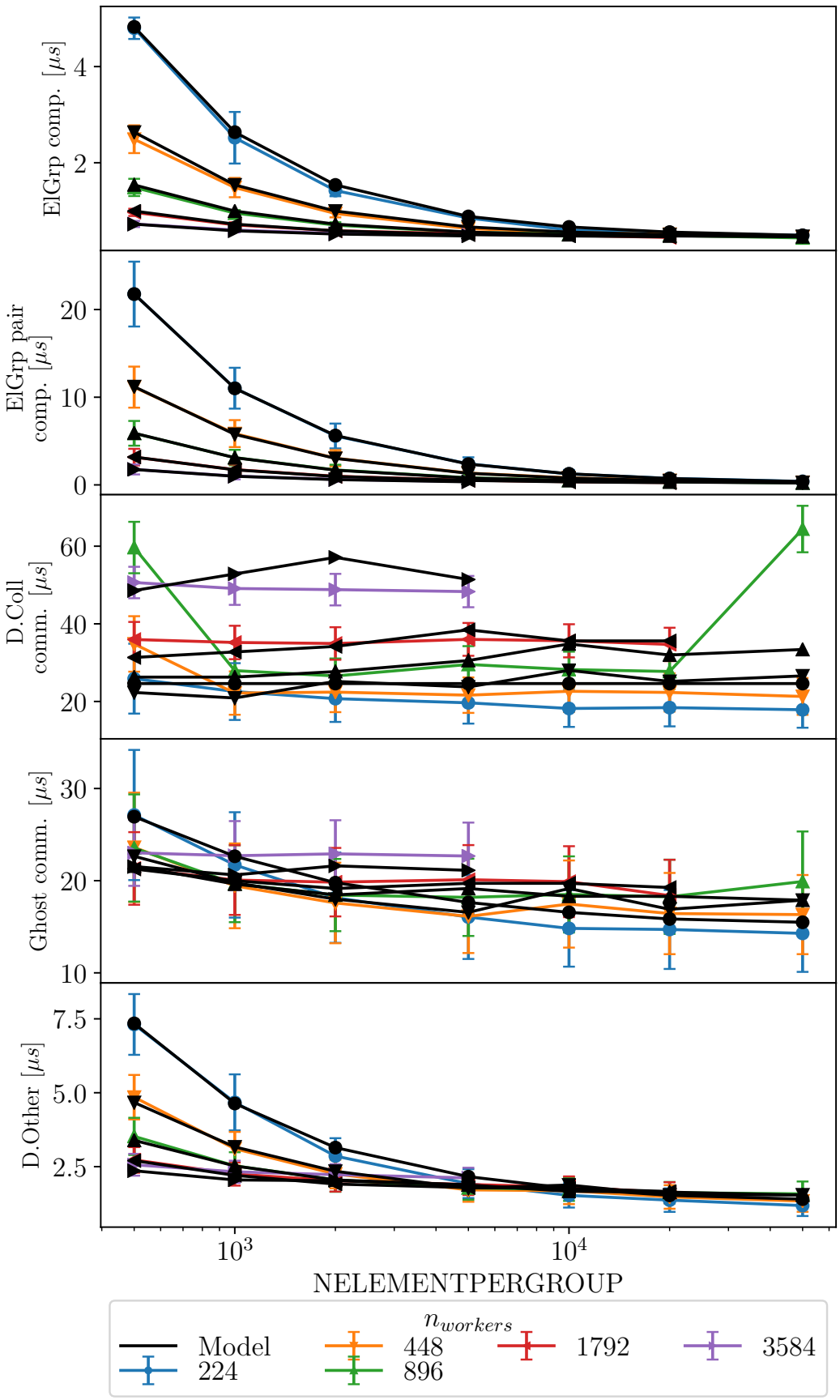


Figure 2.48: Model for cost of the different phases in the coarse grid iteration for Preccinsta 110M on MYRIA

of the cost of both fine and coarse grid iterations, without a significant increase in the number of the formers. It has been shown how, for larger grids, the cost of the single fine grid iteration scales almost perfectly. The number of such iterations which is necessary to converge the system, however, increases. On the other hand, the number of deflation iterations necessary to reach convergence for a single fine grid iteration is reduced, but each iteration costs about twice the amount of time of one on the smaller mesh for the same value of NELEMENTPERGROUP. This could possibly shift the best value of NEPG towards larger values for even larger meshes in order to reduce the number of expensive iterations on the coarse grid.
The model could be useful to devise a strategy to improve the performances of the DPCG solver. Focusing on the cost of the fine grid iteration first, Equation 2.15 and Equation 2.16 indicate that there is a direct proportionality between the number of nodes, pairs and ElGrps and the cost of the relative computation phases. Equation 2.9 and Equation 2.10, together with Figure 2.26, strengthen the proportionality between the number of duplicated nodes and pairs and $n_{ElGpr_{[gt]}}$. As debated above, a reduction in the number of groups by increasing their size would comport then a reduction in the cost of the computation phases of the fine grid iteration. Concerning the communication phases, the values of the coefficients for Equation 2.17 would suggest that a larger number of workers would be beneficial to reduce the cost of the collective communication, which is nonsensical, as further confirmed by the same analysis on the larger grid. The time spent in the collective communications in the fine grid iteration is then to be considered as a constant value independent of $n_{workers}$ but influenced by the grid size. Finally Equation 2.18 indicates that a major contribution to the cost of the point to point communication is possibly linked to size of the internal communicator, which again is proportional to the number of ElGrps. Equation 2.19 is less interesting as the model is not based on the actual numerical implementation of such phase and does not work for the larger grid.
Concerning the coarse grid iteration, Equation 2.20 and Equation 2.21 are not interesting to analyse as they express the cost as a direct function of the amount of work to be performed, and not much can be done to modify such relation. Contrarily to what happens in the fine grid iteration, here the time spent in each collective communication is comparable and even higher than that spent in both computation phases in the iteration. This means that reducing the cost of communication could bring huge gains in term of performance of the entire solver. Equation 2.22 expresses the cost of the collective communication as directly proportional to the number of workers involved and the difference between the maximum and average number of ghost communicators. This indicates two possible strategies to optimise this operation. Firstly, a lower amount of workers involved in the parallel exchange would reduce the overall cost of the operation. Secondly, a more uniform number of neighbours would help reduce the synchronisation time between the processes. Figure 2.49 gives a rough estimation of possible gains in this communication when removing the imbalance and using half the workers in the exchange. Equation 2.23 is misleading as it gives a good estimation only of the average time spent in the

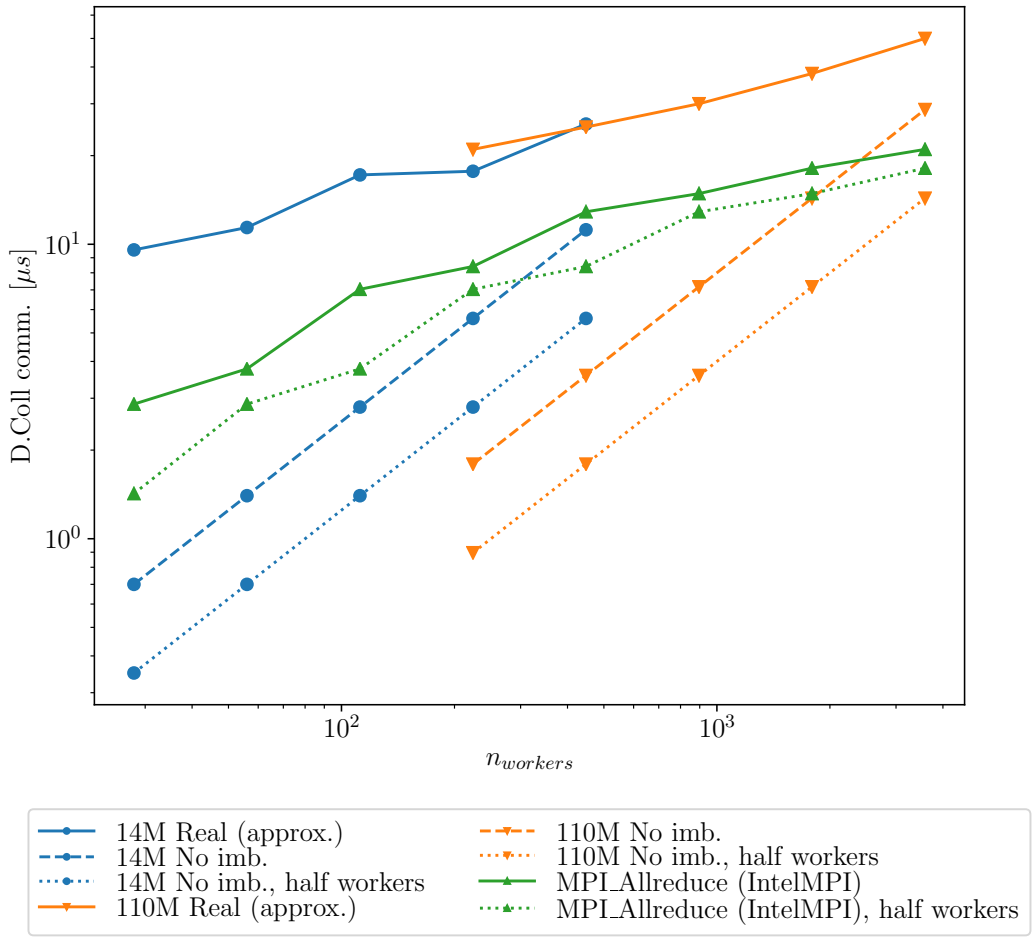


Figure 2.49: Estimation of the gain in the deflation collective communications removing the imbalance and with half of the workers participating in the communication. In green, the cost of a pure `MPI_Allreduce` which represents the minimum cost of the deflation collective communication.

ghost exchange. However, Figure 2.40 shows that there is a large standard deviation across the different workers, which is accounted for in Equation 2.22. The more uniform number of ghost communicator would then indubitably benefit the P2P exchange as well. Equation 2.24 combines considerations already made for other equations, hence this phase would benefit from the same improvements as well. As a meaningful example, Figure 2.49 compares the current (real) time spent in the deflation collective communication with what the model predicts would be the ideal communication time without the imbalance and halving of the participating MPI ranks. This prediction is compared with the real cost of an `MPI_Allreduce`, which is the actual minimum cost that this phase could reach. The model clearly overestimates the possible gain, due to the fact that it's based on a too simplistic linear dependency on $n_{workers}$ while the real cost seems to follow a logarithmic relation. Nonetheless this figure shows how removing the imbalance could save up to 70% on

communication time on this phase, and reducing the number of workers could bring the gain to over 80% in some cases. To resume, the performance model has allowed to deduce that:

- The cost of the computation in the fine grid iteration could be decreased by a reduction in the number of duplicated nodes and pairs, which is achievable using larger groups of elements;
- The cost of the P2P exchange in the fine grid iteration would also benefit from larger groups, as this would mean to have a smaller internal communicator;
- Communications are the largest contributors to coarse grid iteration cost;
- Reducing the number of workers participating in the parallel exchange would help to sensibly reduce the cost of the collective communication;
- A more uniform number of ghost communicator across the different workers would reduce the synchronisation time for the collective communication, while reducing the imbalance in the cost of the P2P exchange as well;
- A trade-off value for NELEMENTPERGROUP has to be used as the number of ElGrps links the performance of the different parts of the solver through the number of iterations on the fine and coarse grid.

2.3 Conclusion

This chapter introduced the code YALES2 on which the entirety of this work is based. In particular, its underlying data structures have been presented, with special focus on the groups of elements resulting from the double domain decomposition and the mechanisms used to communicate among different processes in a parallel computation. The implementation of the DPCG algorithm, and in particular the deflation phase have been described. This chapter also introduces the two benchmarks and the different architectures used to test the performances of the code. Some measurements on one of such benchmarks have been presented to characterise the code and to establish a first assessment of the code performances. Finally, several models have been defined for the performance of the Preccinsta benchmark on the Myria cluster. The grid model allows to determine all useful quantities related to the grid size from only 3 known parameters, with the exception of the size of the internal communicator. The DPCG algorithm has been modelled in two distinct parts: the iteration on the fine and coarse grids. A model has been obtained for each part as a sum of sub-models for the respective communication and computation phases. While the iteration time on the fine grid can be predicted with fair exactness, the deflation has proved to be harder to model with the same precision. In particular, both communication phases are highly dependent on the number of neighbours and on the imbalance in this number across processes. These parameters are much more difficult to model a priori, as they depend on the grid partitioning.

These models are just a first and rudimental attempt at predicting and studying the performance of the code. A more detailed analysis of each of the sub-models is required to provide a better justification for the models themselves and the values of the coefficients that have been obtained. However, in spite of their roughness, they have provided a useful insight in the mechanisms of the DPCG algorithm. In particular, it was shown that the actual MPI exchange has no influence in the cost of the P2P communication of the fine grid iteration, which in fact depends exclusively on the size of the internal communicator. Furthermore, the collective communications seem to have a fairly constant cost, which is independent of the number of workers in the communication. Regarding the deflation, the model showed that ghost communicator exchanges have a stronger dependence on the number of neighbours rather then the amount of data that is exchanged. Most importantly, the cost of the collective communication in the deflation can be approximated quite well by the difference between the maximum and average number of ghost communicators and by the number of workers involved in the communication.
Concerning the influence of NELEMENTPERGROUP on the global performance of the DPCG algorithm, it was shown how the optimal value of this parameter is determined as a compromise on two contrasting effects: the duplication of nodes and pairs against the number of fine grid iterations. Having bigger groups reduces the amount of duplicated data, but also coarsens the deflation grid, which in return provides a less accurate solution, hence more fine grid iterations are needed to reach convergence. This implies that two aspects of the DPCG algorithm should be improved. First, the number of "nodes" on the the deflation grid ($n_{ElGrp_{[gt]}}$) should be independent of the size of the groups of elements on the fine grid. This would allow another degree of freedom to better tune the convergence of the deflation grid, giving the possibility to decouple the number of iterations on the two grids from the duplication of nodes and pairs and the size of the internal communicator. As a consequence, larger groups could be used on the fine grid to reduce such duplication, while a less coarse grid can be used to regulate the number of iterations on both the fine and deflation grid. Second, some work should be done to reduce the cost of the communications in the deflation iteration. A more uniform number of ghost communicators would reduce the imbalance in the P2P exchange, which in turn would decrease the synchronisation time in the collective communication. Finally, reducing the number of processes involved in the collective exchange would sensibly reduce the cost of the latter, as clearly shown in Figure 2.49.

Chapter 3

A new data structure for the deflation

In 2.2.6 some of the limitations of the actual implementation of the DPCG algorithm in YALES2 were brought to light. In particular it was shown that there is a need of decoupling the deflation grid from the cache blocking mechanism of the ElGrps, whereas Figure 2.34 exposed the bad communication patterns due to the current connectivity construction. Even if the latter could have been easily corrected with a full-halo communication system for example, the former one calls for a complete re-organisation of the deflation grid. Consequently a new data structure called *graph* has been introduced in YALES2 to face all these negative aspects.
This chapter is organised as follows: first, a description of the new data structure is given, explaining how this is built on top of the grid; then the performances of the code with the deflation on the graph and on the ElGrps are compared. Successively, an analysis similar to the one done in 2.2.5 fot the ElGrps is done for the deflation on the new structure. As a result of this analysis new limits are exposed, and improvements are proposed to overcome these new performance barriers.

3.1 The graph data structure

The peculiar needs of the code, for which this new data structure should be quickly created or destroyed or re-partitioned in case of a change in the underlying mesh, and the evident similarities between a grid and a graph, where nodes and pairs on the first can be assimilated to vertices and edges on the second, made this choice almost obvious. The notation $\mathcal{G} = (V, E)$ is used to represent a graph as a set of interconnected objects called vertices V, whose pairwise connections are given by the set of edges E. A graph is called directed when its edges have a direction, otherwise it is undirected. As an example, if the edge $E_{(0,1)} = V_0 \rightarrow V_1$ connecting the vertices V_0 and V_1 is different from the edge $E_{(1,0)} = V_1 \rightarrow V_0$ connecting the vertices V_1 and V_0 then the graph is directed, otherwise, if $E_{(0,1)} = E_{(1,0)}$ the graph is undirected. There is an entire branch of mathematics called graph theory that studies the properties of graphs, however for the scope of this work these basic notions will suffice.
The choice on how to implement this partitioned graph data structure was dictated mostly by the needs of the deflation algorithm, but also by the fact that YALES2 relies on libraries such as SCOTCH [90] and METIS [91] for graph partitioning. A direct compatibility with these libraries is consequently primordial for an agile

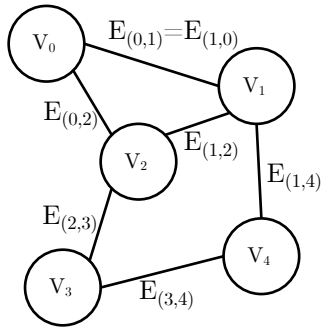


Figure 3.1: Example of a simple undirected graph

interfacing.
Given a graph $\mathcal{G} = (V, E)$ and a number of partitions p, the main objective of a graph partitioner is to subdivide the vertices V of $\mathcal{G}$ in p parts, as balanced as possible according to a certain weight given to each vertex. Usually, in order to evaluate which of the different possible partitions is better, the edge cut is minimised, i.e. the number of edges E which are "cut" by different partitions. This is more or less equivalent to a minimisation of the amount of data that has to be exchanged between partitions. An exhaustive overview of different partitioning methods is given in [92].

3.1.1 Building the graph

The entire data structure could be represented by two connectivity arrays of length equal to the number of edges n_E. These two arrays represent respectively the first and the second vertex of each edge composing the graph. This information, plus its actual number of vertices n_V would be sufficient to represent the graph. However, in order to be able to perform more operations, and to be directly passed to the partitioning libraries without further manipulations, the connectivity between vertices through their edges and other information is also added. Each graph has also a system of buffers called ghost communicators in order to be able to perform parallel data exchanges. Also, each vertex has been assigned a colour, in the same way as the groups of elements on the mesh. In YALES2 the graph structure is implemented as undirected. The edges are present only once and always go from the vertex with the lowest colour to the one with the highest. For some operators however, the direction of the edge might be important, especially when transferring data from the pairs of nodes on the grid to the graph edges to build the deflation operators. A supporting data mask is created on the grid to indicate whether an inversion of a pair must be performed during the data transfer to its corresponding edge.
In order to build a deflation graph from an already partitioned grid, a first graph

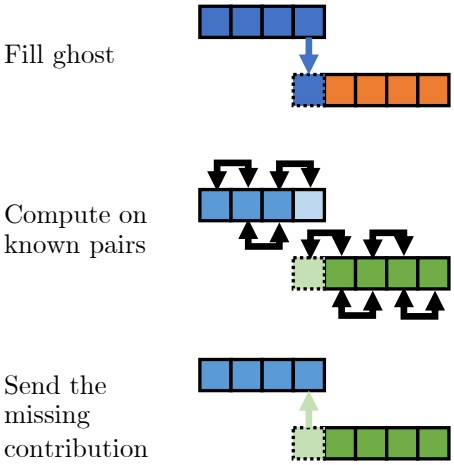


Figure 3.2: Schematic representation of the half-ghost communication pattern

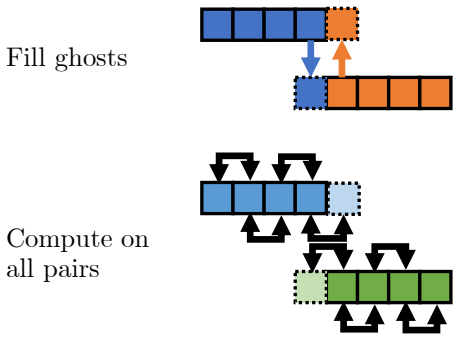


Figure 3.3: Schematic representation of the full-ghost communication pattern

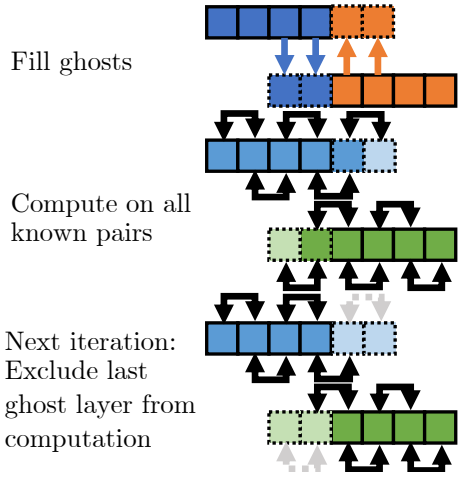


Figure 3.4: Schematic representation of the multi-layer ghost communication pattern

representing the entire partition has to be built. This operation can be quite slow as the connectivity among all nodes of the partition has to be rebuilt. It is important to remark that, while the first partitioning is based on the connectivity of the elements of the grid, this one is done on the actual control volumes that are used for the computation. The NELEMENTPERVERTEX (NEPV) parameter, similar to NELEMENTPERGROUP, can be set to tune the size of each vertex of the final graph. Although the graph is based on the nodes connectivity, the size of the vertices is computed from the number of elements, to maintain consistency with the ElGrps. Since the graph partitioner needs as input a number of parts, it is of no influence whatsoever the way such number is computed. Once the graph has been partitioned, the connectivity between the (fine) grid and the graph must be created in order to be able to build the deflation operators later on and to exchange data between the two. The ghost vertices and edges are created in the same way as for the deflation grid based on the ElGrps, i.e. based on the colour of the vertices. However a supplementary step is added in which each process informs its neighbours of all the edges it has in common with them, this way the ghost structure is now symmetric. This new ghost structure has a negative impact on the computation time for some processes, as those who did not have any ghost groups before now have some additional edges to compute, while there is no improvement for those who already had ghost groups. Nonetheless it was shown that the advantage of performing less computation was cancelled out by the waiting time in the following communication phase. Thanks to this symmetric structure, processes can now fill each other ghosts vertices with the value of w at the beginning of each deflation iteration and then compute everything independently from each other, until the collective communication. A scheme of the new pattern can be found in Figure 3.3.

3.1.2 Graph size model

As for the grid size, a model can be built to determine the graph size as a function of NELEMENTPERVERTEX, as shown in Figure 3.5. The number of vertices n_{Vertex} can be estimated the same way as n_{ElGrp}, with Equation 3.1:

$$\begin{aligned} n_{Vertex} &= ceil\left(\frac{ceil\left(\frac{n_{elem_{[gt]}}}{n_{workers}}\right)}{NEPV}\right) \simeq \frac{n_{elem_{[gt]}}}{n_{workers} \times NEPV}\,, \\ n_{Vertex_{[gt]}} &= n_{Vertex} \times n_{workers}\,. \end{aligned} \tag{3.1}$$

As for the ElGrps, this model for number of vertices accounts only for the internal ones, i.e. does not consider the ghost vertices which are added for the communication. This is done as only the internal ones contribute to the computation time. The number of edges n_{Edge} however, has to take into account both internal and ghost ones, as computation is performed on both kind. In order to model it, a simple linear relation with n_{Vertex} is sought with Equation 3.2:

$$\begin{aligned} n_{Edge} &= \beta_{Edge/Vertex} \times n_{Vertex} \\ \beta_{Edge/Vertex} &= 9\,. \end{aligned} \tag{3.2}$$

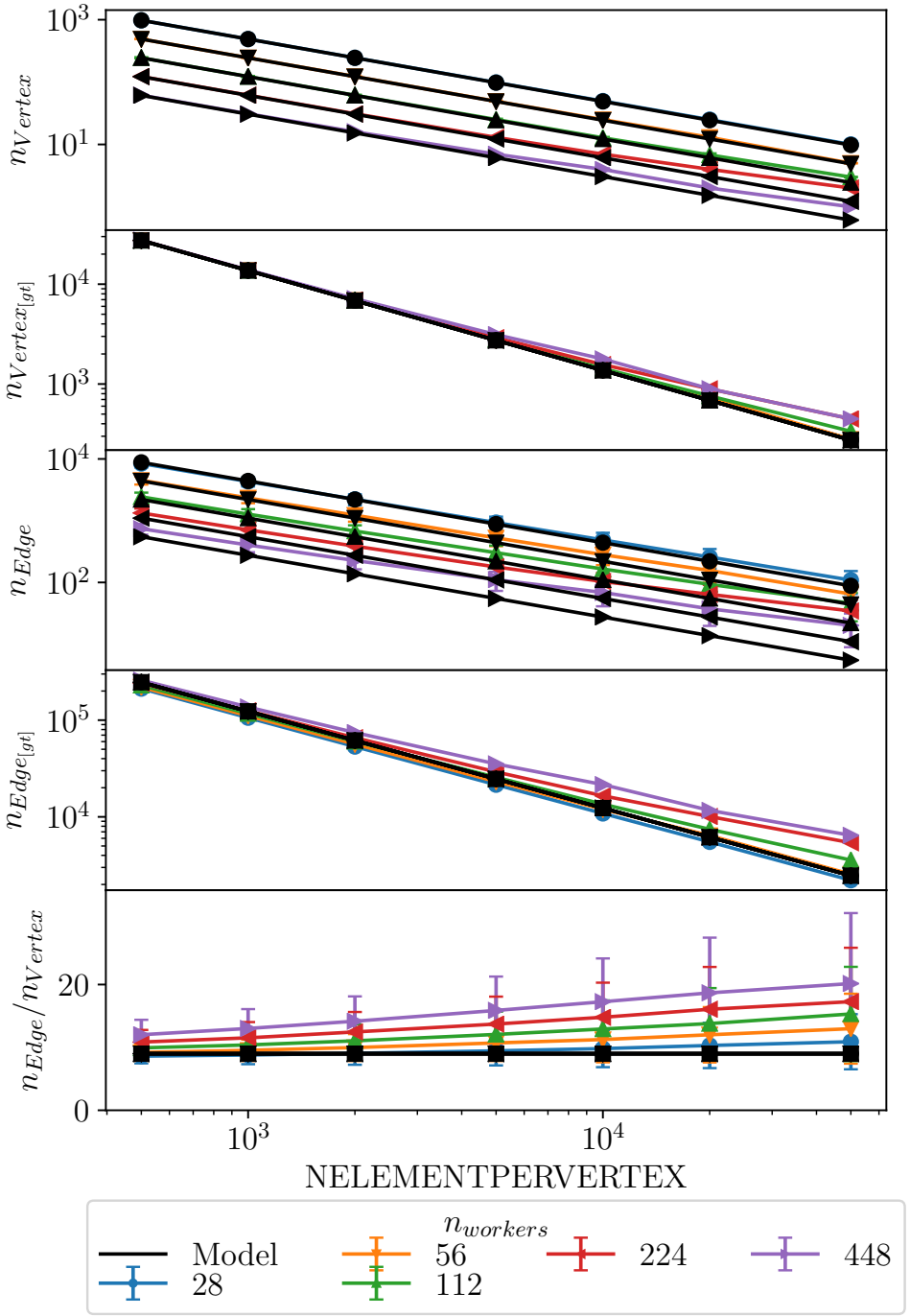


Figure 3.5: Model for the graph size applied to Preccinsta 14M

Figure 3.5 shows that, however simplistic, the two models are in good agreement with the measurements. On the bottom plot of Figure 3.5 it is possible to see that the ratio n_{Edge}/n_{Vertex} is far from constant. However, there is also a large standard deviation for that ratio across the different workers, and the model predicts sufficiently well the average value.

3.2 Deflation performance on the graph data structure

A parametric study has been conducted in order to find the combination of values of NELEMENTPERGROUP and NELEMENTPERVERTEX that provide the lowest return time for the DPCG solver. The results of this study are presented under different forms in Figure 3.6 and the best values are tabulated in Table 3.1. In the figure NELEMENTPERGROUP (NEPCG) is used to indicate the parameter to set the size of the groups on the fine grid, while NELEMENTPERVERTEX (NEPV) has been used to indicate the size of the deflation vertices and, with abuse of language, also the ElGrps when these are used for the deflation as well. The USE_GRAPH_DEFLATION flag is used to indicate whether the coarse grid of ElGrps was replaced by the graph or not.
The complete 3D space of measures is shown on the top of Figure 3.6. Here each point is coloured by the value of the RCT of the corresponding measure. This global view allows to see, for each value of $n_{workers}$, the region of values that provide better performance. In the plots right below, a dimension is removed taking the minimum value of RCT for each parameter. This allows to produce a 2D map of the best values for the two parameters. This picture proves that decoupling the concept of ElGrp and the deflation grid helps to improve the performance of the DPCG solver. The maps corresponding to the ElGrp deflation show a minimum value for NELEMENTPERGROUP ≈ 2000, in accordance with what was seen in 2.2.5. Conversely, the contour plots representing the deflation on the graph show that the best results, which are better than those on the ElGrps, are obtained for much larger values of NELEMENTPERGROUP, but lower values of NELEMENTPERVERTEX. From these contour plots it can be appreciated that the sensibility of the performance to the value of these parameters is not very high, as a wide range of values results in similar performance, especially for the regions where performance is optimal. The last plot shows the minimum RCT value obtained for each of the two methods, for all values of $n_{workers}$. These values are reported, together with the couple of parameters that allowed such result in the Table 3.1. Even if its benefit gradually shrinks for higher worker counts, the decoupling of the ElGrps from the deflation allowed by the new data structure always allows better overall performances for the DPCG solver.
Figure 3.7 shows that the model obtained in 2.2.6 for the number of iterations on the fine grid remains valid also for the graph, without altering the value of the coefficient. This figure is obtained for those cases with NELEMENTPERGROUP=2000, but other values produce equivalent pictures as the size of the groups could influence the number of iterations by a few only through round-off errors due to the IC update.

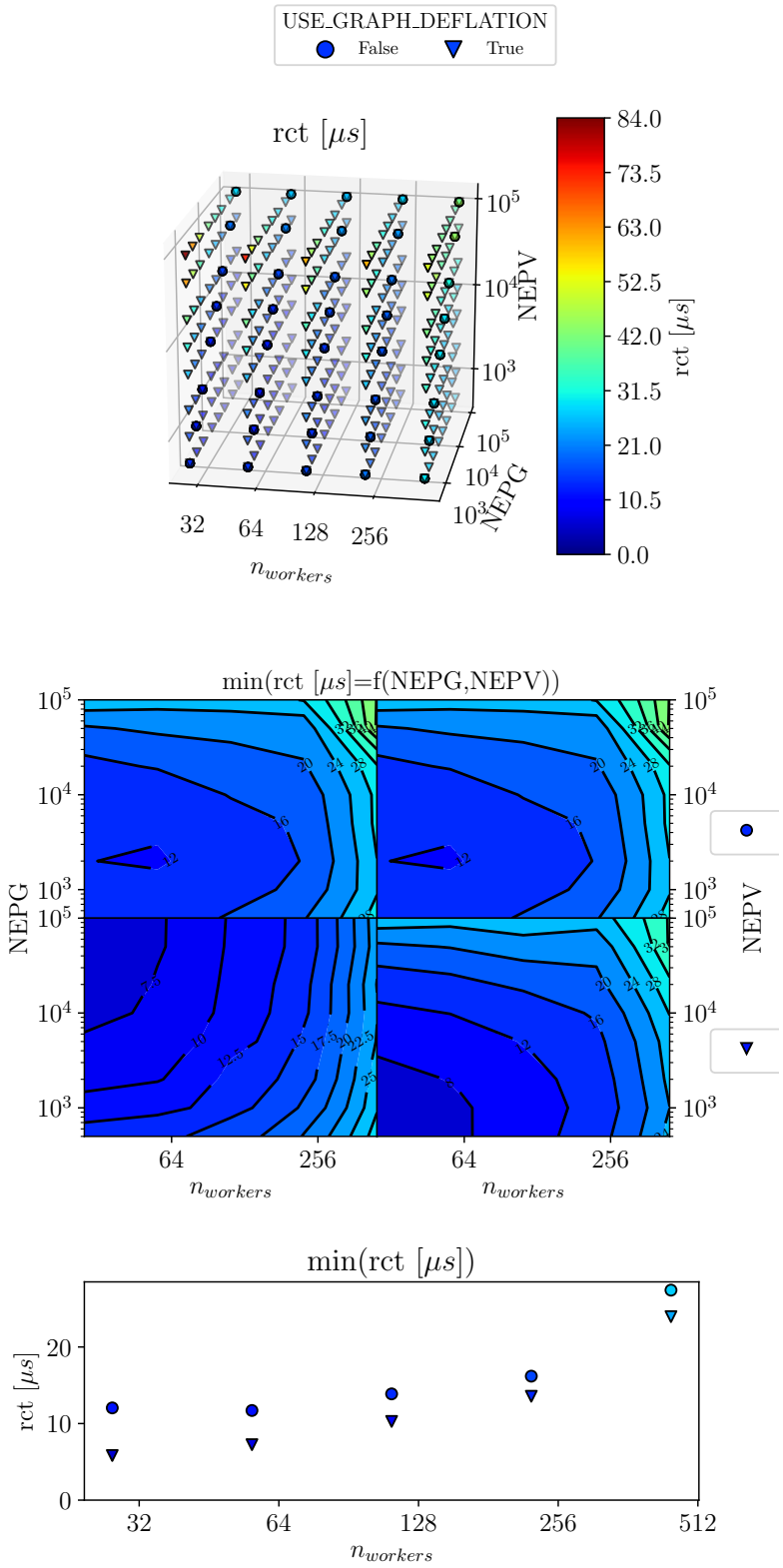


Figure 3.6: NELEMENTPERGROUP vs NELEMENTPERVERTEX analysis of the RCT of a DPCG iteration for Preccinsta 14M on MYRIA

$n_{workers}$	NEPG	NEPV	$rct\,[\mu s]$	USE_GRAPH_DEFLATION
28	2000	2000	12.0	False
28	100000	500	5.8	True
56	2000	2000	11.7	False
56	100000	500	7.2	True
112	2000	2000	13.9	False
112	100000	1000	10.3	True
224	2000	2000	16.2	False
224	100000	1000	13.6	True
448	1000	1000	27.4	False
448	50000	1000	24.0	True

Table 3.1: Synthesis of the best results for the parametric study presented in Figure 3.6.

Equations 2.12 and 2.13 are based on the number of ElGrps and the same principle remains valid for the number of vertices when the deflation is performed with the graph data structure. There is however a considerable difference for the number of deflation iterations. While the model seems to be acceptable for large vertices, the discrepancy for the small values of NELEMENTPERVERTEX is too great to consider this model still valid. More importantly, the total number of deflation iterations is not constant any more, but it is tends to increase with the size of the vertices.
The gain in the runtime can be explained so far by two combined effects: first, using small vertices allows to converge the system with only few iterations on the fine grid and a lower number of total deflation iterations; second, using larger groups is beneficial to the cost of the single fine grid iteration, as shown in Figure 2.39 for example. This was expected as the primary motivation of introducing this new data structure for the deflation was indeed to take advantage of such effects to reduce the DPCG iteration cost.

The other objective of the graph data structure was to improve the parallel performances of the code reducing the communication time in the deflation iteration. Figure 2.34 showed that the P2P exchange pattern imposed by the half-halo ghost mechanism had a negative impact on the overall time as it results in high synchronisation latencies in the collective communication. To avoid such pattern, the graph data structure has been created with a full-halo system that allows the P2P communication to be done at once and not in two steps. In order to understand if this modification is effective, the deflation iteration on the graph is analysed more in detail. Figure 3.8 shows that the models obtained for the deflation in 2.2.6 are still quite valid for the graph. ElGrps and ElGrp pairs have been substituted by vertices and edges in the respective computational models. Even though the coefficients have slightly different values, the dependencies of each phase remain the same, in particular the P2P communication is still fairly independent from the amount of exchanged

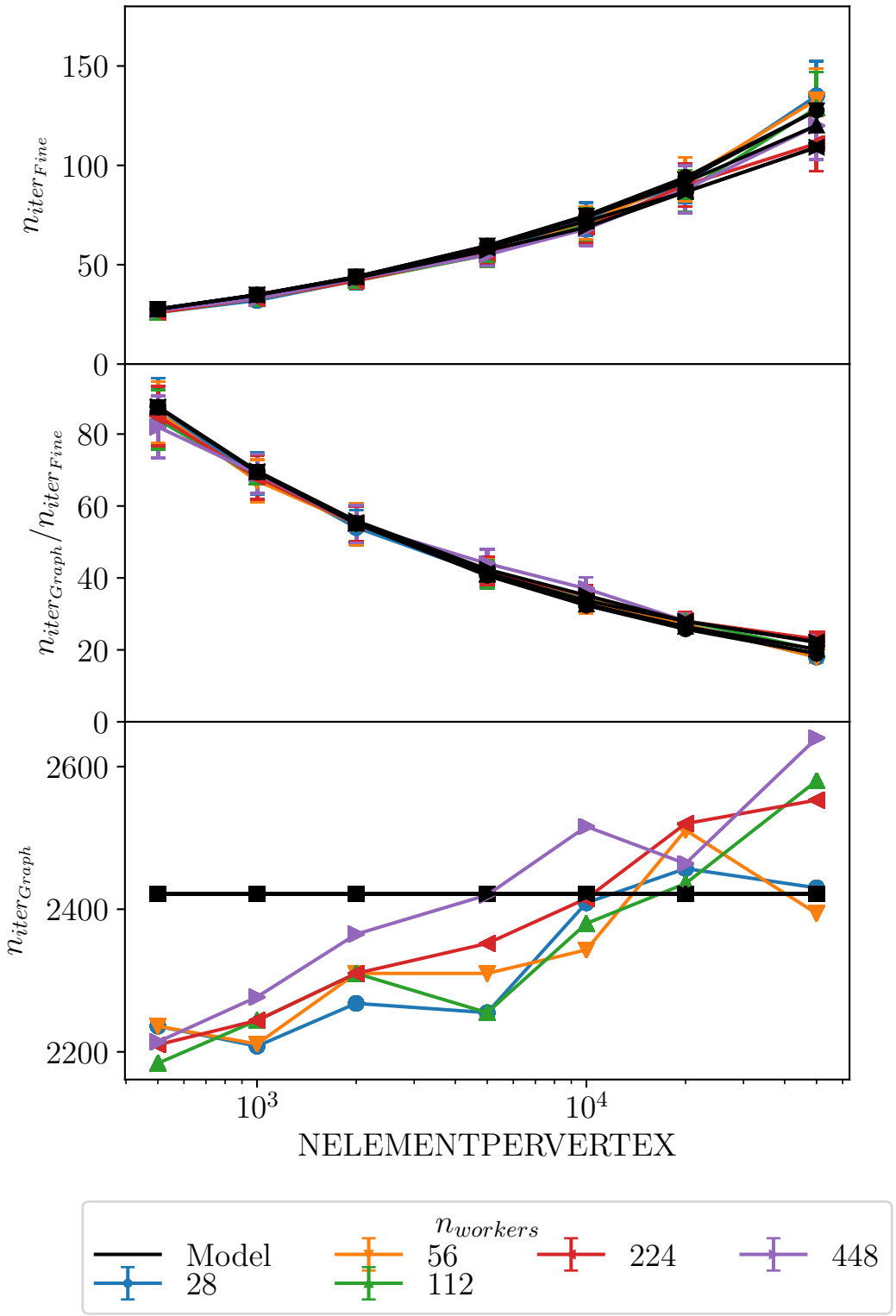


Figure 3.7: Model of the number of iterations on the fine grid and on the graph (NELEMENTPERGROUP=2000) for Preccinsta 14M

data and the collective communication is still much influenced by the imbalance in the number of neighbours for the ghost exchange. This gives a first indication that simply using a full halo system is not enough to improve the performance of the deflation phase. A trace of a few deflation iterations with the new full-halo mechanism is reported in Figure 3.9. This confirms that in spite of the fact that the point-to-point communication is more compact, those processes with more neighbours are always late with respect to the rest, resulting in the same synchronisation penalty on the collective communication. The symmetrical ghost communication fails then to provide a performance improvement over the asymmetrical structure seen for the groups of elements, and some further improvements are necessary.

3.3 Performance comparison

Figure 3.10 compares the performance of the DPCG algorithm with the two data structures for the deflation, for three different sizes of the Preccinsta mesh on the three architectures presented in 2.2.2. The graph allows better performance overall, however the improvement is much more remarkable for the lower worker counts, while for large amount of workers the performance is similar. The ElGrp performance was obtained for the optimal value of NEPG=2000 for all three meshes, while for the graph NEPG was always chosen to result in one ElGrp per worker and NEPV=1000 was used for the 14M mesh, and NEPV=2000 for the other two. These proved to be the values which give the best performance in all cases considered. For the Irene-SKL architecture on the 14M mesh, the speedup spikes for $n_{workers} = 256$ and $n_{workers} = 512$. This behaviour si due to a dramatic worsening in the performances of the ElGrp based deflation algorithm. Several tests were performed, however the bad performance seemed repeatable. Nonetheless those two values can be considered as outliers.
Figure 3.11 gives the comparison between the two methods breaking down the DPCG solver in its most important phases for IRENE-AMD. Similar figures can be obtained for the other platforms. The deflation phase is slightly more expensive for the graph on the 14M mesh: the smaller vertices imply a larger amount of iterations and consequently a larger time spent in this phase. This increased cost is however compensated by the reduction in the number of fine grid iterations, which is partially reflected in the reduction of the cost in the rest of the algorithm. Using only one ElGrp contributes as well to such gain. For the 110M and 878M meshes however, the deflation performances are quite similar as they are performed on a similar grid. The increase in performance is then only due to the reduction of the cost of the rest of the algorithm, coming from using only one ElGrp. There is also some gain it the collective communication of the fine grid solution, however this is less important as its cost is at least one order of magnitude lower than the other phases. As previously underlined, the performance of the deflation step itself has not been improved by the graph data structure, in spite of the full-halo data structure. Further improvements are then necessary to improve the deflation performances, especially its lack of scalability.

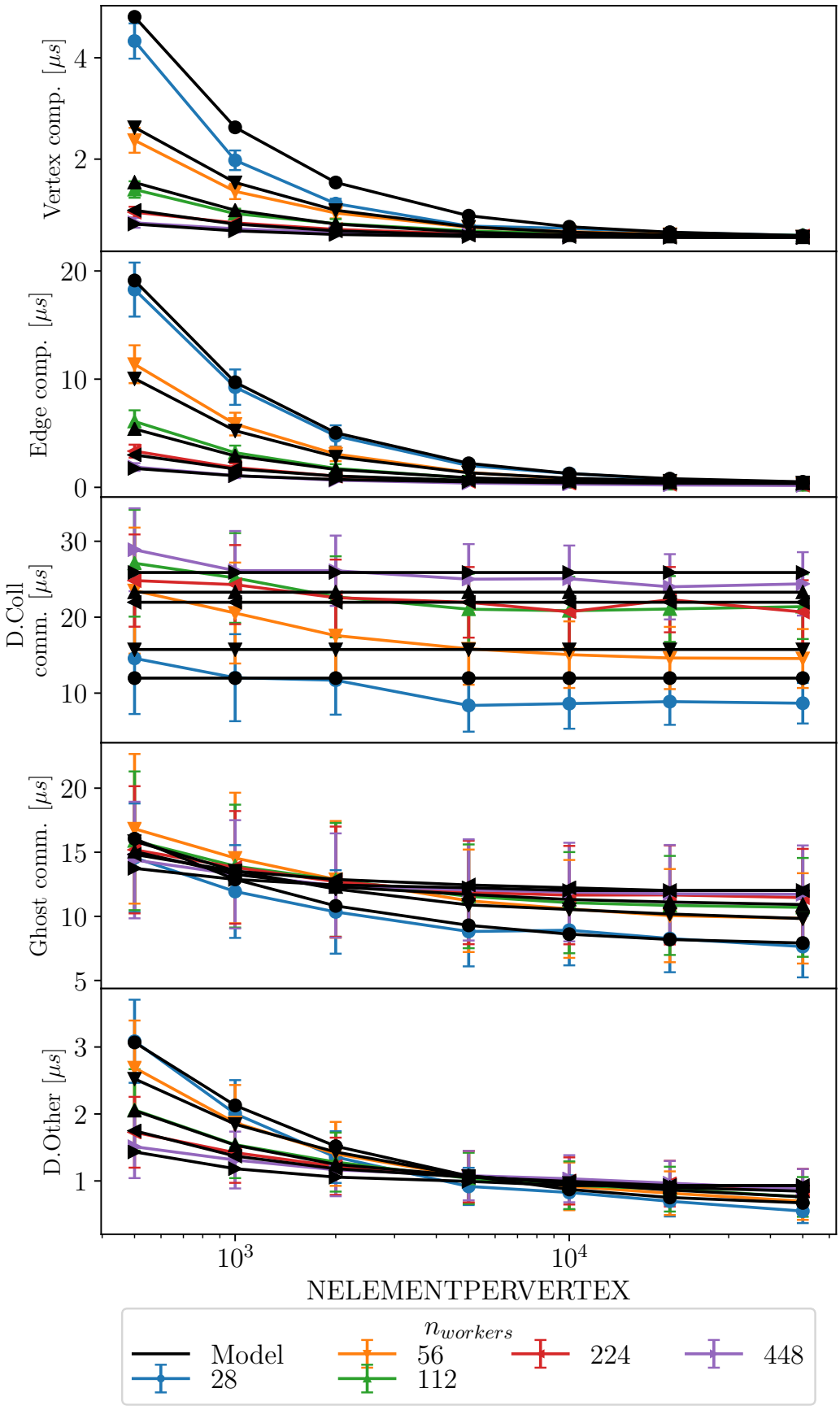


Figure 3.8: Breakdown of a deflation iteration on the graph in different phases with their relative models

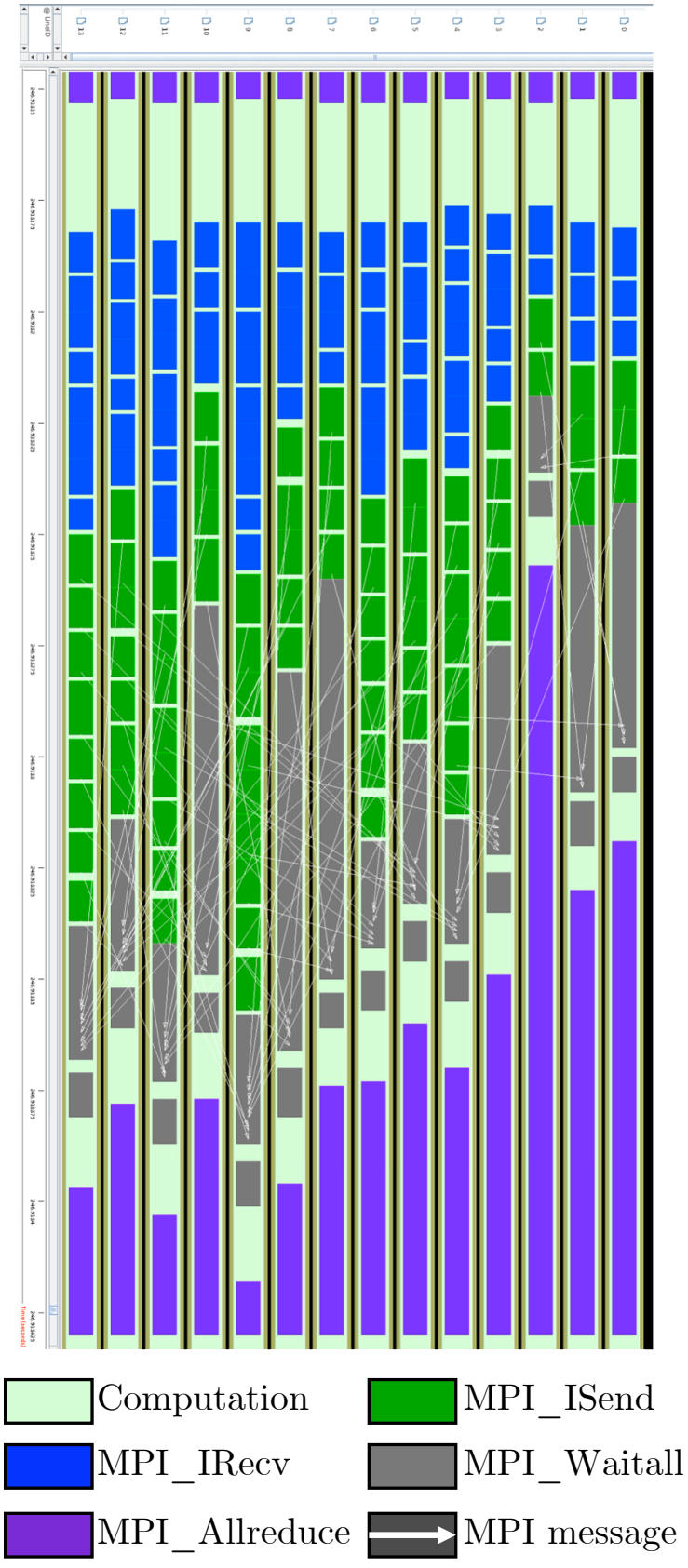


Figure 3.9: TAU trace of a deflation iteration with the graph data structure

It is clear this part of the algorithm is not expensive per-se, but the cost of its communications phases that continuously increases for larger workers prevents the entire code from scaling. In the following, two attempts to reduce the communication cost in the deflation are presented.

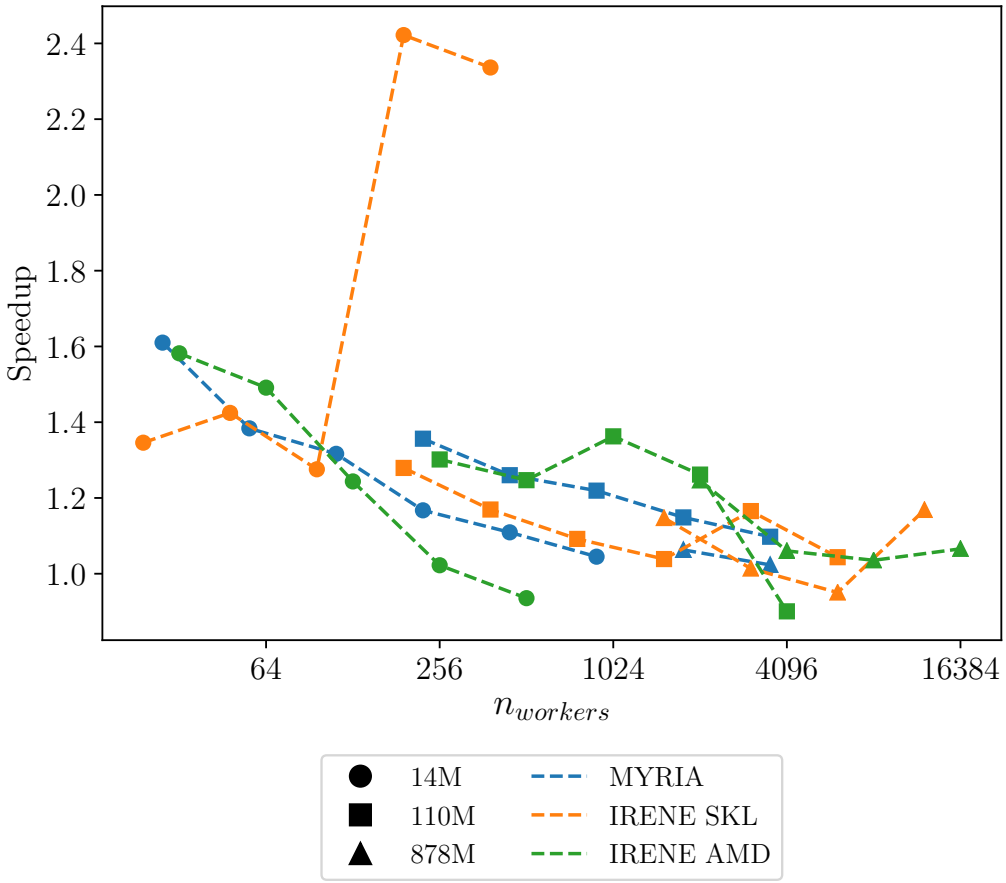


Figure 3.10: Speedup of the graph based DPCG algorithm with respect to the ElGrp one.

3.3.1 Deflation on gathered graph

The analysis of the deflation iteration done above, but also in 2.2.5 and 2.2.6, exposed the fact that communications are the predominant cost in this step of the DPCG solver, while computation accounts only for a minimal part. The first strategy put in place to improve on the performances of the deflation is to gather the graph on a reduced number of workers. This way only a subset of the total number of processes will participate in the collective communication, which should improve the scalability of the code. This come at a cost of an increased amount of computation for those processes assigned to the solution of the deflation problem as only such subset of workers computes and exchange data, while the others wait for the

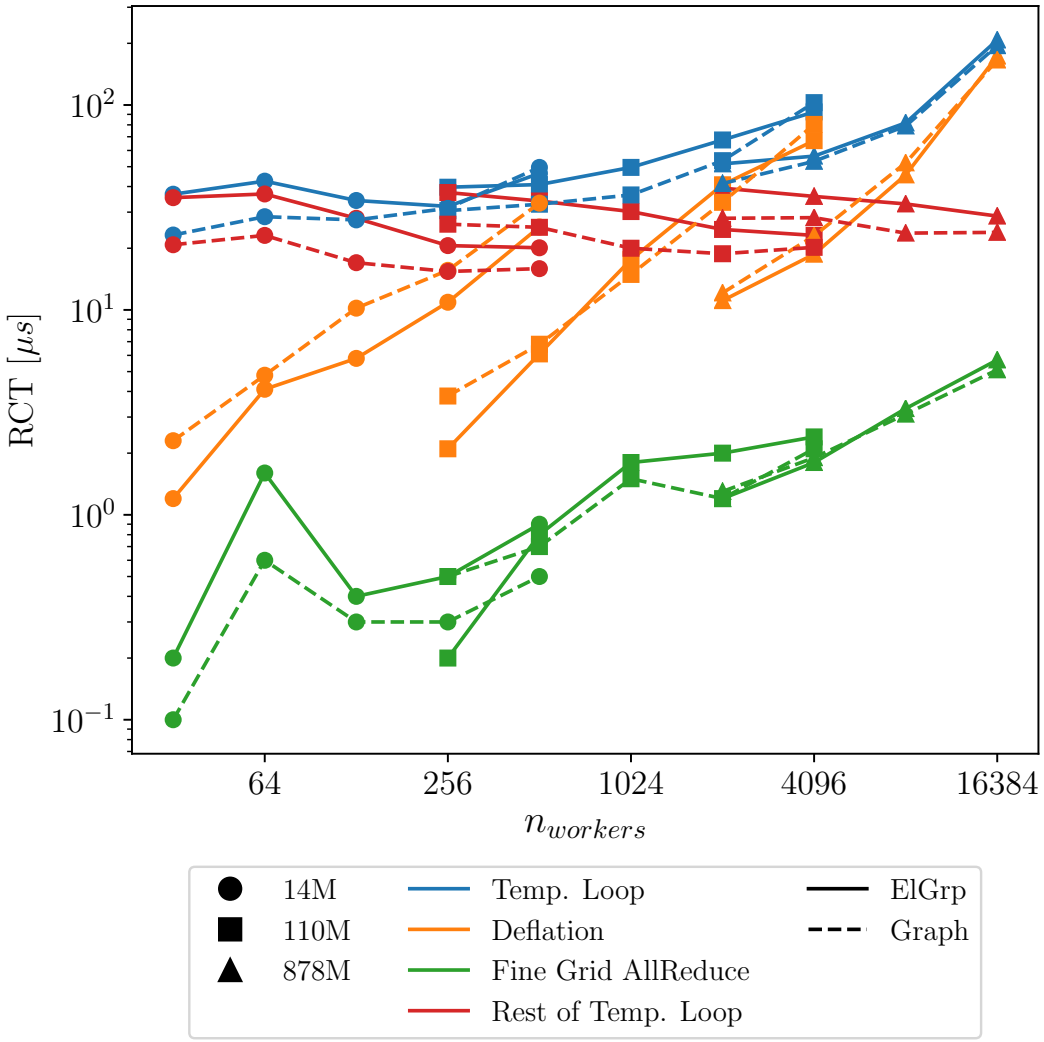


Figure 3.11: Breakdown of the performance comparison of the DPCG algorithm between the graph and the ElGrp based deflation on the IRENE-AMD platform.

computation to be finished and to receive the converged result. In principle this should not be a issue. The computation accounts only for a small percentage of the total cost of the deflation iteration, as shown in Figure 2.33, especially for a high number of workers, to which this particular development is addressed.
In order to build the gathered graph, first a graph of the connectivity of the different workers is created. This graph is then partitioned to decide the way the gathered graph is distributed.
The input parameter NVERTEX_PER_DEFL_WORKER, in short $n_{vertex/D.worker}$ sets the size of each gathered graph. If $n_{vertex/D.worker} \leq n_{vertex_{[gt]}}/n_{workers}$ then no gathering is necessary and the original graph is used, otherwise the gather ratio Γ is computed as $n_{vertex_{[gt]}}/\left(n_{workers} \times n_{vertex/D.worker}\right)$ and from this the new number of partitions $n_{workers_\Gamma}$ is obtained as $n_{workers}/\Gamma$.

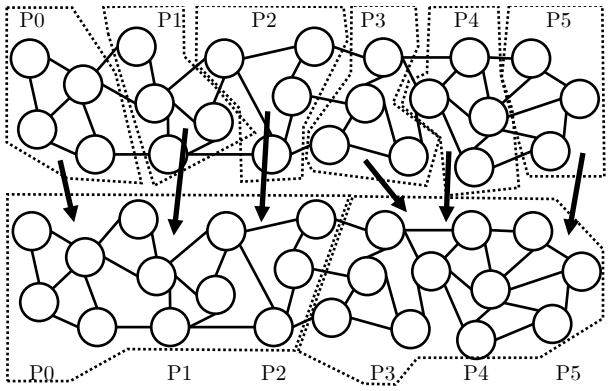


Figure 3.12: Scheme a graph gathered on a subset of processes

The gathering of the deflation graph is based on the partitioning of the workers graph due to the fact that minimising the edge-cut of the latter should imply a minimisation of the number of ghost communications, which is not true for the former. It also guarantees that the deflation graph of each worker is gathered by one worker only and it is not split among several, which reduces the cost of assembling the data on the gathered graph before the deflation step and to re-distribute it once the step is complete. Once this partitioning is done, those workers assigned to the deflation computation build the gathered graph agglomerating all the graphs of the other workers on the same partition and the ghost communicators on the new gathered graph are created. The workers picked to host the gathered graph are chosen according to a policy among COMPACT, SCATTER and PADDING. The policy can be set in the input file via the DEFL_WORKERS_DISTRIBUTION parameter. COMPACT means that the first $n_{workers_\Gamma}$ workers are assigned to the deflation, while SCATTER means that the workers are picked as far away as possible from each other. PADDING allows the user to chose the distance between the workers used for computation. Setting DEFL_WORKERS_DISTRIBUTION=PADDING and DEFL_WORKERS_PADDING=3, for example, means that one each three workers will be used, until $n_{workers_\Gamma}$ is reached. A COMPACT distribution should allow for faster communication as the computing cores are next to each other, while SCATTER gives increased memory bandwidth as the non computing cores close to the working ones are idle. PADDING allows a better control on the distribution.
Restricting the computation of the deflation algorithm to a subset of workers means that a dedicated MPI communicator must be created. This allows such workers to be completely independent from the idle ones. A dedicated MPI communicator is created as well for the gather-scatter operations of each partition,in order to simplify their management.
Data is exchanged between the grid and the gathered graph in two step. First the data on the grid is restricted on the deflation graph by each worker as usual, then the data is gathered by the workers dedicated to the deflation and copied on the gathered graph. These workers proceed to solve the deflation system, while the

rest awaits for the data to be scattered back on their graphs once the algorithm as converged. Then the data is projected back onto the grid an the DPCG algorithm continues as usual.
To be effective however, the increased computational cost on the reduced number of workers which work on the gathered graph must be matched by a reduction in the communication cost. The performance model for the deflation proposed in 2.2.6 should help to find a law to establish if the gathering of the graph is beneficial to the performance. Equation 2.20 and Equation 2.21 can be adapted to the graph deflation, obtaining respectively Equation 3.3 and Equation 3.4:

$$T_{Vertex}\left[\mu s\right] = \alpha_{Vertex} + \beta_{Vertex} \times n_{Vertex}\,, \tag{3.3}$$

$$T_{Edge}\left[\mu s\right] = \alpha_{Edge} + \beta_{Edge} \times n_{Edge}\,, \tag{3.4}$$

where $\alpha_{Vertex} = \alpha_{ElGrp}$, $\beta_{Vertex} = \beta_{ElGrp}$, $\alpha_{Edge} = \alpha_{ElGrpPair}$ and $\beta_{Edge} = \beta_{ElGrpPair}$. Equation 2.22 and Equation 2.23 can be applied directly, while Equation 2.24 can be adapted as well simply substituting $n_{ElgrpPair}$ with n_{Edge}.
Gathering the graph on a reduced amount of workers $n_{workers_\Gamma} = n_{workers}/\Gamma$ would increase the amount of vertices and edges on each of the $n_{workers_\Gamma}$ by the same factor Γ. There is no way to know beforehand whether the number of ghost communicators $\overline{n_{GC}}$ or the difference $\|n_{GC}\|_\infty - \overline{n_{GC}}$ are influenced by the gathering, and if so how, so they are supposed to remain unvaried for the scope of this analysis. In order for the gathering of the graph to be effective, the condition expressed in Equation 3.5 has to be verified.

$$T_{D_\Gamma} < T_D\,, \tag{3.5}$$

where T_D and T_{D_Γ} are respectively the time of a deflation iteration on the graph and on the gathered one. Applying the models and the reasoning above, Equation 3.6 is obtained:

$$\begin{aligned}
&\alpha_{Vertex} + \beta_{Vertex} \times \Gamma \times n_{Vertex} + \\
&\alpha_{Edge} + \beta_{Edge} \times \Gamma \times n_{Edge} + \\
&\beta_{1_{D.Coll}} \times \frac{n_{workers}}{\Gamma} + \beta_{2_{D.Coll}} \times (\|n_{GC}\|_\infty - \overline{n_{GC}}) + \\
&\overline{n_{GC}} \times (\beta_{1_{D.Ghost}} + \beta_{2_{D.Ghost}} \times \overline{n_{item_{GC}}}) + \\
&\overline{n_{GC}} \times (\beta_{1_{D.Other}} + \beta_{2_{D.Other}} \times \overline{n_{item_{GC}}}) + \beta_{3_{D.Other}} \times \Gamma \times n_{Edge} \\
&< \alpha_{Vertex} + \beta_{Vertex} \times n_{Vertex} + \\
&\alpha_{Edge} + \beta_{Edge} \times n_{Edge} + \\
&\beta_{1_{D.Coll}} \times n_{workers} + \beta_{2_{D.Coll}} \times (\|n_{GC}\|_\infty - \overline{n_{GC}}) + \\
&\overline{n_{GC}} \times (\beta_{1_{D.Ghost}} + \beta_{2_{D.Ghost}} \times \overline{n_{item_{GC}}}) + \\
&\overline{n_{GC}} \times (\beta_{1_{D.Other}} + \beta_{2_{D.Other}} \times \overline{n_{item_{GC}}}) + \beta_{3_{D.Other}} \times n_{Edge}\,.
\end{aligned} \tag{3.6}$$

All the contributions that remain unchanged between the left and right side of the Equation 3.6 can be eliminated, obtaining Equation 3.7:

$$\begin{aligned}&\beta_{Vertex} \times \Gamma \times n_{Vertex} + \beta_{Edge} \times \Gamma \times n_{Edge} + \\ &\beta_{1_{D.Coll}} \times \frac{n_{workers}}{\Gamma} + \beta_{3_{D.Other}} \times \Gamma \times n_{Edge} \\ &< \beta_{Vertex} \times n_{Vertex} + \beta_{Edge} \times n_{Edge} + \beta_{1_{D.Coll}} \times n_{workers} + \beta_{3_{D.Other}} \times n_{Edge}\,.\end{aligned} \tag{3.7}$$

The parameter Γ can be factorised as in Equation 3.8:

$$\omega \times \Gamma^2 - (\omega + 1) \times \Gamma + 1 < 0\,, \tag{3.8}$$

where ω is obtained as in Equation 3.9.

$$\omega = \frac{\beta_{Vertex} \times n_{Vertex} + (\beta_{Edge} + \beta_{3_{D.Other}}) \times n_{Edge}}{\beta_{1_{D.Coll}} \times n_{workers}}\,. \tag{3.9}$$

The value of Γ can be obtained for:

$$\begin{aligned}1 < \Gamma < \frac{1}{\omega}\,,\\ 0 < \omega < 1\,.\end{aligned} \tag{3.10}$$

For those cases in which $\omega > 1$ the value of Γ would be bound by $\Gamma < 1$, however, by definition, a necessary condition for the graph to be gathered is $\Gamma > 1$. To conclude, a value of NVERTEX_PER_DEFL_WORKER ($n_{vertex/D.worker}$) can simply be obtained with Equation 3.11:

$$n_{vertex/D.worker} = \Gamma \times n_{Vertex} \simeq \Gamma \times \frac{n_{elem_{[gt]}}}{n_{workers} \times NEPV}\,. \tag{3.11}$$

It is important to underline the fact that this analysis is based on extremely simple models and does not take into account the additional operations of reduction and projection of the graph at the beginning and the end of the algorithm. The analysis performed above also does not take into consideration the imbalance introduced by gathering the graph on fewer workers. As explained above, the first step in gathering the graph consists in creating a first graph of all the workers, and partition that one in $n_{workers_\Gamma}$ parts. This is done because the graph partitioner minimises the edge-cut, which for this graph means to minimise the number of ghost communicators. This however, has the downside to be extremely coarse grained, as the agglomeration factor Γ is quite small. Consequently, even though each worker has a weight equal to its number of vertices, imbalances present in the original graph can be amplified. This is particularly true for high values of $n_{workers}$ as these cases have only few vertices per worker and are more difficult to balance. A possible solution to the problem could be to add a step of load-balancing to re-equilibrate the graph after the partitioning or to split directly the complete deflation graph rather than to use the graph of the workers, measurements showed that this intermediate step does not guarantee a good balance in the number of ghost communicator. The performance

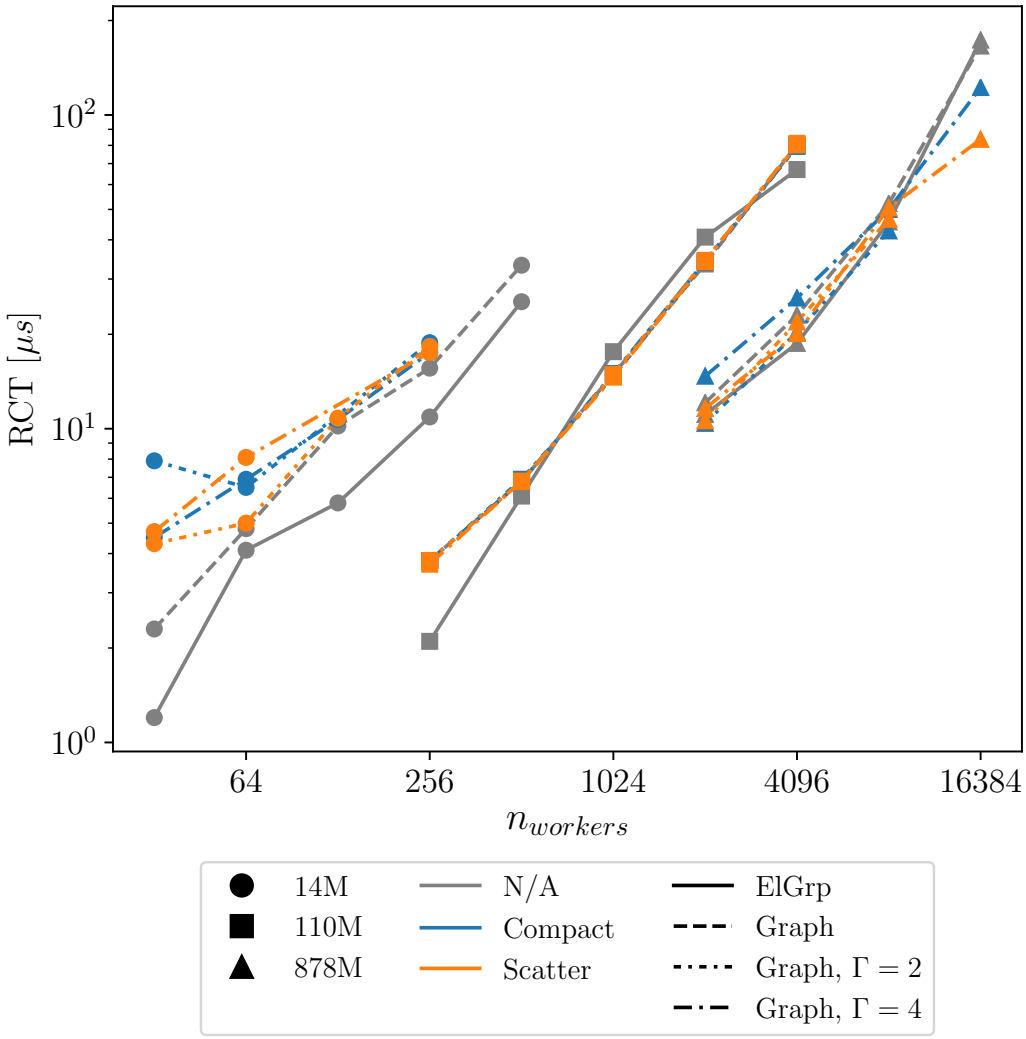


Figure 3.13: Performance of the deflation iteration for the gathered graph for Preccinsta on the IRENE-AMD platform.

of the gathered graph are shown in Figure 3.13 for the Preccinsta benchmark on the IRENE-AMD platform. The three meshes behave quite differently from each other. First, the gathering of the graph does not have a beneficial effect on the 14M mesh, on the contrary, the standard one has always better performances. For the 110M mesh there is no difference between all cases considered and their performance perfectly overlap. This indicates that for this mesh the deflation cost is mainly due to the ghost exchange and gathering the graph on a subset of workers does not reduce the number of exchanges and the communication imbalance. Finally, gathering the graph on the 878M mesh gives a small advantage only for $n_{workers} = 8096$ and $n_{workers} = 16384$. The two different workers distribution COMPACT and SCATTER have been compared as well. It is not clear whether one offers and advantage with respect to the other. However, for this particular platform,

SCATTER combined with the Γ values chosen, should provide an increased memory bandwidth and L3 cache available per core, similarly to what was seen in 2.2.4. The fact that in most cases this theoretical advantage does not translate into a performance gain confirms that the computation is mostly irrelevant. The gain obtained with the SCATTER distribution is substantial only for the 878M mesh and $\Gamma = 4$, particularly for $n_{workers} = 16384$.
Gathering the graph on a subset of workers did not provide the expected advantages, in particular it did not bring any benefit for the scalability of the deflation algorithm for the 14M and 110M meshes. Some improvement was obtained only for the larger amount of workers on the 878M mesh. This failure is due to the fact that using a subset of workers does not reduce the imbalance nor the amount of ghost exchanges, plus it could introduce some further imbalance in the number of vertices and edges on each deflation worker. Such imbalance then hides the possible benefit that can be obtained on the collective communication with fewer workers.

3.3.2 Multiple ghost layers

Although decoupling the group of elements and the deflation proved most effective in improving the performances of the DPCG solver, when the deflation vertex and the ElGrp size coincides, there is no performance improvement. This means that the symmetric ghost structure does not actually bring any tangible benefit. The trace in Figure 3.9 was obtained with the same configuration as the one in Figure 2.34, except that the deflation is computed with the new data structure. Looking at the new trace, it is evident that the communication pattern is much more compact, however there is still an important amount of imbalance between the different processes when they arrive at the synchronisation point imposed by the collective communication. The reason for this imbalance now is clearly the disparity in the number of neighbours. This is hard to correct as it depends directly on the initial mesh partitioning. Graph partitioners that minimise the number of neighbours and that are not oblivious to hardware topology can help to improve parallel performances [93].

Another possible approach would be to skip the ghost exchange completely. This can be achieved extending the ghost area of each processor beyond the first layer of vertices. If n_{layers} layers of ghost vertices are exchanged on the first iteration of the algorithm, then each worker would be able to compute independently for the following n_{layers} iterations without the need of a P2P exchange. At the end of each iteration the values on the most external layer of ghost vertices would be false, however all the other inner layers hold true results, consequently the computation can continue until all ghost layers are consumed and a new update is necessary, as schematised in Figure 3.4. Increasing the number of ghost layers is not used often in parallel computation as the first implication of this technique is that the amount of computation increases quite rapidly adding ghost layers. Nonetheless, it has been already shown in Figure 2.40 how computation is a minimal part compared to the cost of communication, and the deflation graph is sufficiently small that the number of additional ghost cells remains reasonable. Another consequence of avoiding

Algorithm 9: Multiple data update

```
for i ← 1 to n_neighbours do
    MPI_IRECV(recv_buffer[i],n_item[i] × n_data,neighbour[i],...);
end
for i ← 1 to n_neighbours do
    offset = 0;
    foreach data do
        Pack: data → (n_item[i]) → send_buffer[i][offset];
        offset = offset + n_item;
    end
    MPI_ISEND(send_buffer[i],n_item[i] × n_data,neighbour[i],...);
end
received = 0;
while received < n_neighbours do
    MPI_TESTSOME(n_neighbours,recv_request,completed,completed_request)
     if completed = 0 then
        MPI_WAITSOME(recv_requests,n_neighbours,completed,completed_request)
    end
    received = received + completed;
    foreach completed_request do
        offset = 0;
        foreach data do
            Unpack: recv_buffer [i][offset] → (n_item[i]) → data;
            offset = offset + n_item;
        end
    end
end
MPI_WAITALL(send_request,n_neighbours,...);
```

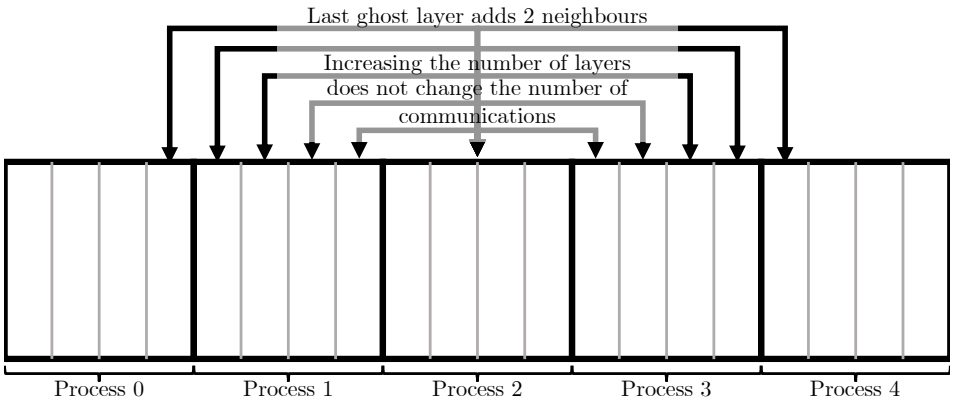


Figure 3.14: Scheme of a channel configuration to test the validity of the multiple ghost layer system

communication at each iteration is the increased amount of data that has to be exchanged when the point-to-point communication takes place. Not only more data needs to be transferred due to the larger ghost size, but more variables need to be updated as well. If with one layer only the value of w had to be transferred, now also p, q and x have to be part of the communication. This is due to the fact that the algorithm is written in such a way that these variables are self dependent, consequently their value must be correct on each ghost layer except the last on the first iteration after communication. The communication of multiple data is optimised as they are packed together directly into the exchange buffer and sent in the same message. As already discussed, the communication time was rather independent on the message size. In an attempt to overlap some communication and computation, the unpacking is performed for each communication as soon as it is received while waiting for the rest. This mechanism is reported in Algorithm 9.
To confirm that the idea of multiple ghost layers could work, an artificial test case is used. A 3D channel is split among a different number of processes along the x direction, and each process then splits its subdomain in several vertices. The test is configured in order to have only one vertex in the y and z direction, while the number of vertices along x can vary. This results in a mono-dimensional deflation graph, whose schematic representation is shown in Figure 3.14. The domain is made periodic in the x direction, consequently each worker has two direct neighbours plus those deriving from the extended ghost layers. Multiple tests are performed, changing the number of ghost layers and x-wise vertices. The first results of these measurements are synthesised in Figures 3.15 and 3.16, which show respectively the average time spent in each deflation iteration and its breakdown in different phases. Figure 3.15 shows that, for this benchmark, the best performance is given by $n_{layers} = 3$. In Figure 3.16 it is possible to see how the cost of the P2P ghost communication decreases as expected when increasing the number of layers. However, the increase in the cost of the computation is more important, and after $n_{layers} = 3$ it surpasses the gain in the ghost exchange. It is interesting to remark that skipping

the ghost exchange has no effect whatsoever in the cost of the collective communication. This is to be expected as this particular benchmark has been constructed in such a way that all the relevant quantities such as n_{vertex}, n_{edge} and n_{ghost} are perfectly homogeneous among all workers, hence without imbalance that negatively affects the collective communication. The seemingly erratic behaviour of the rest of the algorithm is due to the fact that this operation is performed every 10 iterations and it contains a ghost exchange. Consequently, such parallel communication inside the region is actually performed more or less frequently depending on the number of ghosts layers. Figure 3.17 shows the breakdown of the deflation iteration for different values of n_{layers} on the Preccinsta 14M mesh. In contrast with what was seen for the channel benchmark, Figure 3.17 shows that there is no significant reduction in the communication cost increasing the number of ghost layers, which actually constantly increases from $n_{layers} = 3$ onwards. In spite of the positive results observed on the synthetic benchmark analysed above, increasing the number of ghost layers to avoid point-to-point communication at each iteration does not improve the performance of the algorithm for real applications. There are two explanations for this. First, with respect to the 1D example, in 3 dimensions the quantity of supplementary vertices and edges for each additional layer is higher, hence the cost of computation increases more rapidly than what seen in Figure 3.16. Another explanation can be found in Figure 3.18, where vertices are coloured according to which ghost layer they belong to for a certain process. The image is taken from the 2D_cylinder benchmark for visualisation purposes, but the same principle applies in 3D cases as well. It can be seen quite clearly that the more ghost layers are added, the more processes are involved in the data exchange. Adding more "neighbours" to the ghost communication cancels out, or even overshadows the benefit of doing such exchange less often. Furthermore, all processes have an increased number of ghost communicators, consequently the amount of exchanged messages over the network increases exponentially. This effect has been exaggerated on purpose dividing a small mesh among many processes, leaving them with only few vertices each so that the ghost layers would propagate quickly. However this issue appears in most 3D real cases with reasonable vertices sizes due to the highly irregular shapes of the partitions.

3.3.3 Gathered graph with multiple ghost layers

Other than the rapidly increasing number of neighbours, Figure 3.18 highlights the fact that with multiple ghost layers, neighbouring processes solve approximatively the same problem. This means that data is duplicated multiple times, but also that the same data is sent by a process to multiple neighbours. A possible strategy to avoid these numerous repeated exchanges might be to exploit the gathered graph together with the multiple ghost layers. Having bigger deflation graphs on a smaller subset of workers means that extending the ghost layers would be less likely to imply an increase in the number of neighbours. The main downside is that with a larger frontier, the number of supplementary ghost vertices and edges that are included

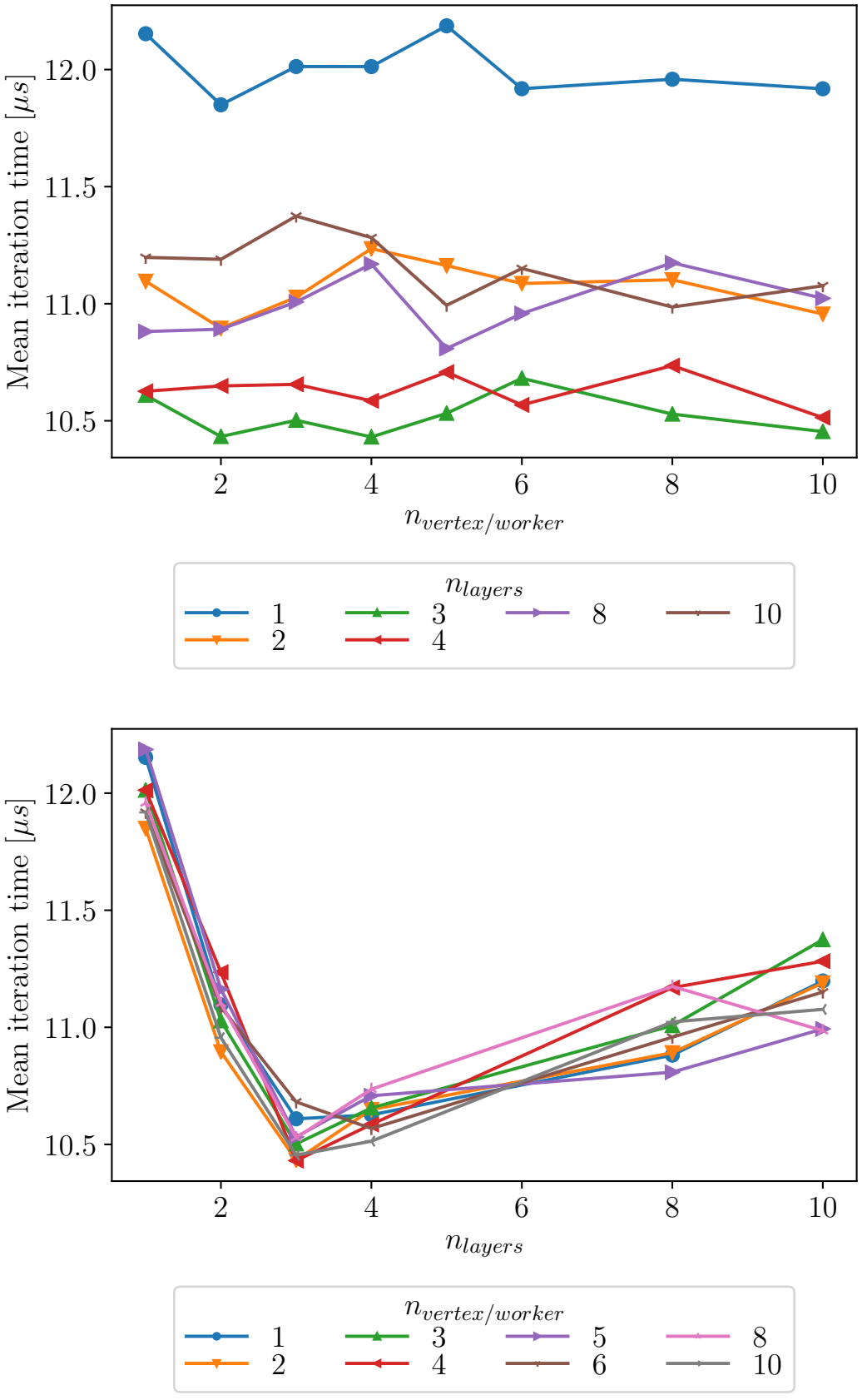


Figure 3.15: Average deflation iteration time with multiple ghost layers on channel benchmark

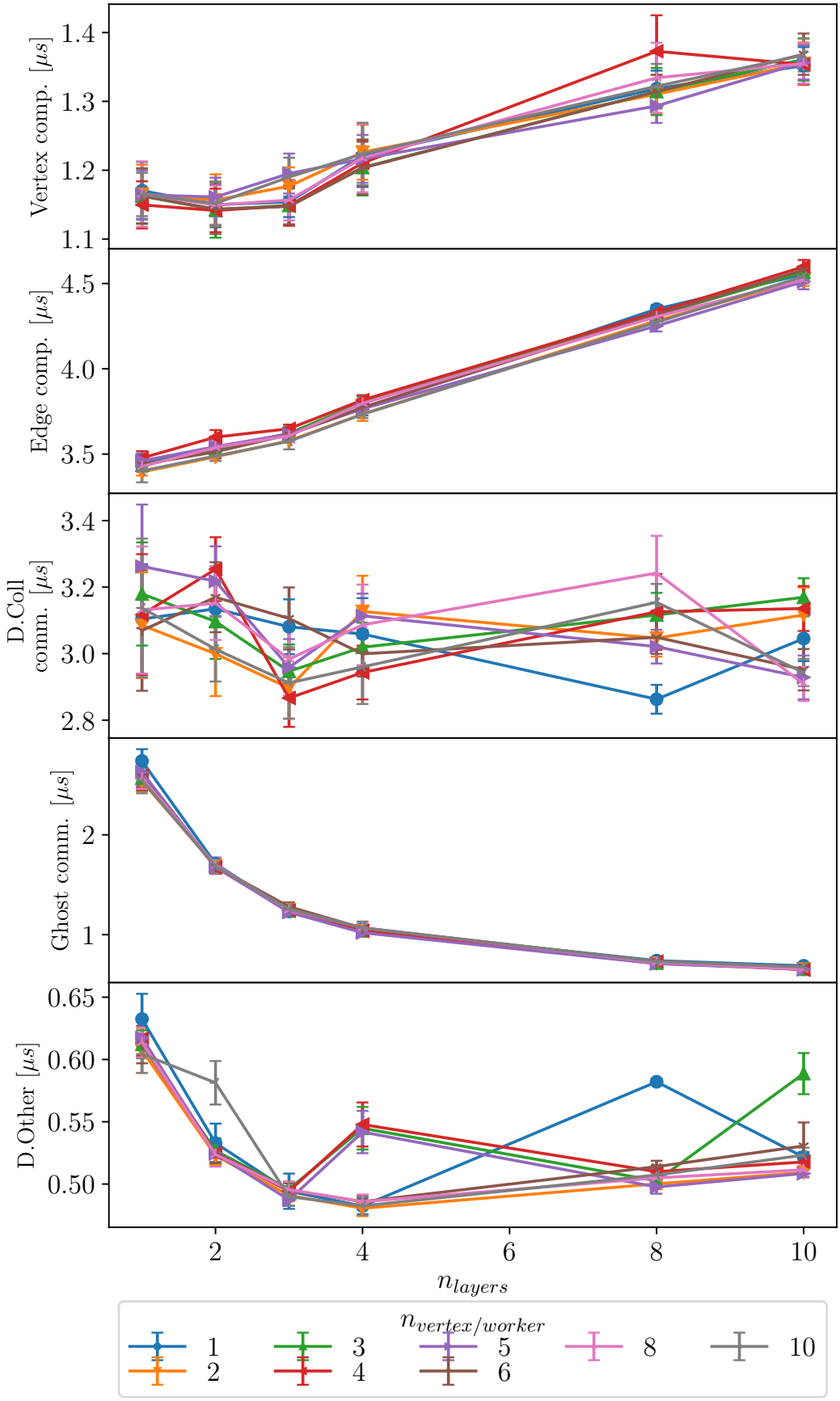


Figure 3.16: Deflation iteration breakdown with multiple ghost layers on channel benchmark

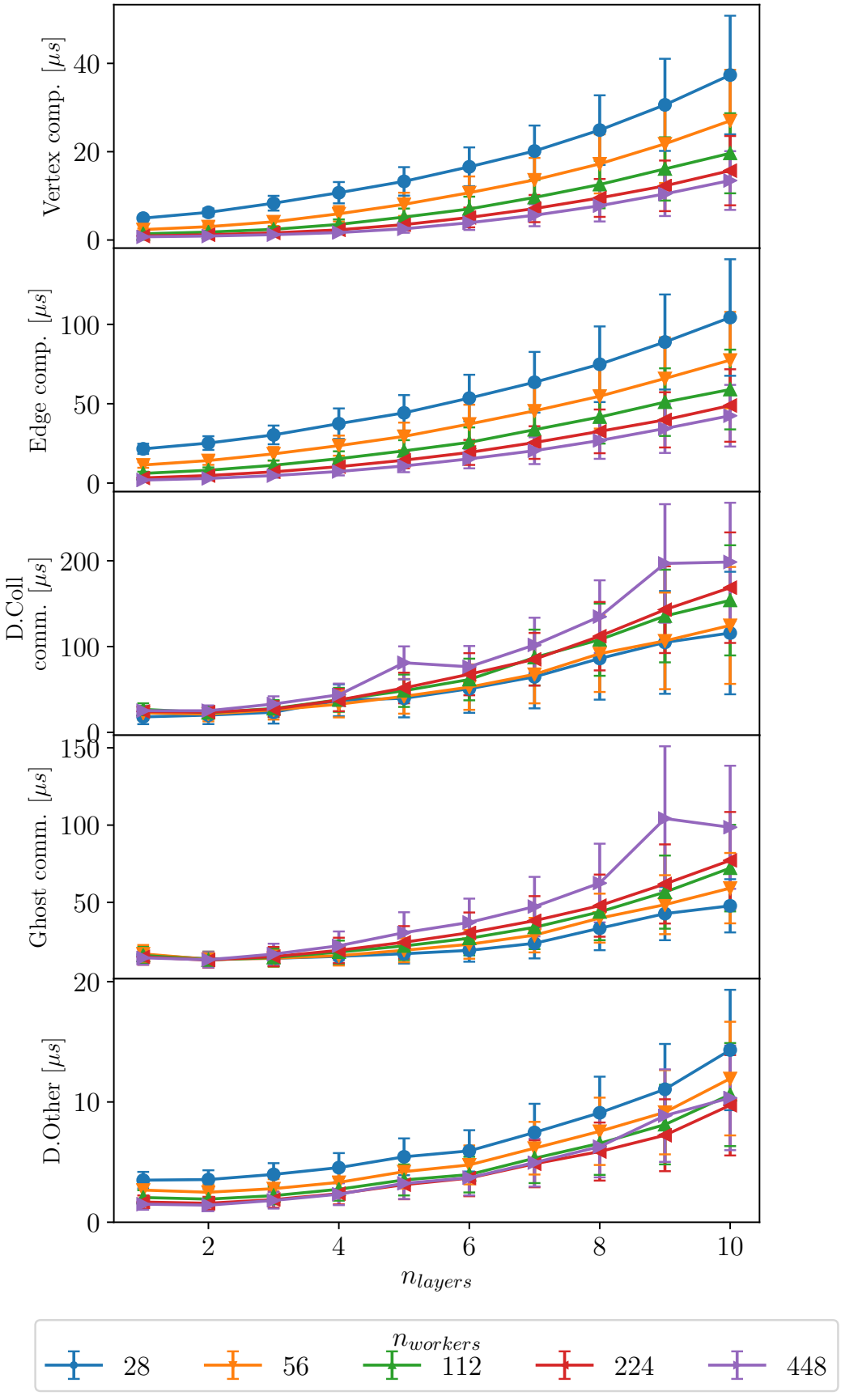


Figure 3.17: Average deflation iteration breakdown with multiple ghost layers for Preccinsta 14M on MYRIA

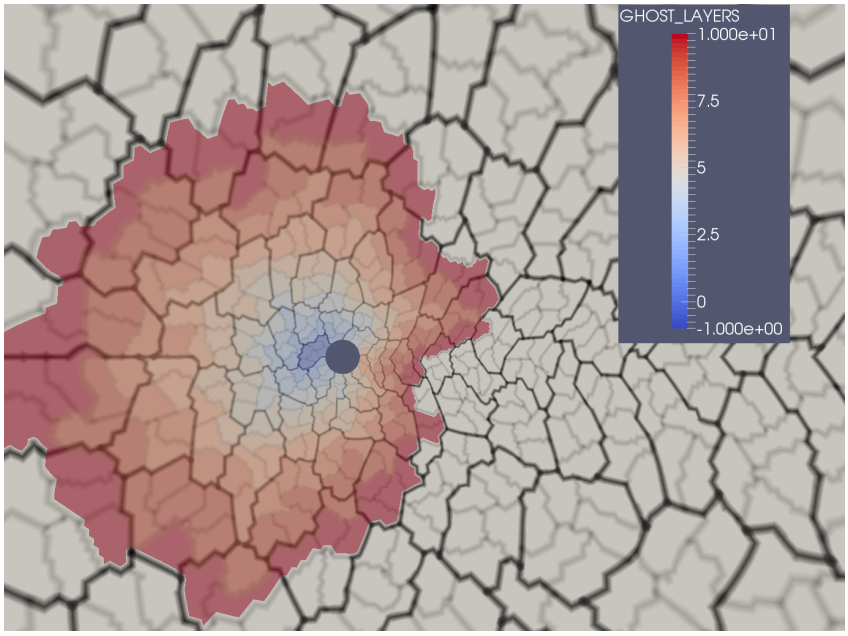


Figure 3.18: 2D_cylinder mesh vertices coloured according to their ghost layer with respect to a process. Interfaces between processes are marked by black lines while vertices are separated by grey ones

with each additional layer is larger with respect to the standard graph. This means also that the exchanged messages will be larger but previous analysis have shown that the cost of the P2P communication depends on the number of messages rather than their size.
Figure 3.19 compares the performances of the graph with different levels of gathering and ghost layers. The three meshes behave similarly to what was seen for the gathered graph in Figure 3.13. For the 14M mesh, there is no clear behaviour: For the 110M mesh, adding more layers is counter-productive, as performance degrades equally for all considered cases. Finally, a slight improvement can be seen in the 878M mesh for the case with $\Gamma = 4$ and $n_{layers} = 2$, with respect to the same level of gathering and one layer of ghosts. $\Gamma = 4$ was seen to provide the best performance for this mesh and adding one layer of ghosts allows to improve even further, especially for the lower values of $n_{workers}$. In all cases considered, $n_{layers} = 3$ gives worse performance than the lower values, indicating that, at least for this particular benchmark, adding more layers does not work even with the gathered graph.

3.4 Conclusion

This chapter presented a new data structure implemented to improve the performance of the deflation algorithm. This was done following the arguments exposed in Chapter 2, which showed how the performance of the DPCG algorithm were limited

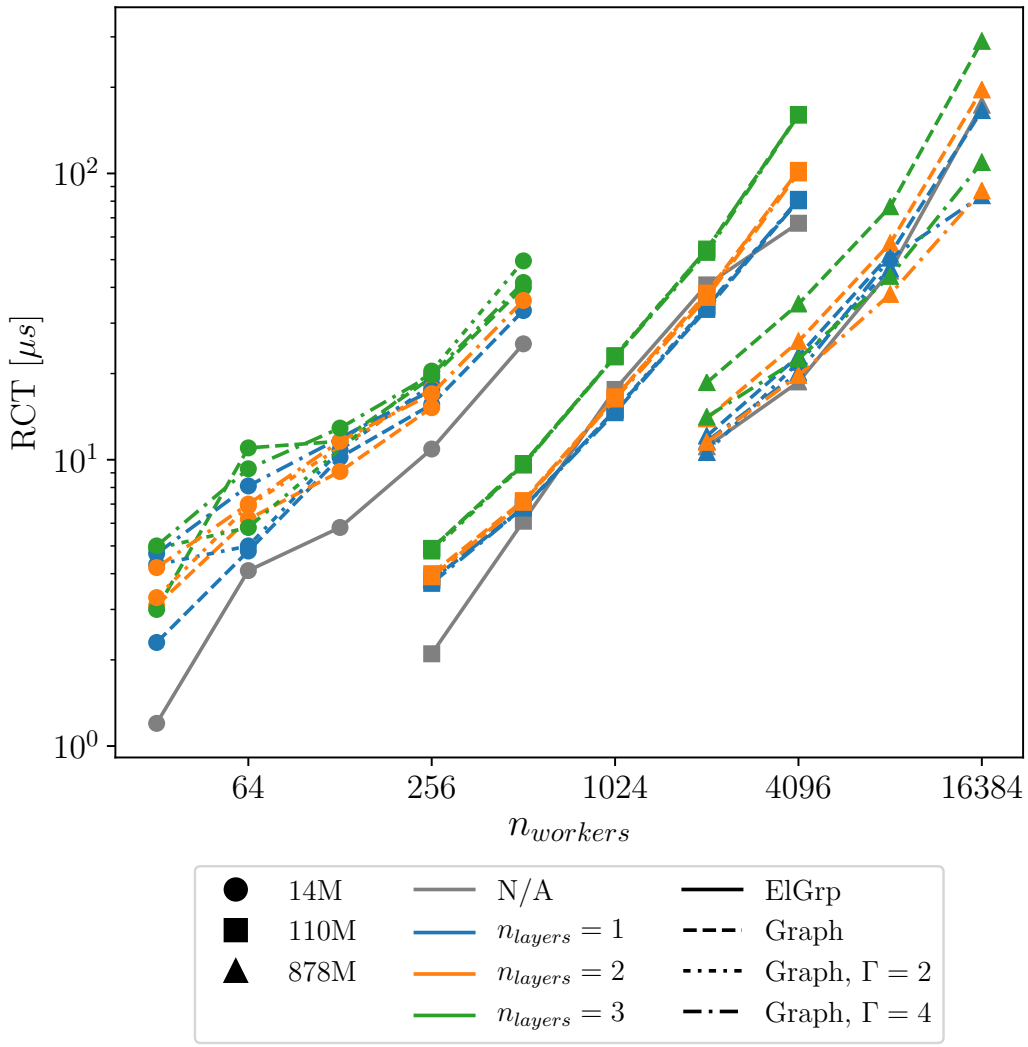


Figure 3.19: Performance comparison of gathered graph with different amounts of ghost layers on IRENE-AMD

by the coupling of the groups of elements with the deflation grid. The graph data structure presented here provides a different support for the deflation, allowing such separation. The measurements exposed in 3.2 showed how the additional degree of freedom allowed a clear improvement in the performance of the DPCG solver. In particular, it was shown that the double domain decomposition does not bring any actual advantage in the speedup of the code. L1 blocking, i.e. using a group size that would fit into L1 cache, could bring a 40% improvement for some key loops in the code[94], however it was shown that there is a much greater penalisation resulting from the massive node and pair duplication implied by such small groups. In addition, smaller ElGrp also means a larger internal communicator, and consequently an higher overhead for its update. For the global performance, with the current data structure of YALES2, minimising the data duplication is actually much more

relevant than improving memory access patterns. Other approaches should be investigated to take advantage of cache blocking without incurring in the duplicated data penalisation [95].
Another hotspot of the deflation algorithm identified in Chapter 2 was the sub-optimal communication pattern due to the half-halo ghost system, which caused an important imbalance among the different processes and consequently a delay on the collective communication. The new data structure tried to solve the problem with a symmetrical full-halo ghost communication, however it was shown how this could not avoid the imbalance caused by the difference in amount of P2P exchanges. In order to improve on this point, a system where multiple layers of ghost cells are exchanged has been put in place. It was shown how this idea, which partially worked on a synthetic 1D benchmark, was not effective on real cases as the number of neighbour quickly increases with more ghost layers. Another technique, consisting in gathering the deflation graph on a subset of workers in order to reduce the number of communicating processes was also put in place without much success, with the exception of few cases. The main problem of this method is that the additional work required by such subset of processes is not matched by the insufficient improvement in communication time.
A modified version of Algorithm 6 that uses additional operations and non-blocking collective communications [96] to try to overlap communication and computation is also implemented in YALES2. Although not presented here, some comparisons tests were performed between the two algorithms with the ElGrp based deflation, but no improvement was obtained by this non-blocking version with respect to the standard one. Although a more precise analysis is needed, the preliminary tests showed that the failure in providing better performances was due to the fact that the delay caused by the ghost exchange is larger than the computational time that the algorithm tries to cover. Furthermore, as the processes that have more communications to perform also have more ElGrp pairs to compute on, there is no way for them to catch up in their delay, consequently there will always exist a synchronisation point, even if the processes are somehow skewed along algorithm. It would however be interesting to test such algorithm with the new data structure. In addition, other possible optimisations to the DPCG algorithm have been proposed [97, 98, 99], which could also be added into the YALES2 library. The new graph data structure could provide the necessary flexibility to take advantage of such implementations.

Chapter 4

Fine grain OpenMP

The two complementary approaches detailed in Chapter 3, multiple ghosts and graph gathering, implemented to try and deal with the cost of the parallel exchanges in the deflation iteration failed due to the additional computational cost surpassing the benefit of a reduced amount of communication. In particular, gathering the graph on a restricted subset of workers means that those have more work to perform while the others are idle and not contributing to the computation, which is a waste of resources, in addition to the cost of the actual gathering and scattering of the data. The work exposed in this chapter tries to deal with this and other issues introducing a second level of parallelisation in the code: OpenMP threads.

Examples of scientific codes experimenting with the hybrid MPI+OpenMP programming model are ever more common and can be found in a variety of applications [100, 101, 68, 102], including CFD [66, 103, 104, 69, 105, 106, 107]. This is justified by the fact that pure MPI codes present some difficulty in scaling to hundreds of thousands of processes and have a high memory footprint [108]. Adding the extra layer of OpenMP threads allows to reduce the number of MPI ranks, improving the efficiency of the communication, while dividing the work among the different threads. Threads are more lightweight than MPI processes, hence they have a lower impact on the system. Furthermore, this approach is inherently hardware aware, as threads belonging to the same process must be placed on the same NUMA region, sharing some level of cache memory (usually L3). Adding an OpenMP layer to an MPI program could then help improving the overall performance, especially on thousands of cores [109] and many-cores processors with large shared memory cache memory [110].

However it is not trivial to exploit such hybrid programming model and obtain better performances than the pure MPI code [111], especially when the code structure does not allow for an additional level of work-sharing or algorithms need adaptation [112, 44]. In most of the examples cited above OpenMP parallelism is exploited by assigning a portion of the mesh of each process to a thread for computation or by loop parallelisation. In YALES2 both approaches can be merged as its data structure makes the code particularly adapted to be parallelised with loop level OpenMP, as all algorithms consist in loops on the completely independent groups of elements (ElGrps), which represent a mesh decomposition. This approach, also called fine grain OpenMP, has the advantage of being easy to implement and the code can be parallelised progressively, as parallelisation is obtained adding simple pragmas around the code loops. Before the beginning of this work, most of the loops of the code were already wrapped by OpenMP pragmas. However, for some parts of the

code, multithreading parallelisation is not that immediate. This can be caused by the fact that some regions do not contain loops (I/O) or because the iterations of the loops are not independent. The main requirement for a loop to be parallelised with multiple threads is that the work of each thread (i.e. the loop iterations) must be independent, otherwise data race conditions can occur. This is mainly the case for the internal communicator update and the deflation algorithm, whose operations are not performed on independent sets of data. The challenges and techniques used to try to parallelise these important parts of the code are detailed in Section 4.1 and Section 4.2 respectively. In Section 4.3 the performances of the hybrid implementation are compared with those of the full MPI one, while a brief summary of the chapter is given in Section 4.4. From this chapter onwards the definition of *workers* employed before also changes to take into account both the MPI ranks and the OpenMP threads and ease the comparison between the different implementations. The number of threads spawned by each rank is indicated by $n_{threads}$. Consequently $n_{workers}$ is equal to n_{ranks} (the number of MPI ranks) for the pure MPI version, while for the hybrid implementation it is $n_{workers} = n_{ranks} \times n_{threads}$. n_{ranks} and $n_{threads}$ will be used where there is the need to specify the exact nature of the parameter, otherwise $n_{workers}$ is used indistinctly for the two versions when referring to the total number of either ranks or ranks and thread.

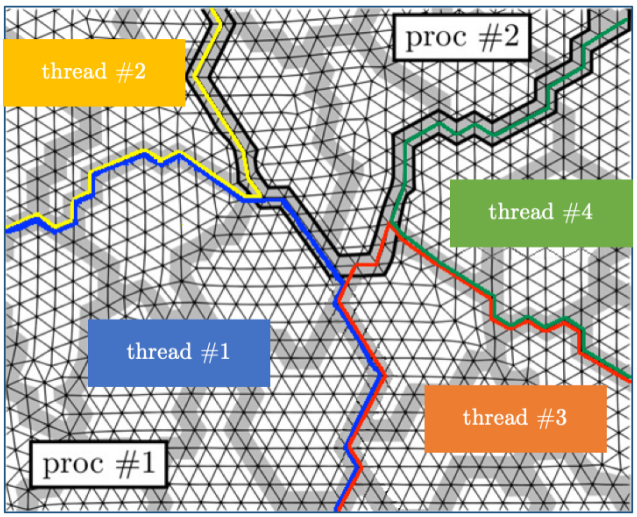


Figure 4.1: Work distribution with OpenMP fine grain model: each thread is in charge of a subset of ElGrps during computation.

4.1 Parallelisation of the internal communicator update

The multithreading of the rest of the code is based on the fact that loops on ElGrps operate on completely independent data. As explained in Section 2.1, the independence between the groups is obtained via the duplication of the shared nodes, pairs and faces. In the internal communicator however, each of these duplicated entities

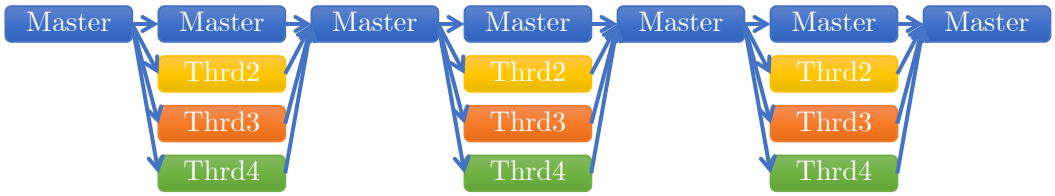


Figure 4.2: OpenMP fork-join mechanism

is represented by only one cell of the array. Consequently, the update of the IC implies data concurrency. Two different threads, updating the IC for two groups sharing a node (or a pair, or a face), will have to read, perform an operation and write on the same memory location. There is no guarantee however on the order in which these operations are performed by the different threads since they are indeed concurrent. As a consequence the result of such parallel operation is undetermined. Algorithm 10 illustrates the multithreaded IC update. The framed line is the one with the concurrent operation leading to wrong results.

Algorithm 10: Undetermined multithreaded internal communicator update

```
OMP PARALLEL DO
foreach ElGrp do
    foreach interface node do
        index = ElGrp[interface node] to IC[node];
        IC[index] = IC[index] ⊕ ElGrp[interface node];
    end
end
```

Several techniques have been put in place to try and efficiently deal with the parallel update of the internal communicator, each of which is detailed in the following subsection.

4.1.1 OpenMP locks

OpenMP foresees the use of *atomic* operations via the `OMP ATOMIC` pragma. The correctness of the result of an atomic operation is guaranteed by the fact that it is performed in such a way that the memory location is not exposed to multiple concurrent reading and writing accesses. The atomic pragma can be applied only to certain specific operations on scalars, consequently it is not adapted to perform updates on data which is of vector or tensor type as these would require each element of these 1D and 2D arrays to be treated by an atomic operation, uselessly increasing the execution time. OpenMP also allows for *critical* regions, a generalisation of the atomic operation which can be applied to any region of the code using the `OMP CRITICAL` pragma. Although critical regions can be named in order to allow the concurrent execution of multiple independent critical regions, these can

not be assigned to a specific memory location. Using a critical region would then be equivalent to perform the update of the IC sequentially.
The extremely poor performance obtained by preliminary tests performed using the `OMP ATOMIC` and `OMP CRITICAL` pragmas indicated that it was not worth using them to parallelise the IC update.
Finally, concurrent regions can be protected by a locking mechanism. A concurrent code region is protected by a lock, which can be acquired (set) by only one thread at a time, and all other threads need to wait for the lock to be released (unset) before being able to acquire it and execute the "locked" region. OpenMP locks can be applied and controlled in a much finer way than critical regions and seem to be particularly adapted for this kind of operations. OpenMP locks are acquired and released calling specific OpenMP functions.
An auxiliary array, with the same size as the internal communicator has to be created. Each element of this array corresponds to an OpenMP lock for the corresponding element in the IC. To perform the update, a thread has to acquire the specific lock, update the IC element and release the lock, as in Algorithm 11.

Algorithm 11: Multithreaded IC update with locks

```
OMP PARALLEL DO
foreach ElGrp do
    foreach interface node do
        index = ElGrp[interface node] to IC[node];
        set_lock(lock_array[index]);
        IC[index] = IC[index] ⊕ ElGrp[interface node];
        unset_lock(lock_array[index]);
    end
end
```

4.1.2 Colouring

Although locking is supposed to force sequential execution only on those elements of the internal communicator on which the execution between multiple thread is actually simultaneous at runtime, acquiring and releasing the lock for each element of the internal communicator causes a quite important overhead. In order to avoid data races, the nodes can be "coloured" in such a way that all operations on the same colour are not concurrent [104, 67]. The number of colours $n_{colours}$ necessary for the code to be free of data races is equal to the highest number of duplications for a node in the domain. In the case of YALES2, this colouring can be done at node level or at the groups level. Both options have some advantages and negative aspects. Colouring the nodes means that threads have to go through each group $n_{colours}$ times (at most) to update all nodes, as in Algorithm 12 and as schematised in Figure 4.3 for a simple 2D cartesian mesh. Colouring the groups means that the entire group has to be put on a different colour if one of its nodes can not belong to the current one. The procedure is detailed in Algorithm 13 and schematised in

Figure 4.4 for a 2D structured grid. While most nodes are duplicated only once or twice ($n_{colors} = 2$ or 3), especially in unstructured 3D meshes, some nodes can cause $n_{colours} \approx 8$. However, above colour 3 or 4, there are only few nodes per colour and measurement have shown that it is better to treat nodes belonging to those colours sequentially rather than in a parallel region, as there is not enough work to match the cost of the scheduler and thread synchronisation. This actually makes colouring the nodes more performant than colouring the groups. In those colours treated sequentially, only few nodes have to be updated in the first case, conversely, in the second case the entire interface of the group needs to be put in the internal communicator. Keeping the multithreading for more colours does not help, as often there are only one or two groups for each colour, hence it's not possible to obtain more parallelisation. The best performing version is then the one in Algorithm 12, where the multithreading is removed after the third colour.

Algorithm 12: Multithreaded IC update with coloured nodes

```
OMP PARALLEL
foreach colour ≤ 3 do
    OMP DO
    foreach ElGrp do
        foreach interface node ∈ colour do
            index = ElGrp[interface node] to IC[node];
            IC[index] = IC[index] ⊕ ElGrp[interface node];
        end
    end
end
OMP END PARALLEL
foreach colour > 3 do
    foreach ElGrp do
        foreach interface node ∈ colour do
            index = ElGrp[interface node] to IC[node];
            IC[index] = IC[index] ⊕ ElGrp[interface node];
        end
    end
end
```

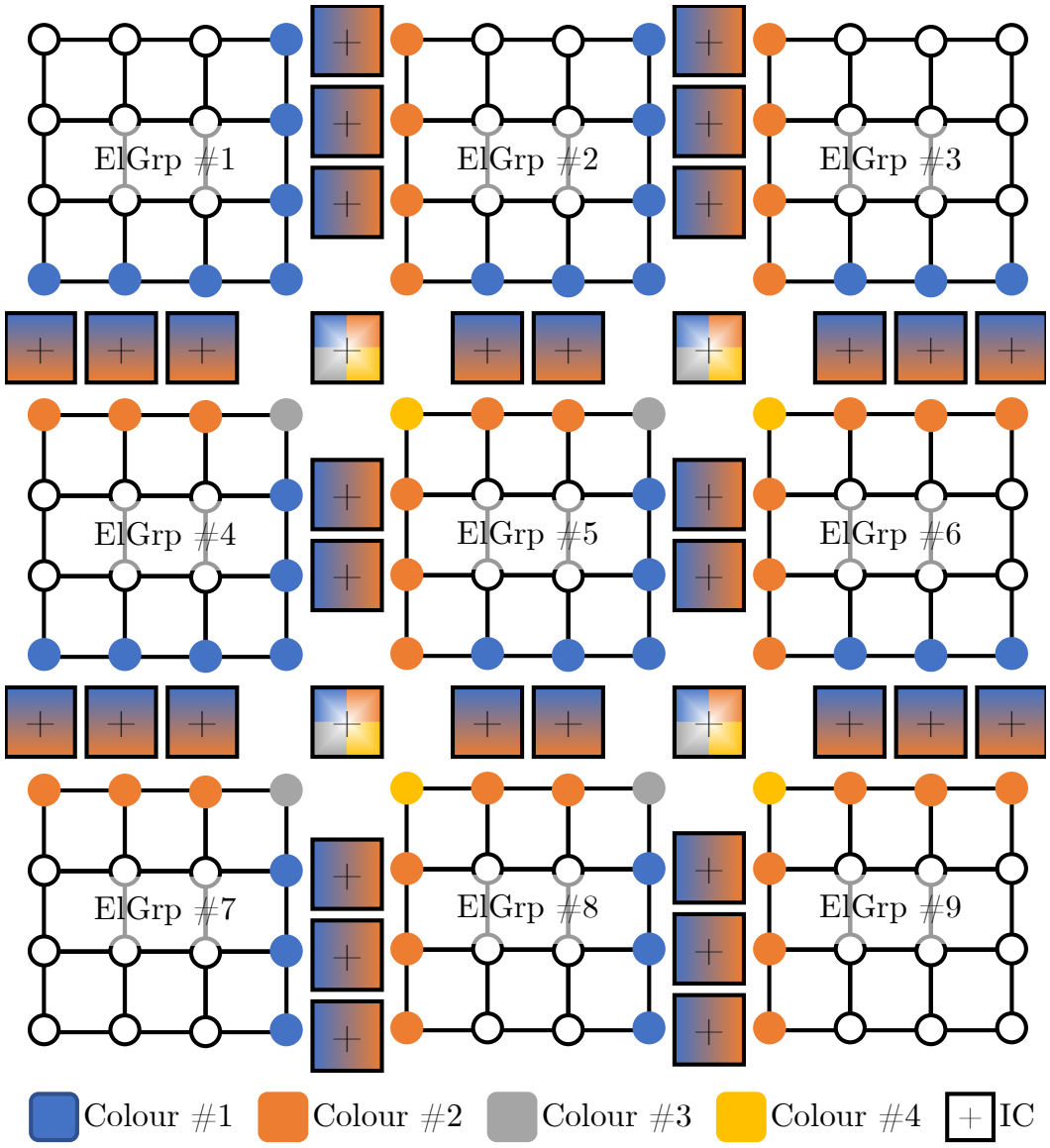


Figure 4.3: Multithreaded IC update with coloured nodes. Starting from the first ElGrp, each node on the IC is assigned the first available colour, no matter which colour have the other nodes on the same ElGrp. The IC cells are depicted to show the different colours on the nodes that share them.

Algorithm 13: Multithreaded IC update with coloured ElGrps

```
OMP PARALLEL
foreach colour do
    OMP DO
    foreach ElGrp ∈ colour do
        foreach interface node do
            index = ElGrp[interface node] to IC[node];
            IC[index] = IC[index] ⊕ ElGrp[interface node];
        end
    end
end
OMP END PARALLEL
```

4.1.3 Gathering

The concurrency in the update of the internal communicator is caused by the fact that data that is present on multiple groups needs to be written on the same memory location in the internal communicator. Colouring the different groups or nodes to avoid data races has some drawbacks, such as the need for thread synchronisation for each colour. However, all array positions in the internal communicator are independent from each other, and data can be read at the same time without concurrency on multiple groups. When the internal communicator is created, only the connectivity form the groups to the IC is created. However it is possible to create the opposite connectivity, in order to be able to get data on the groups from the internal communicator. This can be fully parallelised in one parallel region without the need for intermediate synchronisation, as in Algorithm 14. Aside from the fact that additional connectivity arrays are needed, the main drawback of this technique comes from the fact that groups are loaded into memory multiple times. The same happened also when colouring the nodes, but in that case many nodes of the groups where involved. In this case, a different group is loaded for each position, increasing the memory traffic. There might also be load balancing issues between the threads since nodes have different multiplicities and the scheduler is based on the IC size. Statistically however it is very likely that high and low multiplicity nodes are distributed among all threads, and a `STATIC` scheduling has provided better performances than `GUIDED` and `DYNAMIC` during testing.

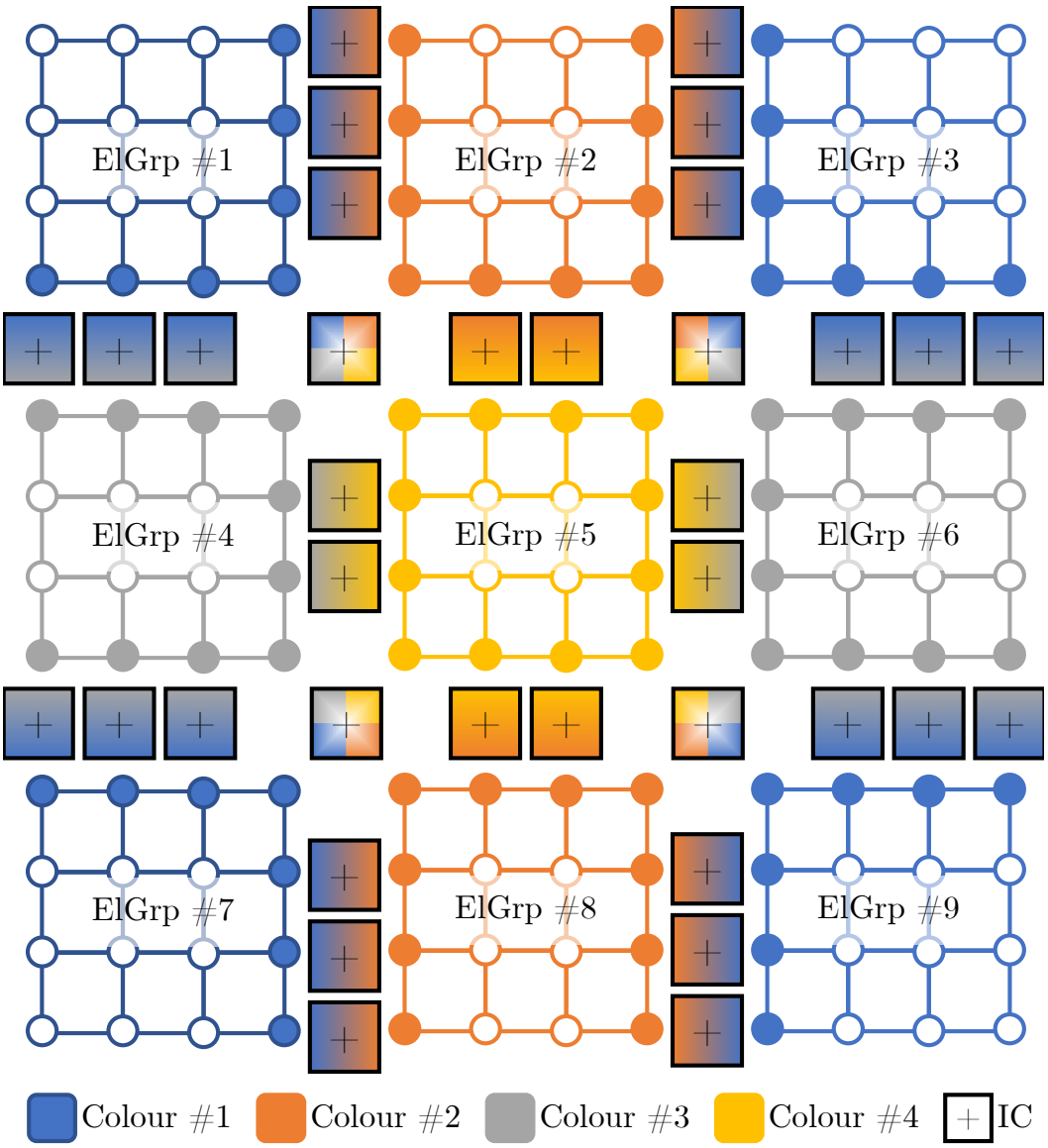


Figure 4.4: Multithreaded IC update with coloured ElGrps. All nodes on the IC of each group are assigned the first available colour. The IC cells are depicted to show the different colours on the nodes that share them.

Algorithm 14: Multithreaded IC update with data gather

```
OMP PARALLEL DO
foreach IC node do
    node_multiplicity = multiplicty_array[IC node];
    for m ← 1 to node_multiplicity do
        ElGrp → ElGrp_list[m,IC node];
        interface node = index_list[m,IC node];
        IC[node] = IC[node] ⊕ ElGrp[interface node];
    end
end
```

4.1.4 Augmented internal communicator

Another technique that was tested consists in updating the IC in two phases, with the aid of an auxiliary array called augmented internal communicator (AIC). Every duplication of a node has a dedicated element in the AIC array, which allows each group to be able to write its own value avoiding racing conditions. Every element of the IC is then able to access all the relative elements in the AIC and perform the update, much similar to the gathering technique. Since all the AIC and IC elements are independent both phases can be parallelised. This procedure is shown in Algorithm 15. Different choices can be made on how to create the AIC. The

Algorithm 15: Multithreaded IC update with augmented communicator

```
OMP PARALLEL DO
foreach ElGrp do
    foreach interface node do
        aug_index = ElGrp[interface node] to AIC[node];
        AIC[aug_index] = ElGrp[interface node];
    end
end
OMP PARALLEL DO
foreach IC_node do
    foreach node duplication do
        a
    end
    ug_index2 = IC_node[node duplication] to AIC[node];
    IC[IC_node] ⊕ AIC[aug_index2];
end
```

two that were tested during the implementation of this technique and were based on different memory access patterns for the two phases. In the first phase, i.e. the copy from the ElGrp to the AIC, it would be better if all the nodes belonging to a group were close to each other in the AIC. There are two main reasons for this. The first is that memory that would be loaded to perform the copy of a group would be

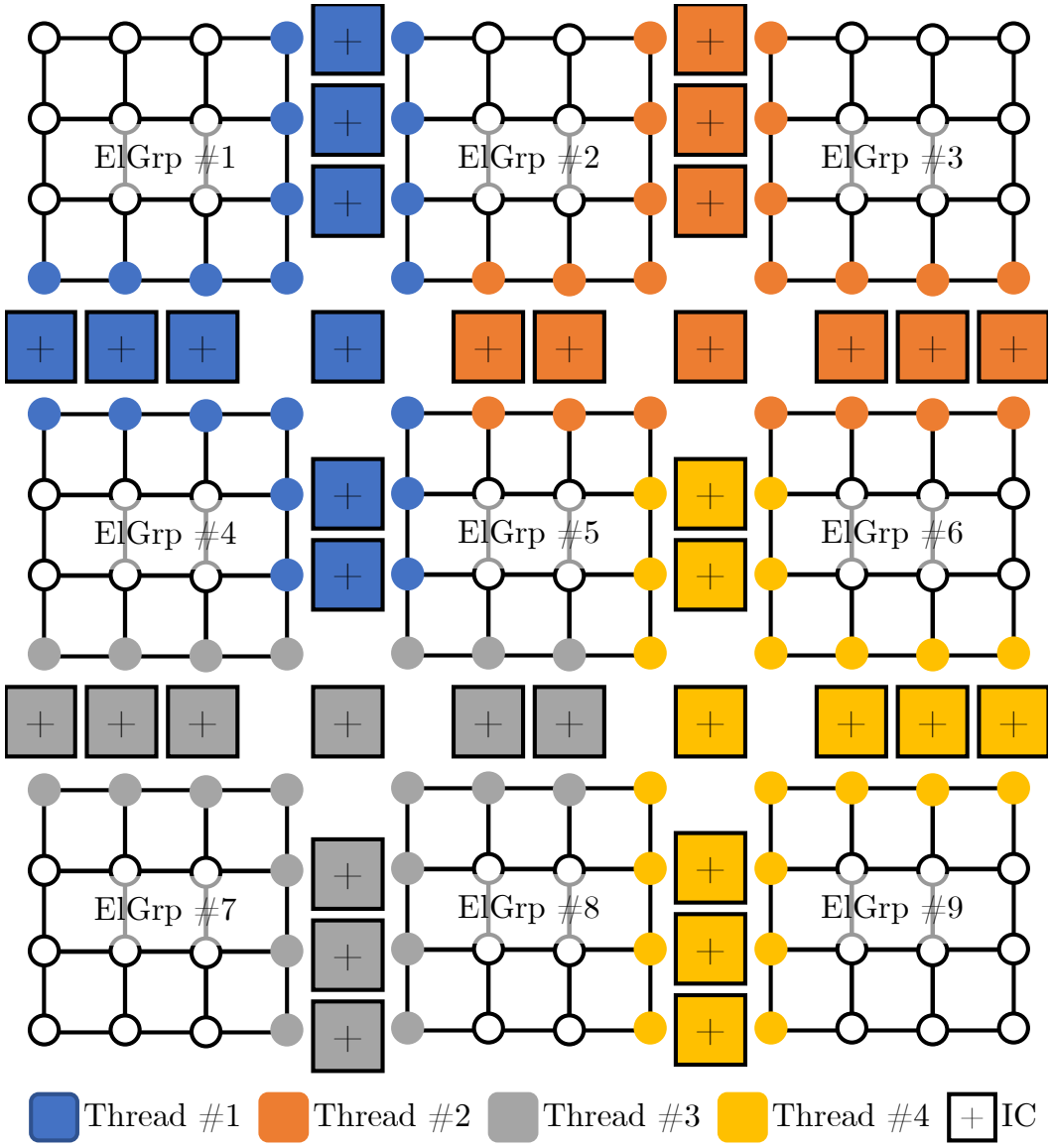


Figure 4.5: Multithreaded IC update with gathering method. Each cell in the IC is assigned to a thread which gathers the data from all ElGrps that have nodes sharing that cell. Nodes on the ElGrps are coloured according to the thread gathering their data.

contiguous, hence with faster access time. The second is that every time a thread writes on a memory page, this gets invalidated, consequently another thread that already had that page loaded in a low cache level, has to re-load it from the memory level in which threads share memory, which usually is the L3 cache. This process, called false sharing, can be particularly bad for performances if this operation is repeated multiple times [113, 114].
On the other hand, for the second phase, it would be better if all duplication of a

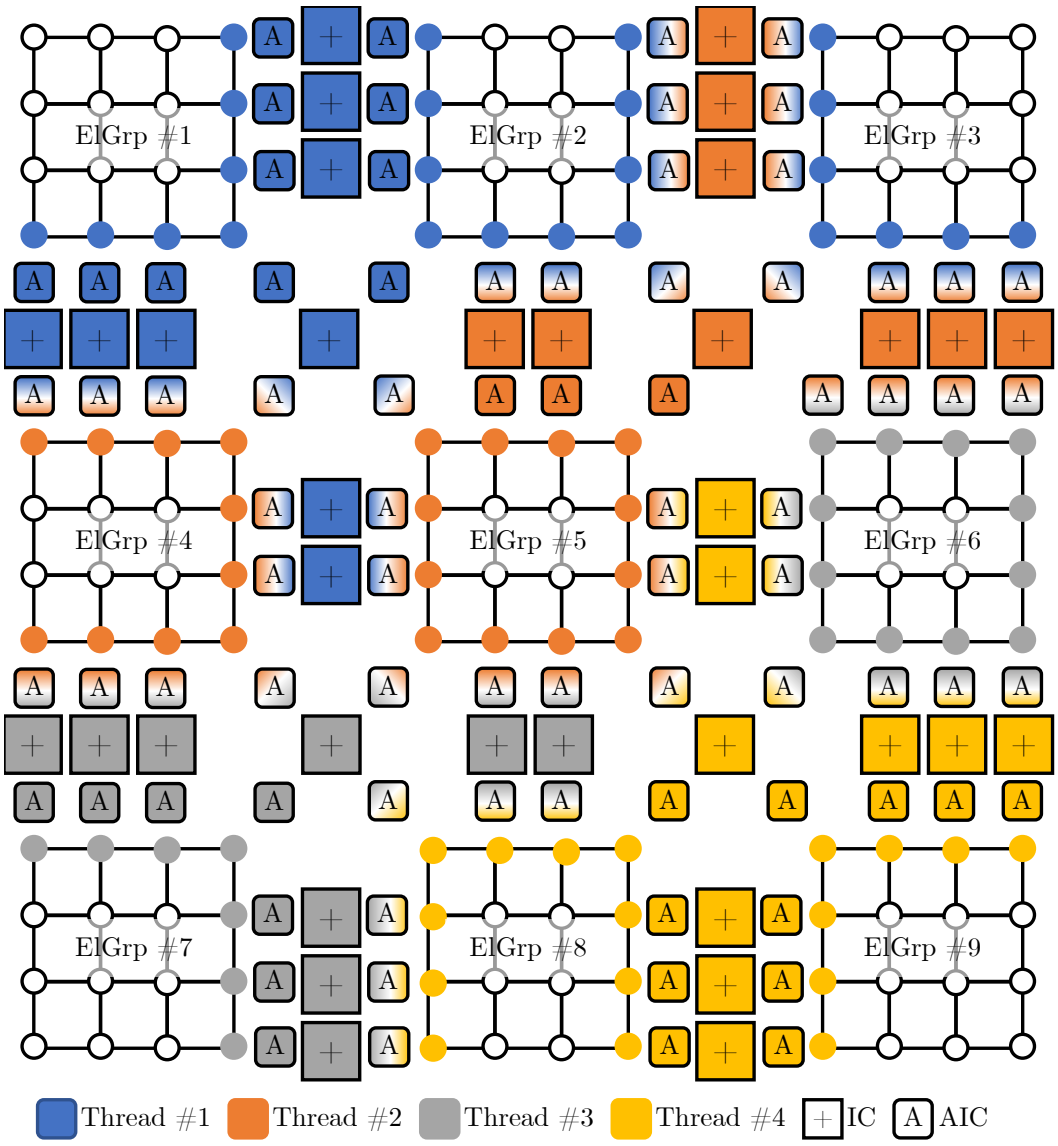


Figure 4.6: Multithreaded IC update with augmented IC (AIC). Each duplication of a node on the IC has a cell on the AIC on which a thread can copy the data. Than each cell on the IC gathers the data from the corresponding cells on the AIC. To indicate that the two operations on AIC (copy and gather) can be done by different threads, those cells have been coloured with both the copying threads (same colour as the node) and the gathering thread (same colour as the corresponding IC cell).

node would be close together, due to the fact that each element of the IC would have to access a contiguous region of the AIC. Multiple threads reading from the same memory page in the AIC however do not invalidate it during this phase, because they would modify it.
Both configurations were tested, and the configuration with the duplications stored

contiguously in the AIC proved to give better results.

4.1.5 Comparison of different techniques to update the IC with multiple threads

Figure 4.7 shows a comparison of the performance of each of the techniques explained in the previous subsections against the sequential update and the pure MPI implementation. It must be specified that the total size of the internal communicator is not the same in the case of the pure MPI and the hybrid versions. In the former, the nodes on the interface between the different processes must exist in the internal communicator of each process, whereas in the latter the interface between threads is not explicit and they share the same internal communicator, resulting in a larger global size for the pure MPI implementation. Two different benchmarks were tested each in two conditions. For the purpose of this test, it is not necessary to compute outside a socket, as the performance of this operation does not change on many nodes. In order to have representative results, the mesh of 3D_cylinder benchmark has been homogeneously refined, resulting in 3.9 million elements, while a small mesh with only 1.7 million elements has been chosen for the Preccinsta case. The test have been executed for two different values of NELEMENTPERGROUP, 2000 and another one, different for the two benchmarks, chosen in order to have only 14 ElGrps, one for each thread.
The bars designated as 14×1 represent the execution of the update on 14 MPI ranks with one thread. While the sequential bar represents the time spent in the update for the pure MPI version, comparing the different methods with the original on one thread allows to establish the additional cost of each technique. As foreseen, for all tested configurations, the cost of acquiring and releasing a lock for each element of the internal communicator is too high, making this technique highly inefficient, while the other three methods perform much better in comparison, with the gather technique slightly less performing than the colouring or the AIC.
For the multithreaded measurements, i.e. the 2×7 and the 1×14 curves, the MPI data corresponds to a sequential (non-threaded) update of the internal communicator. These bars show that using locks is counterproductive, as it produces worse performances than a sequential update in almost all shown cases. The other three techniques are of far greater interest. When the domain is decomposed in such a way that each thread computes only one group, all three give similar performance, showing important improvement with respect to the sequential update. In this case the gather and colouring methods seem to be performing quite well, however the situation drastically changes once the number of groups is increased. For these benchmarks the AIC gives performances which are far better than the rest. The mediocre performance of the gather technique can be explained by the fact that many groups are loaded in memory and then discarded several times. To explain the bad performance of the colouring technique two aspects have to be considered. First, for each colour there is a synchronisation point between all threads, hence a cause of overhead. Second, these tests were performed on unstructured meshes,

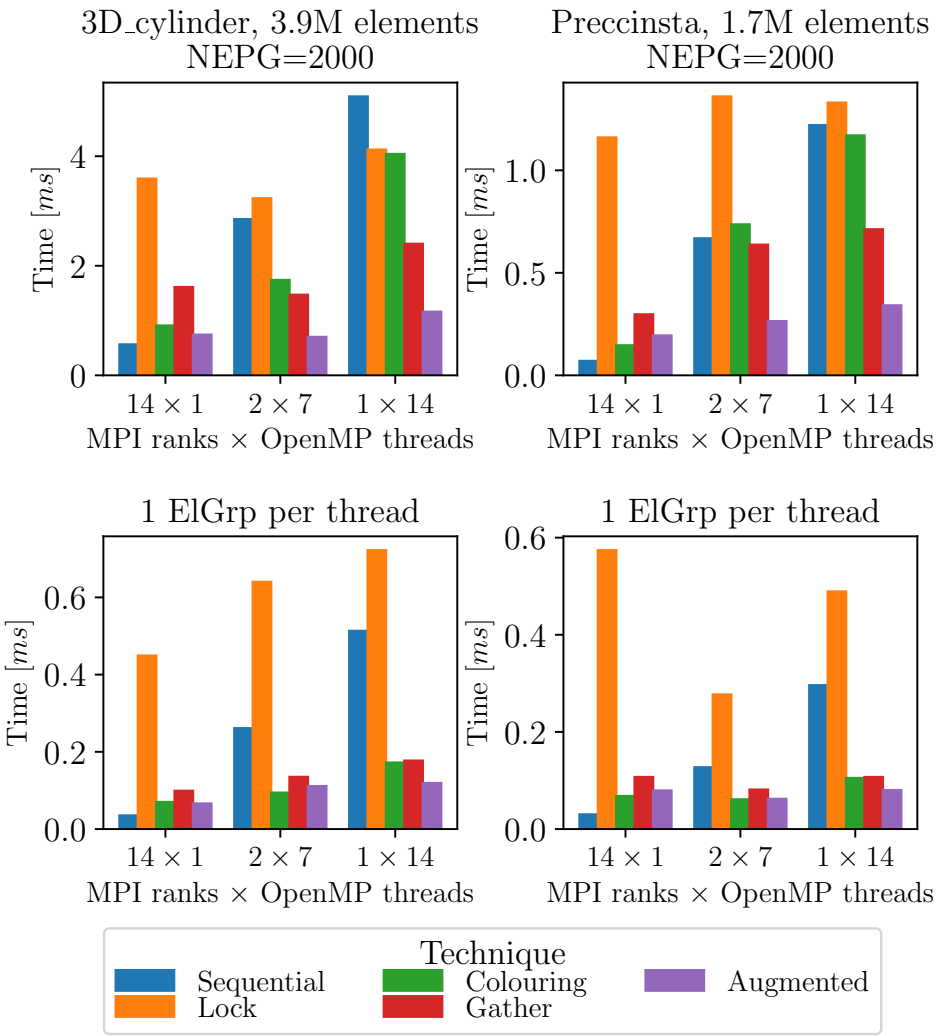


Figure 4.7: Performance comparison of the IC update with different techniques

which means that the shape of the groups of elements is highly irregular, hence nodes have very uneven multiplicity. For structured codes with cubic groups, nodes can either be shared among 2 groups (faces of the cubic groups), 4 groups (edge nodes) or 8 groups (corner nodes), with some exceptions for nodes on the boundaries of the domain. Most of the nodes belong to the faces of the groups, which means that their multiplicity is two and can be divided into two colours, while more colours can be used to treat edges and corner nodes, or these can even be treated sequentially as they are only a small fraction of the total. The irregularity of the shapes of unstructured groups, makes it difficult to pre-determine the amount of nodes with certain multiplicity. Similarly to the structured case, most of the nodes are shared between two groups, but there can be edges with nodes in common to any number of groups. Many colours can then be necessary to treat all nodes in a fully multithreaded way, but some colours might contain only very few nodes, hence they are treated sequentially as explained above.
It is important to remark that, while better than the rest, the multithreaded update with AIC remains still a few times slower than the pure MPI version in all cases.

4.2 Parallelisation of the deflation algorithm

Other than the internal communicator update, the deflation algorithm is another fundamental part of the code that can not be trivially parallelised by OpenMP pragmas. While the computation in the rest of the code is implemented with loops on independent groups of elements, the operations on the deflation are implemented as array operations. The graph data is allocated as contiguous arrays of which each element represents either a vertex or an edge, according to the type of data. Computation on this type data structure is still adapted to be parallelised by the OpenMP work-sharing constructs. The array operations can be transformed in loops on the element of the array, thus allowing to use the `OMP [PARALLEL] DO` pragmas. Different attempts have been made to try and parallelise Algorithm 6 with multiple threads, however an efficient multithreading was prevented by several aspects, described in the following.

4.2.1 Operations on the edges

The operations on the deflation Algorithm 6 are not on loop of ElGrps, but directly on arrays representing data on the groups. For those operations on the ElGrps, or vertices, which is to say all except $s = Aw$, this is not a problem as each vertex is independent and the work can be shared among threads without data races. The operation on the pairs of groups, or edges, $s = Aw$ is more complex and it can be implemented in different ways. An implementation with a loop on the edges such as the one of Algorithm 16 minimises the number of operations, however it prevents vectorisation and it can not be multithreaded as data races could occur. The write operations performed on the different elements of the s array for each edge could indeed be concurrent because different edges share a vertex. Algorithm 17

shows another implementation, which could be parallelised without data races as the array update is performed on independent locations. This implementation however requires more operations, as each edge is interrogated twice, once for each vertex, and there is still an indirection present which prevents vectorisation. In addition, while only two connectivity arrays were required in Algorithm 16, another one is needed in Algorithm 17. This is due to the fact that, while an edge can have two and only two vertices, the number of edges in which a vertex is present can vary. This additional array, together with the double interrogation of each edge, increases the memory traffic and could result in a slower computation overall.

Algorithm 16: $s = Aw$ implemented as a loop on the edges

foreach *edge* **do**
  $v1 = \text{vertex1}[edge]$;
  $v2 = \text{vertex2}[edge]$;
  $s[v1] = s[v1] + A[edge] \times (w[v2] - w[v1])$;
  $s[v2] = s[v2] - A[edge] \times (w[v2] - w[v1])$;
end

Algorithm 17: $s = Aw$ implemented as a loop on the vertices

foreach *vertex* **do**
  foreach *edge of vertex* **do**
    $v1 = vertex$;
    $v2 = \text{other vertex}[edge, vertex]$;
    $s[v1] = s[v1] + A[edge] \times (w[v2] - w[v1])$;
  end
end

4.2.2 Overhead of the parallel region

It is important to point out that the creation of an OpenMP parallel region or a `OMP DO` task scheduling comes with an overhead. If the parallel region do not contain enough work to compensate such cost, the addition of multithreading actually be detrimental to performance with respect to a sequential execution. [115, 116] provide a set of benchmarks and measurements for different OpenMP construct overheads. Similar tests were performed on MYRIA cluster, substituting a general delay function with an array operation such as $a[i] = \beta \times b[i]$ to better simulate the conditions of Algorithm 6. The results of such measurements for the `OMP PARALLEL DO` with `STATIC` scheduling are presented in Figure 4.8, for different array sizes and number of threads. The top image represents the cost of executing an increasing amount of iterations (i.e. the vectors size) with different number of threads. Below a threshold value that depends on the number of threads, the sequential execution is faster than the multithreaded one. In particular, the difference between the curve *NO OMP* and 1 thread represents the overhead of creating the parallel region, as the computation is executed on the same amount of resources. The image on the bottom

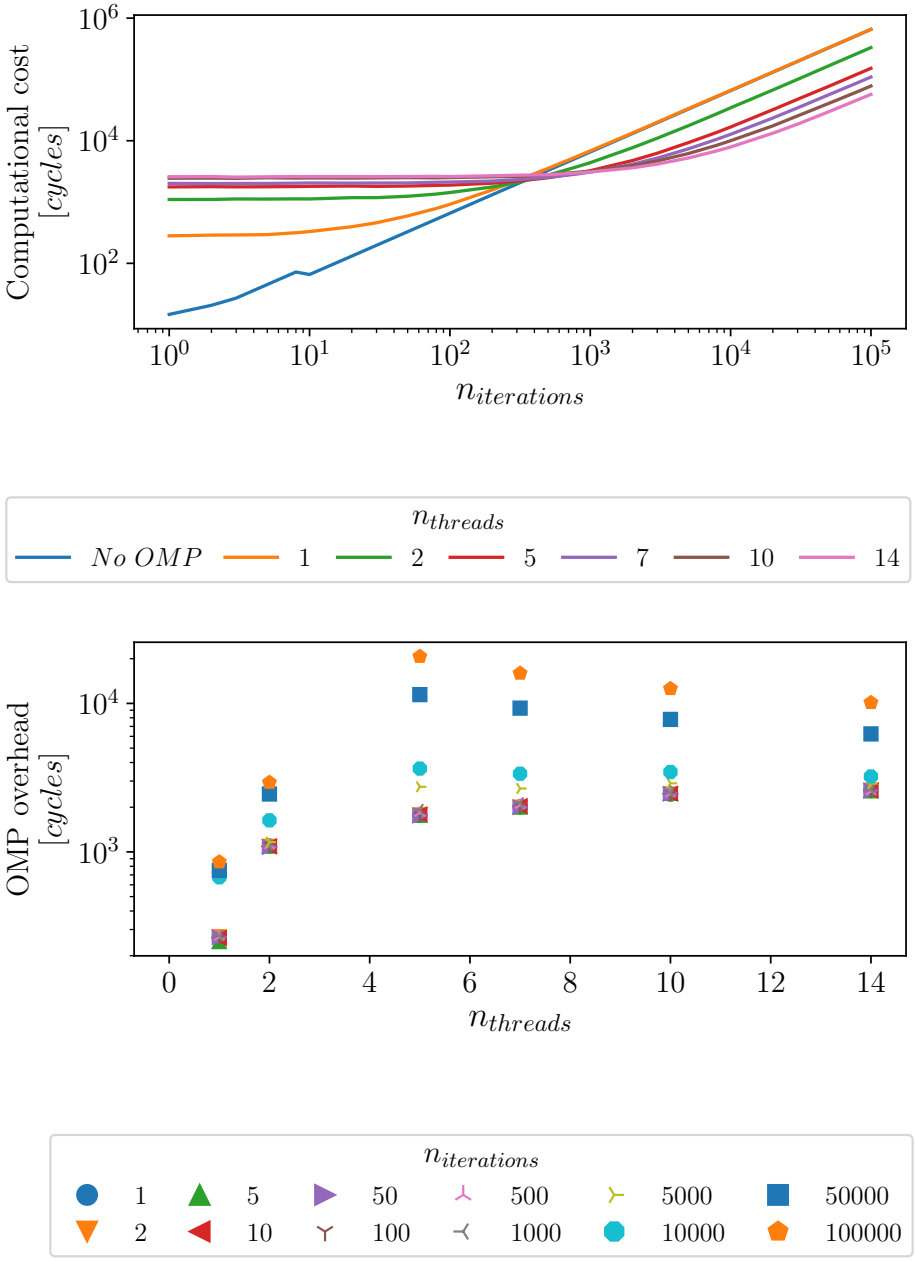


Figure 4.8: Estimation of the overhead of an OpenMP region on MYRIA. The analysed pragma is: `OMP PARALLEL DO SCHEDULE(STATIC)`

represent the overhead, measured as the difference between the real and ideal multithreaded cost. The ideal cost is computed as the $NO\ OMP$ value divided by the number of threads. Such overhead increases with $n_{threads}$ and is independent of the vector size up to a certain limit. For vectors larger than 5000 elements the overhead spikes. For the smaller arrays, all data can be fetched into the L1 cache at once, for the 5000 and 10000 elements the data still fits into L2, while the larger arrays have to be fetched from L3. For the sequential case, the computation is slow enough to allow the prefetcher to pre-load the data in the lower levels of cache, with multiple threads however data gets recycled faster and the prefetcher can not keep up. This effect is less important for the higher amount of threads as each thread computes smaller chunks of the array. Figure 4.8 shows that for operations on arrays that count less than a few hundred elements it is better to perform the operation sequentially rather than use an OpenMP parallel region. Ordinarily the incompressible solver of YALES2 performs best in the range of 100000 → 500000 elements per process. For a multithreaded case in which that amount of elements is processed by each thread, this means that with NELEMENTPERVERTEX=1000 there are about 500 vertices per thread at most, which is just above the threshold to begin to obtain some gain from an OpenMP parallelisation. In many cases however, the mesh size per process or per thread is not as big, resulting in a smaller deflation graph. Furthermore, Algorithm 6 alternates computation and communication phases, imposing several synchronisation points among threads and sections that are executed sequentially. All these factors contribute to the parallelisation overhead [117, 118].

4.2.3 Attempt to overlap communication and computation with OpenMP tasks

OpenMP work-sharing is not limited to the loops. Independent tasks can also be created and be assigned to each thread to be executed in parallel. The analysis performed in Chapter 2 showed how the performance of the deflation are affected negatively by the imbalance in the point-to-point ghost exchange. An attempt has been made to use OpenMP tasks to cover some of this imbalance overlapping the parallel exchange with some computation. All those operations that do not involve ghost vertices can indeed be performed independently from the communication. The strategy adopted is then to assign a thread to perform the communication task while the others compute. A synchronisation point has to be introduced to guarantee that the communication has finished before proceeding to the computation on the ghost vertices [119, 120]. The resulting implementation is shown in Algorithm 18 and a schematic representation is given in Figure 4.9. A group of tasks is created with `OMP TASKGROUP` because this guarantees that all tasks originated inside such group have to be terminated before leaving it. The communication task is trivial to create. The work-sharing ones are created thanks to the `OMP TASKLOOP` pragma that splits the loop in `NUM_TASKS` tasks, which in this case is equal to the number of thread minus the one assigned to the communication. Only the internal vertices, i.e. those belonging to the processor are processed in this loop. In addition, only their edges

Algorithm 18: Computation/communication overlap with OpenMP tasks

```
OMP TASKGROUP
OMP TASK
Perform ghost exchange;
OMP END TASK
OMP TASKLOOP NUM_TASKS(nthreads-1)
foreach internal vertex do
    foreach internal edge of vertex do
        v1 = vertex;
        v2 = other vertex[edge, vertex];
        s[v1] = s[v1] + A[edge] × (w[v2] − w[v1]);
    end
end
OMP END TASKLOOP
OMP END TASKGROUP
OMP DO
foreach vertex do
    foreach ghost edge of vertex do
        v1 = vertex;
        v2 = other vertex[edge, vertex];
        s[v1] = s[v1] + A[edge] × (w[v2] − w[v1]);
    end
end
OMP END DO
```

whose second vertex is also internal are computed. Once outside the `OMP TASKGROUP`, and consequently both communication and computation on internal vertices has finished, an `OMP DO` construct is used to process all vertices, but looping trough only the edges that were not computed before. Figure 4.10 gives a performance comparison of the deflation algorithm performed with three different paradigms on IRENE-AMD: the pure MPI version, already detailed in Chapter 3, the hybrid MPI+OpenMP fine grain with loop parallelisation and the hybrid MPI+OpenMP with the use of the task construct as in Algorithm 18 to try and overlap communication with some computation. It is possible to remark how the task construct does not give better performance with respect to the other hybrid version. This is due to the fact that there is not enough work to cover the communication time and also because most of the vertices have to be recomputed again anyway because they either are on the ghost layer or they share an edge with the ghost ones. The additional synchronisation point and the task scheduling also increase the overhead which in the end results in worse performance.
Concerning the comparison between the hybrid version and the pure MPI one it is possible to remark three distinct behaviours on the different meshes. For the $14M$ one, the hybrid one performs better for lower amount of workers, while the MPI one performs better for higher worker counts. This is due to the fact that the amount of work per thread diminishes increasing the number of workers and the overhead of the OpenMP construct is less important in such cases. When the work is insufficient, the overhead due to the OpenMP parallelisation is so high that it worsen the performance, as it was seen in Subsection 4.2.2. For the $110M$ mesh the MPI version performs always better. Coherently to what was seen in Subsection 3.3.1 and Subsection 3.3.2, this can be explained by the fact that for this mesh the performance is mainly driven by the ghost exchange and the imbalance in the number of edges. Parallelising the work does not give any advantage in this case, on the contrary, performance is worsened by the parallelisation overhead. Finally the behaviour of the $878M$ mesh is similar to what was seen for the gathered graph in Subsection 3.3.1. In this case, for the lower amount of workers the performance is practically identical, especially for $n_{threads} = 4$, and the hybrid versions perform slightly better for the largest worker count. This is explained by the lower amount of ranks participating in the collective communication. In this case it was added also the performance for the graph gathered with $\Gamma = 4$. This case and the one with $n_{threads} = 4$ have the same amount of ranks participating in the collective communication. However the gathered graph performs much better as it does not suffer from the OpenMP parallelisation overhead, which is especially important for the case with the highest worker count, where the work per thread is minimal.

4.3 Performance comparison

This section presents a comparison of the performances of the pure MPI version with the hybrid MPI+OpenMP Fine grain implementation presented in this chapter. For

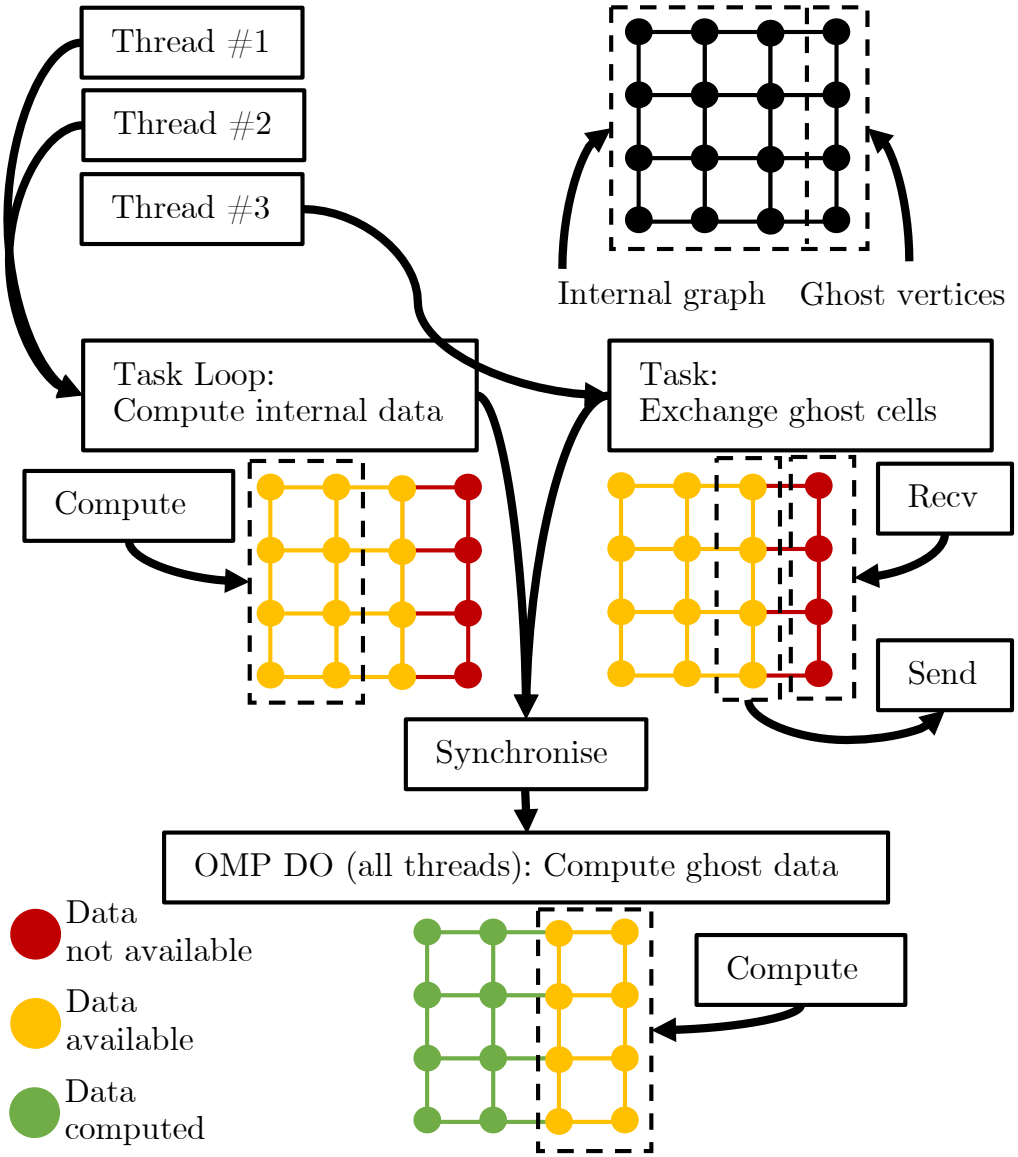


Figure 4.9: Graph computation with ghost exchange performed by `OMP TASK`. A thread is in charge of the ghost exchange while all others compute on the internal vertices and edges. One this phase is completed all threads compute together the ghost vertices and edges.

the hybrid version, NELEMENTPERGROUP was set to a value in order to obtain $n_{ElGrp} = n_{threads}$. This results in groups of the same size between the two versions with the same value of $n_{workers}$. For all infrastructures and test cases the pure MPI version performs better than the hybrid one. In order to better comprehend this behaviour, Figure 4.11 shows the breakdown of the temporal loop for the IRENE-AMD platform as an example. The other two platforms behave similarly. The deflation was analysed in Subsection 4.2.3, showing how in most cases the pure MPI version

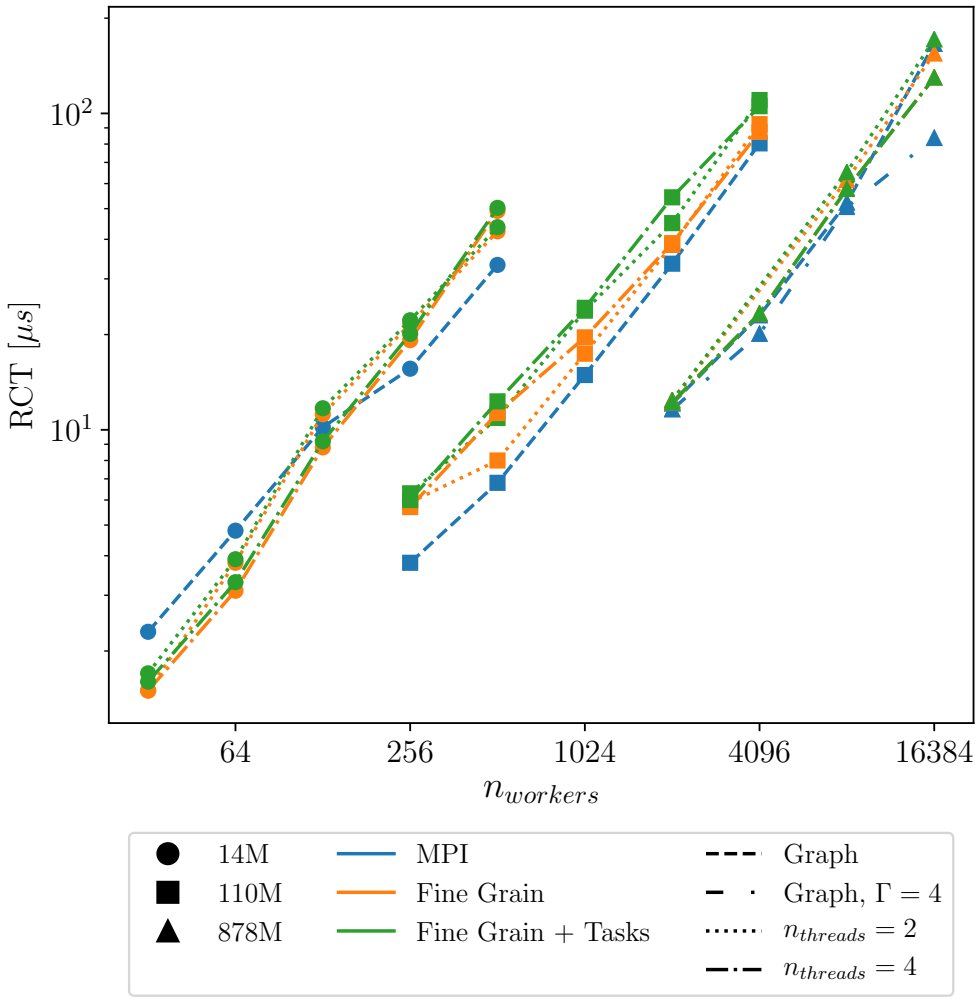


Figure 4.10: Comparison of the performance of the deflation algorithm with the graph data structure on IRENE-AMD between pure MPI, loop level OpenMP and OpenMP with the communication task.

performs better than the hybrid implementation mainly due to the overhead of the parallel regions. The time spent in the Fine Grid Allreduce is generally reduced in the hybrid version, due to the lower amount of MPI ranks participating in the communication. This is especially significant for the $878M$ mesh, were the number of ranks is higher and the cost of the collective communication is more important. However the time spent in this phase is at least one order of magnitude lower than the total time, making this gain quite irrelevant for the total performances. Other than the deflation, also the behaviour in the rest of the temporal loop penalises the overall performance of the hybrid version. Is it possible to see how, in all cases, the hybrid version is slower in this phase as well. Part of the additional cost is due to the overhead in the update of the internal communicator discussed in Section 4.1. This phase is also extremely sensible to the partitioning of the fine grid in groups of elements. While NELEMENTPERGROUP is set to obtain only one group per thread, it is possible that the partitioner can not satisfy this requirement and additional "rogue" undesired ElGrps are created. This has two main consequences: first the update of the IC gets slightly more expensive, second and most important, an imbalance in the threads work is created and carried along the entire code. The parallel regions are created distributing the work statically among the threads (`OMP PARALLEL DO SCHEDULE(STATIC)`) as this gives the lowest overhead in the ideal conditions, i.e. one ElGrp per thread. However, if an additional group is created, there will be a thread processing two groups while all the others only have one. Usually this happens because the different ElGrps have to be contiguous and the partitioner is not able to satisfy this requirement with the imposed amount of groups. For the MPI version this problem is less important as it impacts only the time spent on the IC update, but this is minimal. For the hybrid version instead, all threads have to synchronise and would wait for the late one, accumulating additional wasted time at each parallelised loop. Although a load balancing phase has been added after the first partitioning to try and avoid this problem, it is not possible to always guarantee that the number of ElGrps is equal to the number of threads, especially when a small mesh is partitioned among a lot of processes, resulting in small and often irregular domains.

4.4 Conclusions on the hybrid MPI/OpenMP fine grain implementation

This chapter presented the work done to implement an hybrid MPI+OpenMP version with loop level parallelisation. The different challenges that were encountered during this work have been exposed and some solutions were proposed and analysed. In particular, the parallel update of the internal communicator and the deflation algorithm have proved to be particularly hard to parallelise efficiently. In order to avoid data races in the IC update several techniques have been put in place and compared, however all of them entail some kind of additional operations that increases the overhead of such operation. The deflation instead suffers from the

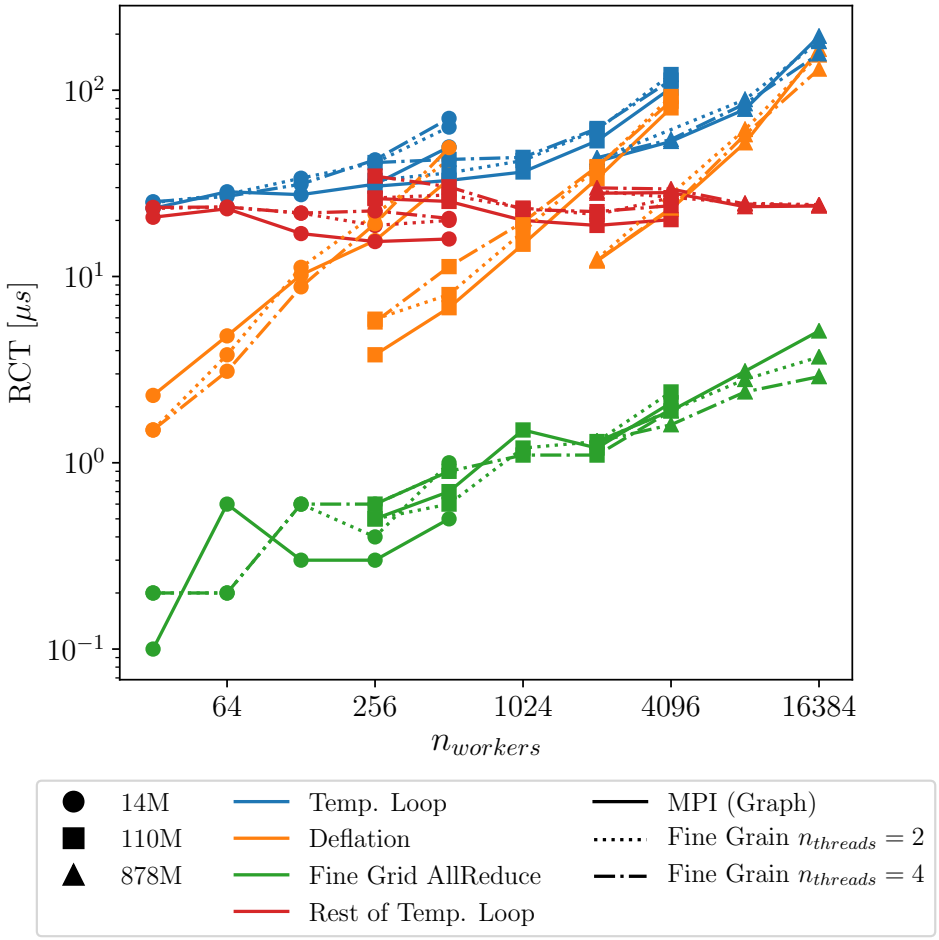


Figure 4.11: Breakdown of the performance comparison of the temporal loop of YALES2 between the pure MPI and the hybrid MPI+OpenMP fine grain versions on the IRENE-AMD platform

excessive overhead of the different parallel regions that overcomes the advantages of parallelising the work. An attempt at using tasks to overlap communication and computation has also been made, but it was unsuccessful. Finally the performance of the pure MPI version and the hybrid one have been compared, showing how the problems previously exposed prevented the latter to perform better than the former in spite of the different optimisation attempts.

This work however has also given some interesting points, especially in the reduction of the cost of the collective communication and the efficient work-sharing in the deflation algorithm shown for the $14M$ mesh and for the $878M$ mesh where the hybrid version performs as well as the pure MPI one. Indeed having less ranks communicating without losing computational capacity is a good strategy, in this case however penalised by the OpenMP overhead.

It is also important to mention that, while the temporal loop consists mainly of

loops on the ElGrps, several other parts of the code can not be parallelised as easily. Examples can be I/O operations or even the initial mesh partitioning and the creation of the data structures, which in the hybrid version are performed only by the master thread. This has the effect of massively increasing the time spent in these phases with respect to the pure MPI version as the mesh size is much larger for the hybrid version.
Attempting to parallelise also these phases and further optimise the temporal loop resulted in the hybrid MPI and coarse grain OpenMP implementation presented in Chapter 5.

Chapter 5

Coarse grain OpenMP and shared memory MPI

The work presented in Chapter 4 tried to improve the performance with the implementation of loop-level OpenMP pragmas. Such approach, also called fine-grained OpenMP, spawns lightweight threads to execute the computational work, allowing to reduce the MPI ranks which communicate and the memory footprint of the code, as data is shared among the threads and does not need to be duplicated on the different ranks. The data structure of YALES2 seemed naturally apt for this type of implementation as the work is mainly structured as loops on the groups of elements. However, the update of the internal communicator and the deflation proved to be hard to parallelise efficiently due to data races or frequent synchronisation points. Furthermore, excluding what's consumed by the MPI library itself, the reduction in data duplication is minimal, as the groups of elements duplicate data themselves, and only a small portion of the internal and external communicators are actually saved.
In this chapter the developments made to implement a hybrid MPI and coarse-grained OpenMP version of YALES2 are exposed. First, the motivations and implications of this model are presented in Section 5.1, then the implementation of collective and point-to-point communications are described respectively in Section 5.2 and Section 5.3. Finally a conclusion is drawn in Section 5.4.

5.1 The coarse grain model in YALES2

Conversely to what happens for the fine grain model, where threads are spawned and killed around work-intensive loops, in this version the OpenMP parallel region is opened once at the beginning of the execution and the threads are kept alive until the end of the program, eliminating the overhead caused by the creation of the multiple parallel regions, as schematically represented in Figure 5.1. This comes with an increased difficulty of implementation, as the threads work and memory consistency has to be managed by the program at all times. The resulting model is similar to the MPI one, SPMD (Single Program Multiple Data) where threads work on independent data, much similarly to MPI ranks [121]. Again, this model does not take full advantage of the shared memory capabilities of the program, but where the shared data actually limits the performance of the code this approach might be give better results than the fine-grained one [122, 123].
To implement this model in YALES2, a choice was made to treat the threads exactly

Figure 5.1: OpenMP coarse grain mechanism. The entire code is into one parallel region and all threads both compute and communicate.

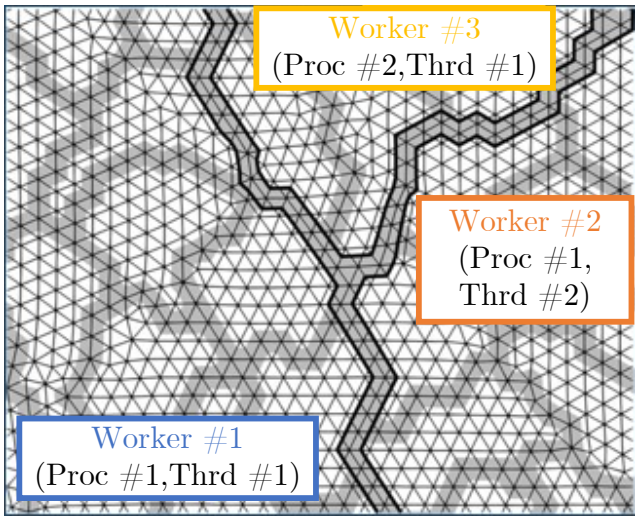


Figure 5.2: Work distribution with OpenMP coarse grain model: each worker has its own subdomain, similarly to the full MPI model.

as if they were MPI ranks, each with its own grid and the rest of the data structure and called with the generalist term *worker*. The resulting mesh decomposition, similar to the MPI one, is represented in Figure 5.2. As a consequence, little to no work had to be done in the data structure itself. Particular care however had to be put to ensure that all data would remain private to each thread and not be shared when not necessary.

5.1.1 Thread-safety in YALES2

For the multithreaded program to run correctly, the entirety of the code must be free from race conditions. This includes calls to external libraries and primitive functions. If an external library is not thread-safe, the calls to its functions must be done inside `OMP CRITICAL` regions, or other lock mechanisms to avoid threads concurrently making such calls.
In `fortran`, the language in which YALES2 is written, also primitive functions such as `open`, `close`, `read`, `write` are not implemented as thread-safe by most compilers. These functions are used extensively throughout the code, consequently it was

not practical, nor desirable, to put critical regions or locks around such functions. Another approach was followed, which consisted in creating a thread-safe wrapping function for each of these primitives. The most notable problems where found in the I/O primitives such as `open` and `close` to which a thread-safe I/O unit number had to be provided.
In addition to I/O operations, the `write` primitive is commonly used to convert data from different numeric types to string in several formats. Specific functions were written to perform such conversion due to the fact that `write` uses a non thread-safe buffer to perform the data conversions. Similar functions where also written to convert strings to numeral types to avoid the use of `read` in analogous contexts.
Finally, in `fortran`, global variables are by default shared amongst threads. Particular care was taken to ensure that such variables where declared `OMP THREADPRIVATE` when they had to be private to each thread, such as counters or pointers to data structures. However, the fact that global variables are shared can be exploited to create common data or pointers for threads to work on or exchange information. This was used extensively for the communication structures exposed in Section 5.2 and Section 5.3.

5.1.2 Communication API and shared memory windows

Another consequence of having the entire code into one `OMP PARALLEL` region is that threads need to exchange data both with the other threads on the same MPI rank and those on other ranks. In order to make the implementation as flexible as possible, and completely transparent to the user, most MPI functions used in the code have been wrapped in a dedicated API, as shown in Algorithm 19. This allows to add new paradigms with minimal modifications to the code and users can switch between them when compiling or, in some cases, even at runtime.
Implementing the coarse grain model in such a way allowed to add, at the same time, some MPI3 shared memory features as well. Shared memory windows have been introduced into the MPI standard, improving the RMA capabilities of MPI and allowing ranks to access shared regions of memory with simple load and store instructions [124]. A MPI shared memory window is a region of memory whose specific allocation allows it to be shared between ranks on the same NUMA node. The procedure to allocate the shared memory window is not straightforward. First,

Algorithm 19: Example of YALES2 communication API for `ISend` operations.

```
Y2_ISend(*args);
    if MPI only then
        MPI_ISend(*args);
    else
        // Different behaviour depending on the specific implementation
    end
```

the MPI_COMM_WORLD communicator has to be split into sub-communicators containing only the ranks in the same NUMA node. This feature is provided by the MPI_Comm_split_type function, which extends the functionalities of MPI_Comm_split taking as a input a parameter defining the communicator type. In this case, the entry parameter MPI_COMM_TYPE_SHARED produces the correct splitting. The additional entry info provides the user the possibility to ask for specific architecture informations to restrict the sub-communicator to smaller shared memory zones, such as sockets or shared cache levels, however there are no standardised info keys so these are implementation dependent, if provided at all. In order to tune the size of the shared communicator, a mechanism has been put in place where MPI_COMM_WORLD is first split into sub-communicators of the desired size via the MPI_Comm_split function. Then such sub-communicators are passed as an input to the MPI_Comm_split_type function that produces a communicator with the shared characteristics but without splitting the input one as all ranks belong to a shared memory region. This process implies that the user knows a priori the architecture characteristics and provides a correct size for the communicator in the input file, however this is exactly what is done to provide the number of threads for the OpenMP implementation, consequently it is not a real issue. The window is then allocated via a call to the MPI_Win_allocate_shared collective function on the shared memory communicator. Each rank can provide its own size for the window, and the memory can be allocated contiguously or in separated locations. The address of the regions allocated by the different ranks are retrieved via the MPI_Win_shared_query function. This mechanism is detailed in Algorithm 20. In that example a rank, designated as master, allocates the entire window in a contiguous region, while all the other ranks allocate empty ones and need to retrieve the address of the master rank window. Finally, a communicator containing only the master ranks of the shared memory sub-communicators can be obtained from MPI_COMM_WORLD with a call to MPI_Comm_split and proper inputs.

Algorithm 20: Example of the allocation of a MPI shared window.

```
Y2_Allocate_shared_window(n_elements, ..., ptr, win);
    if master rank then
        // Allocate a window that can fit all data
        win_size = n_elements;
        MPI_Win_allocate_shared(win_size, ..., ptr, win);
    else
        // Allocate an empty window
        win_size = 0;
        MPI_Win_allocate_shared(win_size, ..., ptr, win);
        // Get the address of master rank shared window
        MPI_Win_shared_query(win, master rank, ..., ptr);
    end
```

For OpenMP, shared memory regions are much simpler to allocate, as it can be done with a simple `malloc` by any thread provided that there is a mechanism to share a pointer to such region. Other than the memory allocation, the differences between OpenMP and MPI3 shared memory mechanisms are minimal. It was decided then to implement them both together in the same API. This means that the two mechanisms can not be used together and the selection between the two has to be done at compile time through specific compilation flags.
Although the data structures did not need particular modifications, considerable effort had to be put into the parallel exchanges. With the data structures completely uncoupled, the different threads need to exchange with each other data using the external communicators but also participate in the collective communications and all sorts of parallel exchanges in the program. As mentioned above, an API wrapping the calls to MPI has been introduced to facilitate the implementation of the required communications functions. The work done to implement both collective and point-to-point communications on shared memory is exposed in the following sections.

5.2 Collective communications on shared memory

Since all threads behave like MPI processes they also have to participate in the collective communications. Intel have recently introduced the `MPI_THREAD_SPLIT` programming model which allows to bind hardware resources to concurrently communicating threads and provide access to MPI objects without locks [125]. However this is only specific to IntelMPI and not a standard programming model, which means it is not available in other MPI implementations.
Collective communications have been implemented then as a multi-step communication: only the thread which is designated as *master* calls the MPI function, while the information is propagated to and from the other threads via dedicated thread-collective communications on shared memory. Figure 5.3 gives a schematic representation of an `Allreduce` communication with 2 MPI ranks and 2 threads each. First, a reduction operation is performed, in which a thread obtains the contribution of all threads on that rank. Then, the same thread calls the `MPI_Allreduce` to obtain the information from the other ranks. Finally the result is propagated through a `Bcast` from the *master* thread to the others. The same operation is detailed in Algorithm 21.
Although different algorithms [126, 38] and platform-specific optimization exist [127], `Allreduce` operations, which are the most critical for YALES2 performances, tend to scale as a function of $\log(n_{workers})$ [128]. This means that a reduction in the number of ranks by a factor equal to the number of threads would mean that the scalability of the MPI part would become a function of $\log(n_{ranks}) = \log(n_{workers}/n_{threads})$, which implies that there is a constant gain which is function of $\log(n_{threads})$. The additional cost due to the shared memory operations is a constant value which is function of $n_{threads}$. As long as this additional cost is lower than the gain obtained in

Algorithm 21: Example of YALES2 communication API for `Allreduce` sum operations with MPI and coarse grained OpenMP or shared memory MPI.

```
Y2_ALLREDUCE(*args);
    if MPI only then
        MPI_Allreduce(*args);
    else
        SHM_Reduce(master_thread, *args);
        if thread is master_thread then
            MPI_Allreduce(*args);
        end
        SHM_Bcast(master_thread, *args);
    end
```

the `MPI_Allreduce` by the reduction of the number of ranks, there is an advantage using such method. There is then the need of an efficient implementation of the collective operations on shared memory. [123] demonstrates that a global reduction on shared memory using `OMP BARRIER` to synchronise the threads performs better than the MPI one. A first version of the `SHM_Reduce` implemented with this technique is shown in Algorithm 22. The first barrier is necessary to guarantee that no thread is

Algorithm 22: Shared memory reduction for a sum operation using barriers.

```
SHM_Reduce(*args);
    OMP BARRIER
    shared_buffer[thread_id] = send_buffer;
    OMP BARRIER;
    if thread is root then
        recv_buffer = sum(shared_buffer[:]);
    end
```

still using the shared buffer. The same buffer is used for all operations (reduction, broadcast, etc.), but even if there was a dedicated one for each operation, it would be still necessary in case of two consecutive operations. Threads which are not the one performing the reduction have no other mean to know whether the root thread has finished performing the operation and consequently the buffer is free. The second barrier is needed to guarantee that all threads have written into the shared buffer and the root can proceed with the reduction operation. Another fundamental role of the barrier is to guarantee memory consistency. The `OMP BARRIER` implicitly performs a `OMP FLUSH` on all shared variables, making sure that all threads see the same values on those memory locations. Each thread has its own private copy of the shared variables and these need to be *flushed* out of the private memory to force the

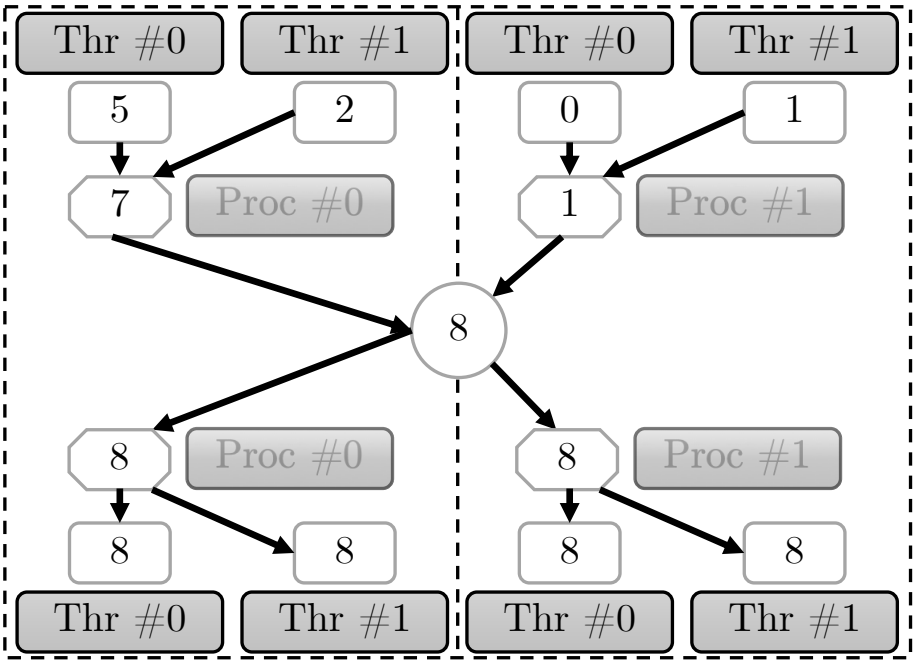


Figure 5.3: Schematic representation of a 3-phases `Allreduce` with multiple threads and processes.

thread to load it again from the shared location and see other threads modifications.

5.2.1 Optimisation of the shared memory collective communications

Although the presented implementation for shared memory collective exchanges performs quite well, the `OMP BARRIER` is a bit overzealous, in the sense that it is not necessary to have all the threads synchronised every time. The first one in particular could even be removed and substituted by a mechanism with multiple buffers, switching from one another at each operation. This mechanism has been tried out but is memory consuming and error prone, due to the fact that the switching mechanism requires atomic operations to guarantee that all threads switch to the same buffer at the same time.
Lighter barriers can be implemented [129, 130], however, as stated above, barriers are too strict and unnecessary. Mutual exclusion on shared memory is a very old problem [131, 132] and different lock mechanisms have been developed since [133, 134, 135]. Although it is not in the scope of this work to implement low level locks, a mutual exclusion mechanism is necessary for the reduction algorithm to work properly. Such mechanism should provide:

- check if the shared array is available for use;
- inform that each thread has written its data;

- inform that each thread has read the data;
- guarantee memory consistency.

This was obtained using the atomic primitives `atomic_load`, `atomic_store` and `atomic_compare_exchange_strong` [136] and 4 flags: `SHM_INACTIVE`, `SHM_ACTIVE`, `SHM_DATA_WRITTEN`, `SHM_DATA_READ`. The use of atomic operations is necessary for the memory consistency. Architectures can have different memory consistency models and most modern processors have abandoned sequential models for relaxed ones, which can provide better performances as compilers are allowed to reorder operations to maximise the throughput [137, 138]. The atomic operations cited above

Algorithm 23: Example of interface to perform atomic operations.

```
set_atomic_flag(flag_value);
    offset = cache_line_size
    atomic_store(shm_flag[thread_id × offset])
```

can however help enforcing sequential coherency where necessary. Sequential consistency means that all memory operations (`load` and `store`) are executed in the order required by the program. Although intuitively logical, this behaviour is not optimal because `load` operations are expensive, especially in case of cache misses, and these can delay the execution of other operations in another independent memory location whose data is already available. Compilers are then allowed to reorder such operations, as long as the result of the program is not altered. The expected behaviour is that `load` operations always return the latest `store` value. Within a thread, the latest `store` to a location is defined by the program order, however, when multiple threads write to a same location it is hard to define what the "latest" write is. Since to communicate between two threads could require more than 10 cycles it is impossible for a thread to know what another one has done in the previous few cycles. Access order to a memory location by multiple threads is then arbitrary and depends on which thread execute the operation first, but this is unreliable as the ordering can change at each program execution. Using atomic operations is then used to enforce some established order among the different threads. In this case, a supplementary shared array called `shm_flag` has been allocated with a number of elements equal to the number of threads. Each thread then sets or waits for a specific value on that array with the atomic functions above. To make the implementation clearer, those directives have been wrapped in a dedicated interface, like the one in Algorithm 23. A further optimisation to the mechanism has been to offset the location of the flag for each thread by the size of a cache line, usually 64 bytes. This way each thread operates on an independent line and false sharing is avoided. This is especially important for the `atomic_load` operations when performed in a loop, as it is the case while waiting for a specific flag value, because a single cache line would be contended by all threads, thus delaying progression. Algorithm 22 is rewritten using the atomic flags in Algorithm 24, while Algorithm 25 shows the

Algorithm 24: Shared memory reduction for a sum operation using atomic flags.

```
SHM_Reduce(*args);
    // wait for the buffer to be available
    wait_atomic_flag(SHM_INACTIVE);
    shared_buffer[thread_id] = send_buffer;
    // inform that data is ready
    set_atomic_flag(SHM_DATA_WRITTEN);
    if thread is root then
        // wait for data from all threads to be ready
        wait_all_atomic_flag(SHM_DATA_WRITTEN);
        // perform the reduction on the shared buffer
        recv_buffer = sum(shared_buffer[:]);
        // set the buffer as available
        set_all_atomic_flag(SHM_INACTIVE);
    end
```

Algorithm 25: Shared memory broadcast using atomic flags.

```
SHM_Bcast(*args);
    // wait for the buffer to be available
    wait_atomic_flag(SHM_INACTIVE);
    // inform that thread is using buffer
    set_atomic_flag(SHM_ACTIVE);
    if thread is root then
        // wait for all threads to be using the buffer
        wait_all_atomic_flag(SHM_ACTIVE);
        // broadcast the value on the shared buffer
        shared_buffer = send_buffer;
        // inform all threads that data is ready
        set_all_atomic_flag(SHM_DATA_WRITTEN);
    else
        // wait for data to be ready
        wait_atomic_flag(SHM_DATA_WRITTEN);
        recv_buffer = shared_buffer;
    end
    // inform that data was read, last thread sets the buffer as available
    set_check_all_and_set_all_atomic_flag(SHM_DATA_READ,SHM_INACTIVE);
```

implementation of the broadcast operation with this new method, which is slightly more complex.
The same interface has been created for MPI3, so collective communications can be performed this way also for an hybrid MPI+MPI3 code. The only difference is in the allocations of the buffers and the `shm_flag` array, which have to be allocated through the dedicated MPI functions.
For collective communications on MPI+MPI3 implementations, however, this approach has far less interest than for MPI+OpenMP, where these communications are not provided. Nevertheless, it could still be used to enforce synchronisation on operations on shared memory windows without using expensive calls to `MPI_Barrier`.

5.2.2 Performance of the collective shared memory communications

In the DPCG algorithm the collective communication which is performed more frequently is the `Allreduce` operation inside the deflation algorithm. For this reason particular effort was spent in trying to optimise this operation. Figure 5.4 shows a comparison of the performances of an Allreduce operation for pure MPI and its hybrid version. Exactly as it happens in the deflation iteration, this `Allreduce` is of type `MPI_SUM` over an array of 4 double precision values. In order to obtain meaningful values and reduce the overhead of the timing functions, several repetitions of the `Allreduce` operation were performed and an average value was obtained. These measurements were performed with both IntelMPI and OpenMPI. The OpenMPI curve was obtained with the library default configuration, while for IntelMPI its *topology aware SHM-based Knary* algorithm was used as it gives the best performances and it is SHM-aware as well.
An estimation of the OpenMP shared memory contribution can be obtained subtracting the MPI one, which can be obtained performing an `MPI_Allreduce` with the same process placement as in hybrid mode. Another way of obtaining OMP_SHM is to perform only the shared memory operations on the same amount of threads as the hybrid mode, but without the MPI operation. Both methods produce similar results and Figure 5.4 shows an average of these estimations. The number of threads being the same for all measurements, 7 in the depicted case, the value obtained for OMP_SHM should be constant. In spite of some variability due to the imprecise estimation methods, the curve obtained is fairly flat, meeting the theoretical expectations. In Figure 5.4 it is possible to see how the MPI curves clearly have a cost that can be modelled as a function $f_{MPI}\left(\log\left(n_{workers}\right)\right)$. Thanks to the approximatively constant value of the OMP_SHM contribution, the hybrid curves still have this logarithmic dependency but are shifted towards lower values. Consequently the cost for the hybrid version could be estimated with a function $f_{HYB}\left(\log\left(n_{workers}/n_{threads}\right)\right)$ +OMP_SHM.
Analysing the different curves it is possible to remark how from $n_{workers} = 28$ and $n_{workers} = 112$ onwards the hybrid mode starts to perform better than OpenMPI and IntelMPI respectively, showing the efficiency of this method.

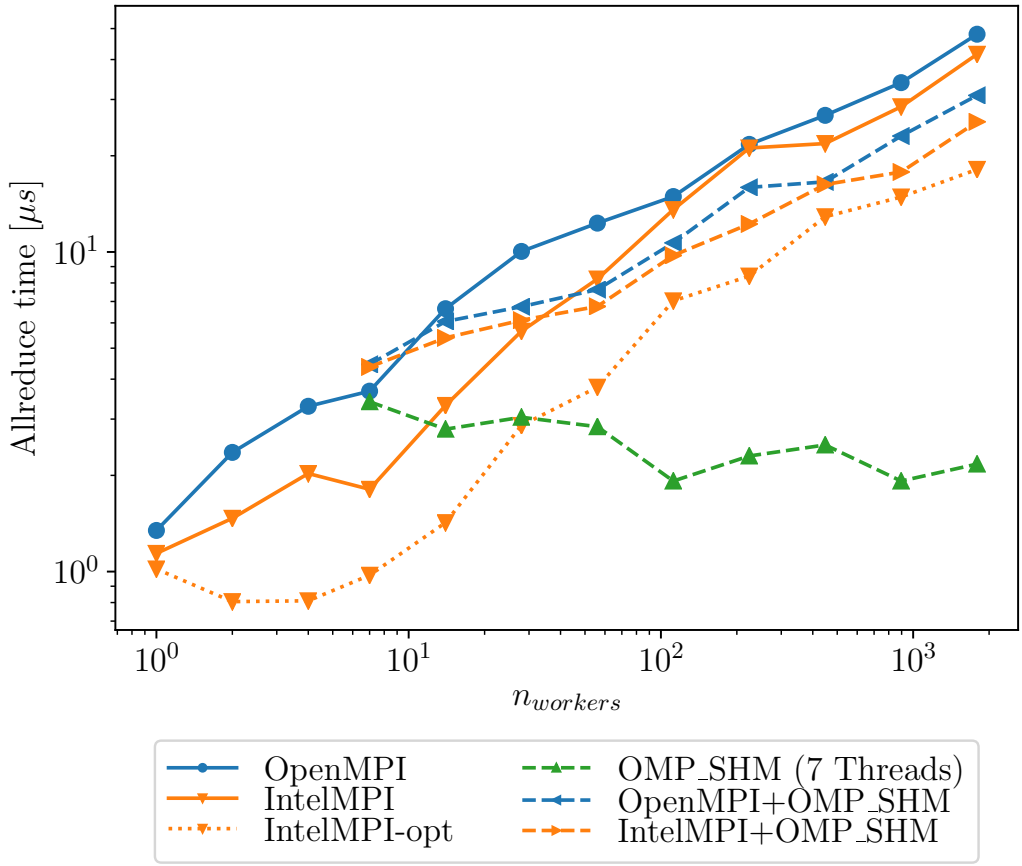


Figure 5.4: Comparison of the performances of the `Allreduce` collective communication in MPI and with the OpenMP shared memory optimisations on MYRIA. IntelMPI and OpenMPI measurements were obtained using the respective default settings. IntelMPI-opt results were obtained using its *topology aware SHM-based Knary* algorithm, while the hybrid measurements were performed with 7 OpenMP threads and MPI default settings.

However, when using its optimised topology aware algorithm, IntelMPI clearly gives better performances than all other methods. For this curve it is possible to notice how the first 4 measurements perform extremely well compared to the other methods. These measurements are all performed inside the same socket, indicating that IntelMPI itself provides an extremely efficient way to perform reduction on shared memory when its topology-aware algorithms are activated.
While it is clearly possible for pure MPI implementations to obtain better performances than the hybrid one presented here, it is important to remark how, for the successful implementation of this coarse-grain OpenMP model, it was primordial to provide a way to perform hybrid collective operations. This section has shown that not only these hybrid collective exchanges are now possible but their performances are in line with those of the most common MPI implementations.

5.3 Two-sided communications for OpenMP threads

When the entire code is multithreaded, threads must be able to perform the necessary point-to-point exchanges to update their own external communicators. All modern MPI implementation provide a `MPI_THREAD_MULTIPLE` thread support, consequently threads can enter MPI directives independently. The MPI standard however is thread oblivious and the most common implementations add locks and critical regions in the MPI directives to control the contention of shared resources among threads [139, 140]. This comes with substantial overhead compared to a pure MPI implementation where `MPI_THREAD_SINGLE` is sufficient, or even a multithreaded implementation where only one thread communicates at a given time (`MPI_THREAD_FUNNELED`). The issue of resources contention is worsened by the fact that the MPI standard does not force non-blocking communication to progress outside MPI calls. This means that in a communication scheme where a process calls `MPI_IRecv`, `MPI_ISend` and finally `MPI_Wait` the data exchange will actually be performed during the `MPI_Wait` even if there is enough time for the communication to be overlapped by the interleaving computation. This results in load imbalance, as seen in Chapter 2, but also in additional concurrency as all threads are all in the same directive trying to complete all the exchanges. To improve on such concurrency, [141] proposes to create a shared message queue where all but one threads post their message request and the last thread is exclusively dedicated to perform the MPI calls pulling from such queue. This method proved to be effective to obtain a high overlap of computation and communication time, however in YALES2 point-to-point communications are structured in such a way that there is not much possibility for such overlap. Furthermore, to have a thread dedicated exclusively to communication would mean to lose a significant amount of computational power for the most part of the code. Finally, this would not bring any advantage in the deflation algorithm as there is not enough work to overlap with the communication. Some propositions have been made in the other sense as well, such as *commtasks* [142], to have an MPI-aware OpenMP implementation which automatically takes care of advancing MPI communications through dedicated tasks. These have yet to be included in the current OpenMP norms, and consequently are not available.

5.3.1 Optimisation of the P2P shared memory communications

A first attempt at avoiding concurrency was made creating packed external communicators onto which each thread would put their own contribution for the corresponding destination worker and then only one thread would take care of calling the MPI directives to perform the exchange while the rest would wait for the communication to complete and to unpack the received data. This method was not effective as the packing presented problems analogous to those seen for the IC update in Chapter 4, it added some more synchronisation points and load imbalance.
Since thread-aware MPI implementation exist, and the MPI standard is moving

towards being more thread aware with propositions of adding so called MPI endpoints [143, 144], it was finally decided to let each thread perform its own point-to-point exchanges, with the perspective of this becoming more popular in the near future and consequently more performing.
At the moment however, standard MPI calls do not provide an immediate way to appoint a particular thread as the target of an outgoing communication, as only the receiver's MPI rank can be specified. In order to do so, the value of the `MPI_TAG` of the communication is modified in order to assume a unique value based on the sending and receiving thread numbers. This introduces some limitations, as for example it's not possible to perform communications with both `MPI_ANY_SOURCE` and `MPI_ANY_TAG` generic sender and tag values as at least one of the two must be known.
Another attempt to reduce concurrency in the MPI directive has been made creating a dedicated communicator for each couple of threads that need to communicate. This proved to be ineffective as most of the resource contention happens in the `MPI_Wait` directives, and these seem to be agnostic to the communicators as they are only based on the request number and pending messages from different communicators are appended to the same queue [145].
Finally, again with the objective of reducing the MPI resources contention a similar system of message exchange was implemented in YALES2 for the communication between threads on the same rank. As each thread has independent domains and behave for all purposes like a MPI rank in the MPI-only version, it has also its own external communicators. However, as threads on the same rank can access each other's memory, it is not necessary to send a message through the network to exchange such data, as it can be simply copied from one location to another. Thanks to the newly introduce API it is easy to chose whether to call the MPI directive in case of an exchange with a worker of another MPI rank or to go through the shared memory system in case of an exchange with a thread on the same rank. For non-blocking communication or posting send and receive requests the choice is based on the partner (sender/receiver) ID. To complete non-blocking communications however, the only information available is the request number. Each MPI implementation has its own system to provide such request number, consequently the shared memory request assignment must be aware of this and provide different values. For example some MPI implementations give ordered positive integer numbers for the requests, and the OpenMP ones must then be negative. In order to keep the interface as similar to the MPI one as possible, all directives have similar names and inputs. For instance `OMP_ISend` is the corresponding of `MPI_ISend`, `OMP_IRecv` for `MPI_IRecv`, etc. Similarly to MPI, also for OpenMP threads a system of queues has been implemented. When a thread calls `OMP_ISend`, a request object containing all the necessary informations (source, destination, tag, etc.) and the pointer to the data to be transferred. Such object is then appended to the `send_queue`. Similarly, when a call to `OMP_IRecv` is performed, a request with the same information and a pointer to the destination buffer is added to the `recv_queue`. A different request number, odd or even, is given if the request is on the send or receive queue

respectively. When a thread calls `OMP_Wait` the request number allows the thread to search the correct queue for the request with the desired number. Once this is found, the other queue is parsed to find a request with matching characteristics. At the request creation an hash was generated from the communication characteristics in order to speed up the process and avoid comparing each of the different parameters for each request. When a matching hash is found all the characteristics are checked individually as a sanity check. Once the two matching requests are found the data is copied between the sending and receiving pointers, and both requests are declared as completed. Since this treatment can be done from both sending and receiving thread independently, once a thread finds the request it's looking for it acquires a lock in order to prevent other thread from intervening on the same object. If a thread lands on a request that is locked or has been marked completed by another one it just skips it and proceeds with its workflow.
An improvement on this system could be to avoid the data copy and exchange only the pointers. However, in YALES2 the data buffers are usually part of encapsulating data structures and are allocated with different sizes depending to the needs of the code, and such sizes are stored in the containing structure. Buffers can then have a different sizes on different threads, and not exchanging back the pointers once the communication is completed could result in memory faults. Unfortunately, in the exchange scheme of Algorithm 8, which is the one employed in YALES2 at the moment, there is no safe way of exchanging back the pointer, as it is needed after the first `Waitall` and the second one has now way of telling if the data has been consumed already by the corresponding process or not. This means that the only way of performing the data exchange with the current algorithm is via a data copy.

5.3.2 Performance of the P2P shared memory communications

In this section the performances of the shared memory implementation of P2P communications are compared with those of IntelMPI-17 (IMPI17) and OpenMPI-3.1.3 (OMPI313). In order to understand the overhead due to the multithreaded mode, the pure MPI implementations have been benchmarked in both `MPI_THREAD_SINGLE` (TS) and `MPI_THREAD_MULTIPLE` (TM) modes. In both cases the compiled library supported multithreading, and the selection between the two modes was made at runtime.
In order to reproduce what happens in the core parts of YALES2, the benchmark used was an all-to-all exchange, implemented with `Y2_Irecv`, `Y2_Isend` and `Y2_Waitall` operations, as shown in Algorithm 26. Although technically all requests can be waited for together, the use of two separated `Waitall` calls mimics the YALES2 parallel data update (Algorithm 8). The benchmark was run in three different conditions:

insocket the benchmark was run inside a socket, i.e., all threads are able to communicate without recurring to the MPI library;

exsocket the benchmark was run with workers on different sockets, consequently

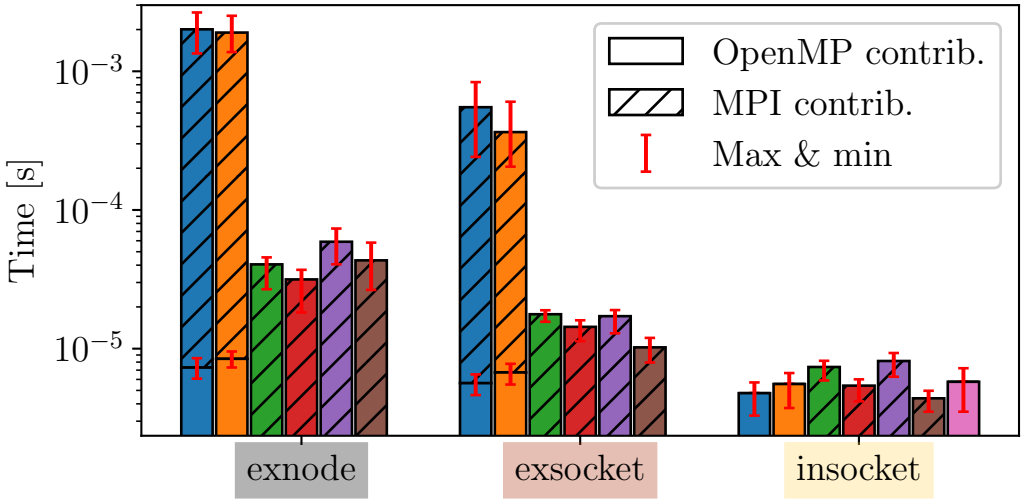
threads need to use the MPI library to communicate with threads outside their own socket;

exnode the benchmark was run with workers on two different nodes. Threads need to use MPI calls to communicate outside the socket, hence outside the node.

Algorithm 26: All-to-all benchmark algorithm.

```
// Loop of Irecv
forall other workers do
    Y2_Irecv(*args);
end
// Loop of Isend
forall other workers do
    Y2_Isend(*args);
end
// Waitall recv requests
Y2_Waitall(*args);
// Waitall send requests
Y2_Waitall(*args);
```

While for the pure MPI benchmarks at each physical core was assigned an MPI rank, for the pure OpenMP and hybrid versions the process placement was such that only one rank was present on each socket, and each core of such socket was bound to a OpenMP thread. Figure 5.5 shows the performance comparison between all benchmarks for the entire algorithm. It is possible to remark how for the *insocket* conditions the pure OpenMP and hybrid implementations (which in this case are identical) have performances in line with those of the pure MPI ones. In the *exsocket* and *exnode* conditions however, the hybrid implementations perform much worse than the pure MPI ones. It is possible to see how the OpenMP contribution remains unchanged, as the number of exchanges on shared memory is the same on all three conditions, but the MPI contribution is two orders of magnitude higher with respect to the pure MPI implementations. The number of exchanges per thread using the MPI library goes from 14 on the *exsocket* condition to 42 (14×3) for the *exnode* one. The MPI time, however, increases about 5 times between the two. Also, in both cases is much higher than the pure MPI benchmarks, which have to communicate with 6 more ranks (those on the same socket). This is due to the fact that these MPI implementations, although allowing multiple threads to call the MPI functions at the same time, they have critical regions and lock inside to guarantee thread-safety. It is obvious how the poor efficiency of such implementations finally jeopardise all other effort to gain performance using multiple threads.
Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9 give the same performance comparison for each of the for steps of the Algorithm 26. It is worth underlining how the MPI contribution for the hybrid implementation in the *exsocket* and *exnode*



	HybIMPI17	HybOMPI313	IMPI17TM	IMPI17TS	OMPI313TM	OMPI313TS	OpenMP
exnode-OpenMP	7.31E-06	8.45E-06	0.00E+00	0.00E+00	0.00E+00	0.00E+00	/
exnode-MPI	2.00E-03	1.89E-03	4.05E-05	3.16E-05	5.90E-05	4.33E-05	/
exnode-Total	2.01E-03	1.90E-03	4.05E-05	3.16E-05	5.90E-05	4.33E-05	/
exsocket-OpenMP	5.64E-06	6.74E-06	0.00E+00	0.00E+00	0.00E+00	0.00E+00	/
exsocket-MPI	5.45E-04	3.58E-04	1.77E-05	1.44E-05	1.72E-05	1.02E-05	/
exsocket-Total	5.51E-04	3.64E-04	1.77E-05	1.44E-05	1.72E-05	1.02E-05	/
insocket-OpenMP	4.78E-06	5.56E-06	0.00E+00	0.00E+00	0.00E+00	0.00E+00	5.77E-06
insocket-MPI	0.00E+00	0.00E+00	7.38E-06	5.41E-06	8.15E-06	4.38E-06	0.00E+00
insocket-Total	4.78E-06	5.56E-06	7.38E-06	5.41E-06	8.15E-06	4.38E-06	5.77E-06

Figure 5.5: Performance comparison of shared memory and pure MPI implementations on the all-to-all benchmark of Algorithm 26 on the Myria platform. It is important to remark that the vertical axis has a logarithmic scale

conditions are much worse in Figure 5.6 and Figure 5.7, which correspond respectively to the loop of `Y2_Irecv` and `Y2_Isend` calls. This is supposedly due to the fact that such functions are called in a loop, consequently different threads are continuously contending the locks, while the unique call to the `Y2_Waitall` allows a mere sequential execution of the critical regions. The `MPI_Waitall` time is however much higher for the hybrid implementation than for the pure MPI ones. It is also interesting to remark how, in the *insocket* benchmark, the shared memory implementation outperforms the MPI ones for the `Isend` and `Irecv` calls, while its much slower for the `Waitall`. This is probably due to the fact that for the shared memory implementation the first two operations only consist in generating a request object and associating a pointer, while the data exchange is performed in the `Waitall`. Furthermore, `Isend` and `Irecv` calls are almost lock-free, while `Waitall` operations are more delicate and several locks are necessary to guarantee thread-safety.

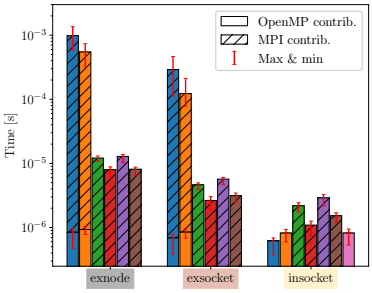


Figure 5.6: Performance comparison of shared memory and pure MPI implementations of the `Y2_Irecv` loop of Algorithm 26 on the Myria platform.

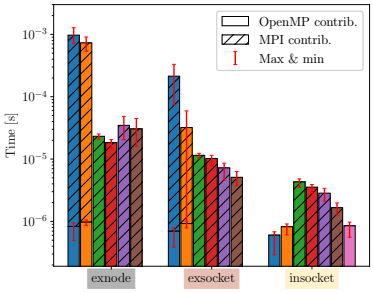


Figure 5.7: Performance comparison of shared memory and pure MPI implementations of the `Y2_Isend` loop of Algorithm 26 on the Myria platform.

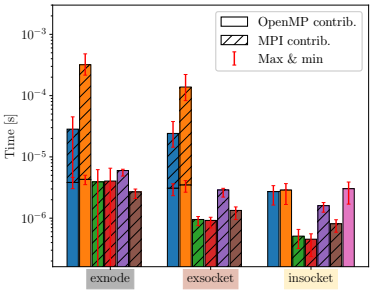


Figure 5.8: Performance comparison of shared memory and pure MPI implementations of the `Y2_Waitall` on the receive requests of Algorithm 26 on the Myria platform.

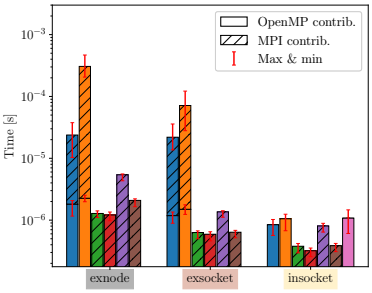


Figure 5.9: Performance comparison of shared memory and pure MPI implementations of the `Y2_Waitall` on the send requests of Algorithm 26 on the Myria platform.

5.3.3 Attempt at one-sided communications

Multithreaded two-sided communications have proven completely unusable in the coarse grain model, mainly because of MPI libraries that are internally sequentialised. An attempt was also made to use one-sided MPI communications to see if these were less impacted by critical regions. The YALES2 algorithms and data structures do not foresee the use of one-sided communications, consequently this attempt only tried to mimic the two-sided ones. The `Y2_` functions of the communication API have been extended with an optional entry used to specify the window for RMA operations in case one-sided operations are to be used. RMA operations have been experimentally implemented only in the hybrid version for threads that need to communicate with other ranks, while the pure MPI implementation still uses the standard two-sided ones.
Due to the way data exchanges are written in YALES2, the receive requests are always posted before the send ones. For this reason it was decided that in the `Y2_Irecv`, the RMA API would only set up a request, as it happens for the shared memory two-sided communications. On the other hand, the `Y2_Isend` RMA function also performs a data exchange via a RMA operation to a shared buffer, as shown in Algorithm 27. For this data exchange to work, the intermediate buffer must be allocated via MPI functions as a shared window. In order to be able to notify the receiving process of the data transfer, the notifications are exchanged with a similar mechanism on a specific shared window. The `Waitall` behave differently

Algorithm 27: Isend mimic with RMA operations.

```
// Create Isend RMA request
isend_req = new_RMA_request(*args);
// Take a lock on the shared window
MPI_Win_Lock(MPI_LOCK_EXCLUSIVE,isend_req->window);
// Copy data on shared buffer
MPI_Put(data,isend_req->window,...);
// Release the lock on the shared window
MPI_Win_Unlock(isend_req->window);
// Notify that data was written
MPI_Win_Lock(MPI_LOCK_EXCLUSIVE,isend_req->notification_window);
MPI_Put(RMA_PUT_FLAG,isend_req->notification_window,...);
MPI_Win_Unlock(isend_req->notification_window);
```

depending if they are called on a receive or send request. In the first case, once the corresponding send notification is found, the data is copied from the shared window to the receive buffer with a simple `memcpy` and the send request is reset with a call to `MPI_Put`, similarly to what is done in the `Isend` operation, but with a different flag value. In the second case instead, the calling thread only waits for its notification to be reset from the receiving thread on the other side.
It is clear that MPI one-sided communications are not intended to be used to re-

produce the behaviour of two-sided ones, however this was a quick attempt to solve the problem exposed above for multithreaded two-sided communications.
Unfortunately, even though the results are not presented here due to lack of extensive benchmarking, RMA operations seem to have the same problem of the two-sided ones and are not multithreaded.
This failed attempt has nonetheless allowed to implement in YALES2 some basic RMA structures and mechanisms which will be useful in the future to fully implement proper one-sided communications.

5.4 Conclusion

This chapter has presented the attempt at introducing a OpenMP coarse-grain model into YALES2. This was done to address the issues of the fine-grain model exposed in Chapter 4, namely the frequent thread synchronisation and the parallel region overhead.
The main implication of the coarse-grain model is that all threads must communicate, consequently shared memory collective and P2P communication mechanisms were introduced.
For the shared memory collective communications presented in Section 5.2, it was possible to obtain performances comparable with, and even better than, some MPI implementations.
For the P2P communications however, in spite of the efforts made to allow threads on the same rank to communicate through shared memory, the critical regions and locks present into the routines of the most common MPI implementations caused enormous performance pitfalls when threads have to communicate with other ranks (i.e. using the MPI routines). Such is the negative impact of the internal sequentialisation of MPI routines that the entire code runtime is impacted and it was impossible to obtain performances even comparable to the pure MPI or the fine grain model.
A brief attempt to use one-sided communications was made, but these suffer from the same pitfalls of the two-sided ones, and were not developed further.
In conclusion, the hybrid MPI+OpenMP coarse-grain model allowed to introduce several improvements to the code, such as the full thread safety, the abstracted communication interface and some basic structures for shared memory and one-sided MPI operations, however the performance pitfalls in the P2P exchanges prevent it from being effective. Better performances could possibly be obtained with fully multithreaded MPI implementations or with deeper modifications in the data structures of the code.

Chapter 6

Conclusion and perspectives

To summarise what was presented in this work, Chapter 1 gave an overview of numerical methods, parallel communication paradigms and HPC tools used in CFD. Chapter 2 presented the data structure and some performance measurements of YALES2, the CFD code used in this work, trying to give some simplistic models for its performances, underlining its strong points and exposing its issues and bottlenecks. Chapter 3 introduced a new data structure, which was implemented to try and address the problems exposed in the previous chapter. Chapter 4 described the work done for the introduction of an OpenMP fine-grain implementation in order to divide the work amongst multiple threads and reduce the number of MPI ranks communicating. Chapter 5 presented the efforts made to implement a OpenMP coarse-grain model that aimed to address the issues of the fine-grain one.
This last chapter draws some final conclusions on what was presented and tries to give some perspective for future developments and improvements on this work.

6.1 Conclusion

The performance measurements and relative models obtained in Chapter 2 showed that most of the code scales perfectly to relatively high numbers of MPI ranks. This is not true for the deflation algorithm, which scales badly, even at low rank counts. The deflation, however, is the core part of the Poisson linear solver, which is where most of the computational time is spent. It was shown that the lack of scalability is mainly due to the high cost of communication with respect to the computation in this particular part of the code. Furthermore the communication time is highly influenced by the imbalance in the computational load but mostly in the number of point-to-point exchanges.
The graph data structure introduced in Chapter 3 tried to address these issues modifying the ghost cells structure in order to improve the P2P communication pattern and balance the computational load on the number of edges. In addition, the possibility to use multiple ghost layers and to gather the graph on a subset of workers were introduced, but didn't give the expected benefits. In particular, the multiple ghost layers suffered from a rapid increase in amount of P2P communication needed and computational cost. The gathered graph also suffered from an increase of work per rank, which in most cases surpassed the benefits in communication time.
To address these issues a OpenMP fine-grain model was introduced in Chapter 4. The efficient parallelisation of the deflation algorithm proved challenging due to the need for thread synchronisation and the memory boundedness of the code. In the

end, either the pure MPI model or the gathered graph were performing better than the fine-grain implementation.
To overcome the limitation of the fine-grain model, a OpenMP coarse-grain implementation was introduced, which didn't give the expected results, mainly due to fact that MPI implementations have critical regions and locks inside their routines to allow threads to safely call them concurrently, but with enormous performance pitfalls.
During this work then, several different attempts were made to improve on the parallel performances of the YALES2 code, with moderate success. Mainly two different approaches were followed: modifying the data structures and intervening on the communications paradigms. The former allowed to obtain a mild improvement on the general performances of the code, while the introduction of the hybrid MPI+OpenMP models did not give any tangible benefit. This is possibly due to the fact that the work done on the two approaches was not complementary enough, in the sense that the data structures where modified in order to extract performance from the pure MPI communication model, but they were not modified in order to adapt them to the hybrid paradigms. On the other hand, during the implementation of the hybrid models, a lot of effort was put into trying to fit these models into the existing data structure and not into changing the latter to adapt it to the new communication paradigms.
In spite of the effort, only moderate performance benefits were obtained during this work, however, many issues of the code were addressed, a general improvement on the flexibility of the code was obtained, particularly thanks to the graph data structure and the abstraction of the communication routines into a dedicated interface.
To conclude, it is then extremely difficult to obtain high performances for massively parallel low-mach CFD solver on non-structured meshes. This is not an issue of YALES2 only, but it's common to all this family of codes due to their inherent characteristics, such as the low arithmetic intensity and the high sensitivity to the imbalance of communication in the Poisson solver. New techniques, like task based or asynchronous programming models, or even more advanced solutions like space-time mesh adaptation [146] should be introduced, providing at the same time more adapted data structures that allow to exploit such models fully and more naturally.

6.2 Perspectives

In Chapter 3 it was underlined how the performances of the deflation algorithm itself were not improved by the graph data structure due to the fact that the new P2P communication pattern was not able, by itself, to reduce the imbalance of that exchange. It was also shown how such imbalance came from the inequality in the number of ghost communicator across the different workers. A possible future development for the graph data structure could be to introduce a load balancing step at the moment of its construction in order to better balance the number of ghost communicators. At the moment the number of ghost communicators is not taken

into account at all during the graph construction. Depending on the partitioner used, the first domain decomposition is usually done minimising the edgecut, i.e. the amount of data exchanged between the resulting domains. The graph is then built as a new decomposition on top of the first one, consequently the number of parallel exchanges is already determined. Similarly to what happens when the graph is gathered on a smaller amount of workers, the vertices could be moved across the workers in order to balance the number of parallel exchanges as well as the number of nodes and edges.
The work presented in Chapter 4 showed that parallelising with multiple threads the work inside the deflation algorithm was not an easy task, especially for the continuous need of thread synchronisation and the fact that the algorithm is memory bound. Gathering the graph on a smaller amount of workers and having only one rank computing proved to be equivalent or even better performing but still limited by the bad communication patterns. The graph data structure also has reorganised the P2P exchanges in order to be performed all together at the beginning of the algorithm. Such configuration is ideal for the implementation of one-sided communications. Due to lack of time, one-sided communication were never properly tested during this work, however, all the mechanisms necessary to use them are in place. In spite of the bad performances of the P2P exchanges, the work exposed Chapter 5 showed promising results in the shared memory collective exchanges and smart use of shared memory windows. Another interesting development would be to introduce a mechanism to allocate the graph on shared memory windows rather than local memory, and use RDMA and one-sided communications to perform the ghost exchanges. This could help reduce dramatically the traffic on the network and speed-up the communication process. Furthermore, processes sharing a graph on the same window do not have to create ghost cells as they can directly access the memory of the neighbour processes, reducing again the number of parallel exchanges necessary and the memory consumption.
Such allocation mechanism would also be useful for the introduction of new paradigms alternative to MPI, such as PGAS, coarray-Fortran, etc. The work done on the abstraction of the communication routines will ease this process even further, however the parallel data structures (graph, ghost and external communicators) should be re-thought with this new exchange mode in mind.
Finally, GPGPUs are more and more present on modern clusters. CFD codes are particularly hard to adapt in order to efficiently exploit such architectures. There are different ongoing projects trying to efficiently porting YALES2 on GPU, and they showed that better performances are obtained when having one ElGrp rather than many small ones. The graph data structure allows to compute the fine grid efficiently on GPU and the deflation algorithm back on CPU since it decouples the concept of ElGrp from the deflation grid. However it should be possible, with moderate effort, to adapt the graph to this new architecture, for example changing the CSR format in which it is stored for others which are more appropriate for GPU computation.

Long résumé

Les deux dernières décennies ont vu une augmentation exponentielle de l'importance des simulations CFD (Computational Fluid Dynamics) dans le processus de conception industrielle. Cette évolution est due aux énormes progrès technologiques réalisés tant au niveau des logiciels que du matériel. D'une part, les logiciels CFD et les méthodes numériques sont devenus plus fiables et plus précis, d'autre part, la puissance de calcul a augmenté massivement, permettant de simuler des géométries plus complexes et plus raffinées en un temps raisonnable. En particulier, les architectures sont devenues massivement parallèles, c'est-à-dire composées de centaines de milliers de cœurs de calcul communiquant entre eux, et nous entrons aujourd'hui dans l'ère dite exascale [1], ce qui signifie que les machines sont désormais capables d'exécuter un nombre d'opérations par seconde de l'ordre de 10^{18} (exaFLOPs).
La puissance de calcul ne pourra pas croître indéfiniment à ce rythme, car les puces informatiques se heurtent à des barrières technologiques et l'efficacité énergétique devient un enjeu [2]. Les processeurs ont évolué en tenant compte de tous ces aspects, et les puces multicœurs sont désormais la norme dans les clusters HPC (High Performance Computing) modernes [3, 4], ainsi que les GPGPU (General Purpose Graphic Processing Unit) [5, 6]. Par conséquent, les logiciels doivent constamment s'adapter à l'évolution du matériel si l'on veut exploiter pleinement les capacités de calcul de ces machines.
L'objectif de ce travail est d'analyser les problèmes et les goulots d'étranglement qui empêchent un code CFD de s'adapter efficacement, c'est-à-dire de maintenir le même niveau de performance lorsqu'il est exécuté sur un grand nombre de cœurs, et de proposer des solutions pour surmonter ces obstacles.
Les modèles de communication classiques basés sur le communications MPI point-à-point et collectives ont probablement atteint la limite de leur efficacité [65] et deviennent de plus en plus le piège des performances dans les codes CFD.
Afin d'éviter ces goulets d'étranglement en matière de communication, des solutions plus adaptées au matériel ont été étudiées depuis longtemps, en particulier le modèle hybride OpenMP/MPI [41].
L'objectif de ce modèle hybride est de réduire l'empreinte mémoire et le coût de communication de l'implémentation MPI en utilisant un nombre plus faible de processus, sans réduire les ressources employées dans les phases de calcul, où les processus sont remplacés par des threads. Cette méthodologie est particulièrement adaptée à l'architecture multi-cœur où plusieurs threads peuvent tourner sur les différents cœurs en partageant la mémoire à l'intérieur d'un socket, bien que la complexité supplémentaire du mélange des deux modèles de programmation ne garantisse pas toujours une amélioration des performances par rapport aux implémentations MPI pures [44].
Toutefois, plusieurs travaux relatifs à l'hybridation des codes CFD, tels que [66],[67],[68] et [69], donnent des résultats encourageants et soutiennent que cela pourrait être la

bonne approche pour améliorer la scalabilité du code, ce qui motive les stratégies adoptées dans ce travail.
YALES2 [70, 71, 72], le code utilisé pour ce travail, est une boîte à outils numérique massivement parallèle développée depuis 2009 et actuellement utilisée à la fois dans le milieu universitaire et dans l'industrie. Il comprend de nombreux solveurs qui peuvent traiter divers problèmes physiques, tels que la simulation d'écoulements incompressibles, la combustion, les écoulements diphasiques, le transfert de chaleur, le transfert radiatif et bien plus encore.
Tous les différents solveurs reposent sur une bibliothèque numérique commune de haute performance, écrite en Fortran orienté objet. La bibliothèque de base comprend des solveurs linéaires hautement optimisés, le raffinement automatique du maillage [73] et l'équilibrage de la charge, l'intégration des volumes finis d'ordre élevé, les I/O parallèles, etc. Toutes ces bibliothèques sont conçues pour le calcul sur des maillages non structurés, qui sont mieux adaptés à la description de géométries complexes. L'approche des volumes finis centrée sur les nœuds, sur laquelle les méthodes numériques sont basées dans YALES2, garantit la conservation des quantités transportées. Un schéma de convection d'ordre 4 nouvellement implémenté [74] rend YALES2 particulièrement bien adapté aux simulations LES car les tourbillons ne sont pas artificiellement amortis lorsqu'ils sont transportés sur de grandes distances. Le parallélisme dans YALES2 est géré par une décomposition de domaine mise en œuvre avec le paradigme MPI, dans lequel le maillage est réparti entre plusieurs processus.
Le solveur de Navier-Stokes incompressible à basse fréquence est le plus utilisé et la plupart des méthodes numériques mises en œuvre pour ce solveur sont utilisées comme base pour les autres. Ce solveur utilise la procédure détaillée dans la section 1.3 pour discrétiser l'équation 1.15 sur un maillage, fournissant un système linéaire dans lequel les inconnues sont la pression à chaque nœud du maillage, qui est ensuite résolu avec les méthodes numériques présentées dans la section 1.3.2.
Plusieurs modèles ont été définis pour les performances du benchmark Preccinsta sur le cluster Myria. Le modèle de grille permet de déterminer toutes les quantités utiles liées à la taille de la grille à partir de seulement 3 paramètres connus, à l'exception de la taille du communicateur interne. L'algorithme DPCG a été modélisé en deux parties distinctes: l'itération sur les grilles fines et grossières. Un modèle a été obtenu pour chaque partie comme une somme de sous-modèles pour les phases de communication et de calcul respectives. Alors que le temps d'itération sur la grille fine peut être prédit avec une assez grande exactitude, la déflation s'est avérée plus difficile à modéliser avec la même précision. En particulier, les deux phases de communication dépendent fortement du nombre de voisins et du déséquilibre de ce nombre entre les processus. Ces paramètres sont beaucoup plus difficiles à modéliser a priori, car ils dépendent du partitionnement de la grille. Ces modèles ne sont qu'une première tentative rudimentaire de prédiction et d'étude des performances du code. Une analyse plus détaillée de chacun des sous-modèles est nécessaire pour mieux justifier les modèles eux-mêmes et les valeurs des coefficients qui ont été obtenus.

Cependant, malgré leur caractère approximatif, ils ont fourni un aperu utile des mécanismes de l'algorithme DPCG. En particulier, il a été montré que l'échange MPI réel n'a aucune influence sur le coût de la communication P2P de l'itération de la grille fine, qui dépend en fait exclusivement de la taille du communicateur interne.De plus, les communications collectives semblent avoir un coût assez constant, qui est indépendant du nombre de processus dans la communication.
En ce qui concerne la déflation, le modèle a montré que les échanges de communicateurs fantômes dépendent davantage du nombre de voisins que de la quantité de données échangées. Plus important encore, le coût de la communication collective dans la déflation peut être assez bien approximé par la différence entre le nombre maximum et moyen de communicateurs fantômes et par le nombre de travailleurs impliqués dans la communication.
En ce qui concerne l'influence de NELEMENTPERGROUP sur la performance globale de l'algorithme DPCG, on a montré comment la valeur optimale de ce paramètre est déterminée comme un compromis sur deux effets contrastés: la duplication des nœuds et des paires par rapport au nombre d'itérations de la grille fine.
Le fait d'avoir des groupes plus grands réduit la quantité de données dupliquées, mais rend aussi la grille de déflation plus grossière, ce qui fournit en retour une solution moins précise, d'où la nécessité d'un plus grand nombre d'itérations de la grille fine pour atteindre la convergence. Cela implique que deux aspects de l'algorithme DPCG doivent être améliorés.
Premièrement, le nombre de "nœuds" sur la grille de déflation ($n_{ElGrp_{[gt]}}$) devrait être indépendant de la taille des groupes d'éléments sur la grille fine. Cela permettrait d'avoir un autre degré de liberté pour mieux ajuster la convergence de la grille de déflation, en donnant la possibilité de découpler le nombre d'itérations sur les deux grilles de la duplication des nœuds et des paires et de la taille du communicateur interne. En conséquence, des groupes plus grands pourraient être utilisés sur la grille fine pour réduire cette duplication, tandis qu'une grille moins grossière peut être utilisée pour réguler le nombre d'itérations sur la grille fine et la grille de déflation.
Deuxièmement, un travail doit être fait pour réduire le coût des communications dans l'itération de déflation. Un nombre plus uniforme de communicateurs fantômes réduirait le déséquilibre dans l'échange P2P, ce qui à son tour diminuerait le temps de synchronisation dans la communication collective. Enfin, la réduction du nombre de processus impliqués dans l'échange collectif permettrait de réduire sensiblement le coût de ce dernier, comme le montre clairement la Figure 2.49.
Dans 2.2.6, certaines des limites de l'implémentation actuelle de l'algorithme DPCG dans YALES2 ont été mises en lumière. En particulier, il a été démontré qu'il est nécessaire de découpler la grille de déflation du mécanisme de blocage de cache des ElGrps, tandis que la figure 2.34 a exposé les mauvais schémas de communication dus à la construction actuelle de la connectivité.
Même si cette dernière aurait pu être facilement corrigée avec un système de communication full-halo par exemple, la première nécessite une réorganisation complète de la grille de déflation. Par conséquent, une nouvelle structure de données appelée

graph a été introduite dans YALES2 pour faire face à tous ces aspects négatifs.
La structure de données de graphe présentée dans le chapitre 3 fournit un support différent pour la déflation, permettant une telle séparation. Les mesures exposées dans 3.2 ont montré comment le degré de liberté supplémentaire permettait une nette amélioration des performances du solveur DPCG.
En particulier, il a été montré que la double décomposition de domaine n'apporte aucun avantage réel dans l'accélération du code.
Le blocage L1, c'est-à-dire l'utilisation d'une taille de groupe qui tiendrait dans le cache L1, pourrait apporter une amélioration de 40% pour certaines boucles clés du code, mais il a été démontré qu'il y a une pénalisation beaucoup plus importante résultant de la duplication massive de nœuds et de paires impliquée par de si petits groupes.
En outre, une ElGrp plus petite signifie également un communicateur interne plus grand, et par conséquent une surcharge plus importante pour sa mise à jour. Pour la performance globale, avec la structure de données actuelle de YALES2, la minimisation de la duplication des données est en fait beaucoup plus pertinente que l'amélioration des modèles d'accès à la mémoire. D'autres approches devraient être étudiées pour tirer parti du blocage du cache sans pénaliser les données dupliquées.
Un autre point sensible de l'algorithme de déflation identifié dans le chapitre 2 était le schéma de communication sous-optimal dû au système de communication *ghost* avec un demi-halo, qui provoquait un déséquilibre important entre les différents processus et par conséquent un retard sur la communication collective. La nouvelle structure de données a tenté de résoudre le problème avec une communication *ghost* symétrique full-halo, mais il a été montré que cela ne pouvait pas éviter le déséquilibre causé par la différence de quantité d'échanges P2P.
Afin d'améliorer ce point, un système où plusieurs couches de cellules fantômes sont échangées a été mis en place. Il a été montré que cette idée, qui a partiellement fonctionné sur un benchmark synthétique 1D, n'était pas efficace sur des cas réels car le nombre de voisins augmente rapidement avec l'augmentation des couches *ghost*.
Une autre technique, consistant à rassembler le graphe de déflation sur un sous-ensemble de processus afin de réduire le nombre de processus communicants a également été mise en place sans grand succès, à l'exception de quelques cas. Le principal problème de cette méthode est que le travail supplémentaire requis par ce sous-ensemble de processus n'est pas compensé par une amélioration suffisante du temps de communication.
Une version modifiée de l'algorithme 6 qui utilise des opérations supplémentaires et des communications collectives non bloquantes [96] pour essayer de superposer communication et calcul est également implémentée dans YALES2. Bien que non présentés ici, quelques tests de comparaison ont été effectués entre les deux algorithmes avec la déflation basée sur ElGrp, mais aucune amélioration n'a été obtenue par cette version non bloquante par rapport à la version standard.
Bien qu'une analyse plus précise soit nécessaire, les tests préliminaires ont montré que l'échec à fournir de meilleures performances était dû au fait que le retard causé par l'échange de fantômes est plus important que le temps de calcul que l'algorithme

tente de couvrir. De plus, comme les processus qui ont plus de communications à effectuer ont également plus de paires ElGrp à calculer, il n'y a aucun moyen pour eux de rattraper leur retard, par conséquent il existera toujours un point de synchronisation, même si les processus sont d'une certaine manière biaisés le long de l'algorithme. Il serait cependant intéressant de tester un tel algorithme avec la nouvelle structure de données.
En outre, d'autres optimisations possibles de l'algorithme DPCG ont été proposées [97, 98, 99], qui pourraient également être ajoutées à la bibliothèque YALES2. La nouvelle structure de données de graphe pourrait fournir la flexibilité nécessaire pour tirer parti de telles implémentations.
Les deux approches complémentaires détaillées dans le chapitre 3, les multiples couches de comunicateurs *ghost* et la collecte du graphe sur un sous-ensemble de processus, mises en œuvre pour tenter de gérer le coût des échanges parallèles dans l'itération de déflation ont échoué en raison du coût de calcul supplémentaire dépassant le bénéfice d'une quantité réduite de communication. En particulier, la collecte du graphe sur un sous-ensemble restreint de processus signifie que ceux-ci ont plus de travail à effectuer pendant que les autres sont inactifs et ne contribuent pas au calcul, ce qui constitue un gaspillage de ressources, en plus du coût de la collecte et de la dispersion des données.
La deuxième partie de ce travail a essayé de traiter ce problème et d'autres en introduisant un deuxième niveau de parallélisation dans le code: les threads OpenMP. Les exemples de codes scientifiques expérimentant le modèle de programmation hybride MPI+OpenMP sont de plus en plus courants et peuvent être trouvés dans une variété d'applications [100, 101, 68, 102], y compris la CFD [66, 103, 104, 69, 105, 106, 107]. Ceci est justifié par le fait que les codes MPI purs présentent des difficultés à passer à l'échelle de centaines de milliers de processus et ont une empreinte mémoire élevée [108]. L'ajout d'une couche supplémentaire de threads OpenMP permet de réduire le nombre de rangs MPI et d'améliorer l'efficacité de la communication, tout en répartissant le travail entre les différents threads.
Les threads sont plus légers que les processus MPI, ils ont donc un impact plus faible sur le système. En outre, cette approche est intrinsèquement liée au matériel, car les threads appartenant au même processus doivent être placés sur la même région NUMA, partageant un certain niveau de mémoire cache (généralement L3).
L'ajout d'une couche OpenMP à un programme MPI pourrait alors contribuer à améliorer les performances globales, en particulier sur des milliers de cœurs [109] et des processeurs à plusieurs cœurs avec une grande mémoire cache partagée [110].
Cependant, il n'est pas trivial d'exploiter un tel modèle de programmation hybride et d'obtenir de meilleures performances que le code MPI pur [111], notamment lorsque la structure du code ne permet pas un niveau supplémentaire de partage du travail ou que les algorithmes doivent être adaptés [112, 44].
Dans la plupart des exemples cités ci-dessus, le parallélisme OpenMP est exploité en assignant une partie de la maille de chaque processus à un thread pour le calcul ou par parallélisation de boucle. Dans YALES2, les deux approches peuvent être fusionnées car sa structure de données rend le code particulièrement adapté pour être

parallélisé avec un niveau de boucle OpenMP, puisque tous les algorithmes consistent en des boucles sur les groupes d'éléments complètement indépendants (ElGrps), qui représentent une décomposition du maillage.
Cette approche, également appelée OpenMP *fine grain*, a l'avantage d'être facile à mettre en œuvre et le code peut être parallélisé progressivement, la parallélisation étant obtenue en ajoutant de simples pragmas autour des boucles du code.
Avant le début de ce travail, la plupart des boucles du code étaient déjà entourées de pragmas OpenMP. Cependant, pour certaines parties du code, la parallélisation du multithreading n'est pas si immédiate. Cela peut être causé par le fait que certaines régions ne contiennent pas de boucles (I/O) ou parce que les itérations des boucles ne sont pas indépendantes. La principale condition pour qu'une boucle puisse être parallélisée avec plusieurs threads est que le travail de chaque thread (c'est-à-dire les itérations de la boucle) doit être indépendant, sinon des *race conditions* sur les données peuvent se produire.
C'est principalement le cas pour la mise à jour du communicateur interne et l'algorithme de déflation, dont les opérations ne sont pas effectuées sur des ensembles indépendants de données. Les défis et les techniques utilisés pour essayer de paralléliser ces parties importantes du code sont détaillés dans la section 4.1 et la section 4.2 respectivement.
La mise à jour parallèle du communicateur interne et l'algorithme de dégonflement se sont avérés particulièrement difficiles à paralléliser efficacement. Afin d'éviter les *race conditions* sur les données dans la mise à jour du IC, plusieurs techniques ont été mises en place et comparées, mais toutes impliquent une sorte d'opération supplémentaire qui augmente la surcharge de cette opération.
La déflation souffre plutôt du surcoût excessif des différentes régions parallèles qui annule les avantages de la parallélisation du travail. Une tentative d'utiliser des tâches pour superposer la communication et le calcul a également été faite, mais elle n'a pas abouti. Enfin, les performances de la version purement MPI et de la version hybride ont été comparées, montrant comment les problèmes précédemment exposés ont empêché la seconde de mieux fonctionner que la première, malgré les différentes tentatives d'optimisation. Cependant, ce travail a également donné quelques points intéressants, notamment la réduction du coût de la communication collective et le partage efficace du travail dans l'algorithme de déflation montré pour le maillage de $14M$ et pour le maillage de $878M$ où la version hybride est aussi performante que la version pure MPI.
En effet, le fait d'avoir moins de rangs communiquant sans perdre de capacité de calcul est une bonne stratégie, dans ce cas cependant pénalisée par l'overhead OpenMP.
Il est également important de mentionner que, si la boucle temporelle consiste principalement en des boucles sur les ElGrps, plusieurs autres parties du code ne peuvent pas être parallélisées aussi facilement. Il peut s'agir par exemple des opérations d'I/O ou même du partitionnement initial du maillage et de la création des structures de données, qui, dans la version hybride, sont effectués uniquement par le thread maître. Cela a pour effet d'augmenter massivement le temps passé dans ces phases par rapport à la version MPI pure, car la taille du maillage est beaucoup

plus grande pour la version hybride.
En outre, si l'on exclut ce qui est consommé par la bibliothèque MPI elle-même, la réduction de la duplication des données est minime, car les groupes d'éléments dupliquent eux-mêmes des données, et seule une petite partie des communicateurs internes et externes est réellement èpargnée.
En essayant de paralléliser également ces phases et d'optimiser davantage la boucle temporelle, on a obtenu l'implémentation hybride MPI et OpenMP à *coarse grain* présentée au chapitre 5.
Contrairement à ce qui se passe dans le modèle *fine grain*, où les threads sont créés et tués autour de boucles à forte intensité de travail, dans cette version, la région parallèle OpenMP est ouverte une fois au début de l'exécution et les threads sont maintenus en vie jusqu'à la fin du programme, éliminant ainsi la surcharge causée par la création de multiples régions parallèles, comme représenté schématiquement dans la Figure 5.1.
La principale implication du modèle à *coarse grain* est que tous les threads doivent communiquer, c'est pourquoi des mécanismes de communication collective à mémoire partagée et P2P ont été introduits.
Pour les communications collectives à mémoire partagée présentées dans la section 5.2, il a été possible d'obtenir des performances comparables, voire supérieures, à certaines implémentations MPI.
Cependant, pour les communications P2P, malgré les efforts déployés pour permettre aux threads d'un même rang de communiquer via la mémoire partagée, les régions critiques et les verrous présents dans les routines des implémentations MPI les plus courantes ont causé d'énormes problèmes de performance lorsque les threads doivent communiquer avec d'autres rangs (c'est-à-dire en utilisant les routines MPI). L'impact négatif de la séquentialisation interne des routines MPI est tel que l'ensemble du temps d'exécution du code est affecté et qu'il a été impossible d'obtenir des performances même comparables à celles du modèle MPI pur ou du modèle à *fine grain*.
Une brève tentative d'utiliser des communications *one-sided* a été faite, mais celles-ci souffrent des mêmes pièges que les communications bilatérales et n'ont pas été développées davantage.
En conclusion, le modèle hybride MPI+OpenMP *coarse grain* a permis d'introduire plusieurs améliorations dans le code, telles que la *thread safety* totale, l'interface de communication abstraite et certaines structures de base pour la mémoire partagée et les opérations MPI *one sided*, mais les pièges de performance dans les échanges P2P l'empêchent d'être efficace. De meilleures performances pourraient éventuellement être obtenues avec des implémentations MPI entièrement multithreadées ou avec des modifications plus profondes des structures de données du code.
Pour résumer ce qui a été présenté dans ce travail, le chapitre 1 a donné un aperu des méthodes numériques, des paradigmes de communication parallèle et des outils HPC utilisés en CFD. Le chapitre 2 a présenté la structure des données et quelques mesures de performance de YALES2, le code CFD utilisé dans ce travail, en essayant de donner quelques modèles simplistes de ses performances, en soulignant ses points forts et en exposant ses problèmes et ses goulets d'étranglement.

Le chapitre 3 présente une nouvelle structure de données, qui a été implémentée pour essayer de résoudre les problèmes exposés dans le chapitre précédent. Le chapitre 4 décrit le travail effectué pour l'introduction d'une implémentation OpenMP *fine grain* afin de diviser le travail entre plusieurs threads et de réduire le nombre de rangs MPI communiquant.
Le chapitre 5 a présenté les efforts réalisés pour mettre en œuvre un modèle OpenMP *coarse grain* visant à résoudre les problèmes du modèle *fine grain*. Les mesures de performance et les modèles relatifs obtenus au chapitre 2 ont montré que la plupart du code s'adapte parfaitement à un nombre relativement élevé de rangs MPI.
Ce n'est pas le cas de l'algorithme de déflation, qui s'échelonne mal, même avec un faible nombre de rangs. La déflation, cependant, est la partie centrale du solveur linéaire de Poisson, où la plupart du temps de calcul est passé. Il a été démontré que le manque de scalabilité est principalement dû au coût élevé de la communication par rapport au calcul dans cette partie particulière du code. De plus, le temps de communication est fortement influencé par le déséquilibre de la charge de calcul, mais surtout par le nombre d'échanges point à point.
La structure de données du graphe introduite dans le chapitre 3 a tenté de résoudre ces problèmes en modifiant la structure des cellules *ghost* afin d'améliorer le modèle de communication P2P et d'équilibrer la charge de calcul sur le nombre d'arêtes.
En outre, la possibilité d'utiliser plusieurs couches *ghost* et de rassembler le graphe sur un sous-ensemble de processus a été introduite, mais n'a pas donné les avantages escomptés. En particulier, les couches *ghost* multiples ont souffert d'une augmentation rapide de la quantité de communication P2P nécessaire et du coût de calcul. Le graphe rassemblé a également souffert d'une augmentation du travail par rang, qui, dans la plupart des cas, dépassait les avantages en termes de temps de communication.
Pour résoudre ces problèmes, un modèle OpenMP *fine grain* a été introduit dans le chapitre 4. La parallélisation efficace de l'algorithme de déflation s'est avérée difficile en raison du besoin de synchronisation des threads et de la limitation de la mémoire du code. En fin de compte, le modèle MPI pur ou le graphe rassemblé étaient plus performants que l'implémentation hybride *fine grain*.
Pour surmonter les limites du modèle à grain fin, une implémentation OpenMP *coarse grain* a été introduite, mais elle n'a pas donné les résultats escomptés, principalement en raison du fait que les implémentations MPI ont des régions critiques et des verrous à l'intérieur de leurs routines pour permettre aux threads de les appeler simultanément en toute sécurité, mais avec d'énormes problèmes de performance.
Au cours de ce travail, plusieurs tentatives différentes ont été faites pour améliorer les performances parallèles du code YALES2, avec un succès modéré. Deux approches différentes ont été suivies : modifier les structures de données et intervenir sur les paradigmes de communication.
La première a permis d'obtenir une légère amélioration des performances générales du code, tandis que l'introduction des modèles hybrides MPI+OpenMP n'a pas apporté d'avantage tangible. Cela est peut-être dû au fait que le travail effectué sur les deux approches n'était pas assez complémentaire, dans le sens où les structures de

données ont été modifiées afin d'extraire les performances du modèle de communication MPI pur, mais elles n'ont pas été modifiées afin de les adapter aux paradigmes hybrides.
D'autre part, lors de la mise en œuvre des modèles hybrides, beaucoup d'efforts ont été déployés pour essayer d'intégrer ces modèles dans la structure de données existante et non pour modifier cette dernière afin de l'adapter aux nouveaux paradigmes de communication. Malgré ces efforts, seuls des bénéfices modérés en termes de performance ont été obtenus au cours de ce travail, cependant, de nombreux problèmes du code ont été résolus, une amélioration générale de la flexibilité du code a été obtenue, en particulier grâce à la structure de données du graphe et à l'abstraction des routines de communication dans une interface dédiée.
Pour conclure, il est donc extrêmement difficile d'obtenir des performances élevées pour un solveur CFD massivement parallèle à basse fréquence sur des maillages non structurés. Ce n'est pas un problème propre à YALES2, mais il est commun à toute cette famille de codes en raison de leurs caractéristiques inhérentes, telles que la faible intensité arithmétique et la grande sensibilité au déséquilibre de la communication dans le solveur de Poisson.
De nouvelles techniques, comme les modèles de programmation asynchrones ou basés sur les tâches, ou même des solutions plus avancées comme l'adaptation de la maille spatio-temporelle [146], devraient être introduites, fournissant en même temps des structures de données plus adaptées qui permettent d'exploiter les capacités de communication du solveur de Poisson.
Dans le chapitre 3, il a été souligné que les performances de l'algorithme de déflation lui-même n'ont pas été améliorées par la structure de données du graphe, car le nouveau modèle de communication P2P n'était pas capable, à lui seul, de réduire le déséquilibre de cet échange.
Il a également été démontré que ce déséquilibre provenait de l'inégalité du nombre de communicateurs fantômes entre les différents travailleurs. Un développement futur possible pour la structure de données du graphe pourrait être d'introduire une étape d'équilibrage de la charge au moment de sa construction afin de mieux équilibrer le nombre de communicateurs *ghost*. Pour l'instant, le nombre de communicateurs *ghost* n'est pas du tout pris en compte lors de la construction du graphe. En fonction du partitionneur utilisé, la première décomposition du domaine est généralement effectuée en minimisant l'edgecut, c'est-à-dire la quantité de données échangées entre les domaines résultants.
Le graphe est ensuite construit comme une nouvelle décomposition par-dessus la première, par conséquent le nombre d'échanges parallèles est déjà déterminé. De la même manière que lorsque le graphe est rassemblé sur un plus petit nombre de travailleurs, les sommets peuvent être déplacés entre les travailleurs afin d'équilibrer le nombre d'échanges parallèles ainsi que le nombre de nœuds et d'arêtes.
Le travail présenté dans le chapitre 4 a montré que la parallélisation avec plusieurs threads du travail dans l'algorithme de déflation n'était pas une tâche facile, surtout en raison du besoin continu de synchronisation des threads et du fait que l'algorithme est limité par la mémoire. Rassembler le graphe sur un plus petit nombre de tra-

vailleurs et n'avoir qu'un seul rang de calcul s'est avéré équivalent ou même plus performant, mais toujours limité par les mauvais schémas de communication.
La structure de données du graphe a également réorganisé les échanges P2P afin qu'ils soient effectués tous ensemble au début de l'algorithme. Cette configuration est idéale pour la mise en œuvre de communications *one sided*. Par manque de temps, les communications *one-sided* n'ont pas été testées correctement au cours de ce travail, cependant, tous les mécanismes nécessaires à leur utilisation sont en place.
Malgré les mauvaises performances des échanges P2P, le travail exposé au Chapitre 5 a montré des résultats prometteurs dans les échanges collectifs en mémoire partagée et l'utilisation intelligente des fenêtres de mémoire partagée.
Un autre développement intéressant serait d'introduire un mécanisme pour allouer le graphe sur des fenêtres de mémoire partagée plutôt que sur la mémoire locale, et d'utiliser RDMA et les communications *one-sided* pour effectuer les échanges *ghost*. Cela pourrait contribuer à réduire considérablement le trafic sur le réseau et à accélérer le processus de communication.
De plus, les processus partageant un graphe sur la même fenêtre n'ont pas besoin de créer des cellules *ghost* car ils peuvent accéder directement à la mémoire des processus voisins, ce qui réduit encore le nombre d'échanges parallèles nécessaires et la consommation de mémoire. Un tel mécanisme d'allocation serait également utile pour l'introduction de nouveaux paradigmes alternatifs à MPI, tels que PGAS, coarray-Fortran, etc.
Le travail effectué sur l'abstraction des routines de communication facilitera encore plus ce processus, mais les structures de données parallèles (graphe, communicateurs *ghost* et externes) devraient être repensées en tenant compte de ce nouveau mode d'échange.
Enfin, les GPGPU sont de plus en plus présents sur les clusters modernes. Les codes CFD sont particulièrement difficiles à adapter afin d'exploiter efficacement ces architectures. Différents projets en cours tentent de porter efficacement YALES2 sur GPU, et ils ont montré que de meilleures performances sont obtenues en ayant une seule ElGrp plutôt que plusieurs petites. La structure de données du graphe permet de calculer efficacement la grille fine sur GPU et l'algorithme de déflation sur CPU puisqu'elle découple le concept de ElGrp de la grille de déflation. Cependant, il devrait être possible, avec un effort modéré, d'adapter le graphe à cette nouvelle architecture, par exemple en changeant le format CSR dans lequel il est stocké pour d'autres qui sont plus appropriés pour le calcul sur GPU.

Bibliography

[1] Fabrizio Gagliardi, Miquel Moreto, Mauro Olivieri, and Mateo Valero. The international race towards Exascale in Europe. *CCF Transactions on High Performance Computing*, 1(1):3–13, 2019.

[2] Nikola Rajovic, Lluis Vilanova, Carlos Villavieja, Nikola Puzovic, and Alex Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439–443, 2013.

[3] Christian Märtin. Multicore Processors: Challenges, Opportunities, Emerging Trends. *Embedded World Conference*, pages 25–27, 2014.

[4] Paweł Gepner and Michał F. Kowalik. Multi-core processors: New way to achieve high system performance. *PARELEC 2006 - Proceedings: International Symposium on Parallel Computing in Electrical Engineering*, pages 9–13, 2006.

[5] Stan Posey. Considerations for GPU acceleration of parallel CFD. *Procedia Engineering*, 61:388–391, 2013.

[6] Gang Chen, Guobo Li, Songwen Pei, and Baifeng Wu. High performance computing via a GPU. *2009 1st International Conference on Information Science and Engineering, ICISE 2009*, pages 238–241, 2009.

[7] Alexandre Joel Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22(104):745–745, 1968.

[8] J Kim and P Moin. *Application of a Fractional - Step Method to Incompressible Navier - Stokes Equation.* 1984.

[9] Katuhiko Goda. A multistep technique with implicit difference schemes for calculating two- or three-dimensional cavity flows. *Journal of Computational Physics*, 30(1):76–95, 1979.

[10] M. H. Carpenter, C. A. Kennedy, Hester Bijl, S. A. Viken, and Veer N. Vatsa. Fourth-order Runge-Kutta schemes for fluid mechanics applications. *Journal of Scientific Computing*, 25(1):157–194, 2005.

[11] Matthias Kraushaar. Application of the compressible and low-mach number approaches to large-eddy simulation of turbulent flows in aero-engines. dec 2011.

[12] F. H. Lee, K. K. Phoon, K. C. Lim, and S. H. Chan. Performance of Jacobi preconditioning in Krylov subspace solution of finite element equations. *International Journal for Numerical and Analytical Methods in Geomechanics*, 26(4):341–372, 2002.

[13] J. M. Tang, R. Nabben, C. Vuik, and Y. A. Erlangga. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing*, 39(3):340–370, 2009.

[14] Gregory F. Pfister. An introduction to the InfiniBand architecture. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 617–632, 2001.

[15] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. *Proceedings - 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, HOTI 2015*, pages 1–9, 2015.

[16] Declan Delaney, Tomás Ward, and Seamus McLoone. On Consistency and Network Latency in Distributed Interactive Applications: A Survey-Part I. *Presence: Teleoperators and Virtual Environments*, 15(2):218–234, apr 2006.

[17] Amitava Dutta-Roy. The cost of quality in Internet-style networks. *IEEE Spectrum*, 37(9):57–62, sep 2000.

[18] https://developer.nvidia.com/about-cuda.

[19] https://www.openacc.org/.

[20] Aaftab Munshi. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, aug 2009.

[21] https://www.top500.org/lists/green500.

[22] Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*, volume 2015-Decem, pages 739–748. IEEE, sep 2015.

[23] Johannes Hofmann, Jan Eitzinger, and Dietmar Fey. Execution-Cache-Memory Performance Model: Introduction and Validation. sep 2015.

[24] www.top500.org.

[25] G M Moore. Moore's Law ,Electronics. *Electronics*, 38(8):114, 1965.

[26] Jack J Dongarra. The LINPACK Benchmark: An explanation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 297 LNCS, pages 456–474. 1988.

[27] https://www.hpcg-benchmark.org.

[28] https://www.top500.org/lists/hpcg/list/2020/06.

[29] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful Visual Performance model for multicore Architectures. *Communications of the ACM*, 52(4):65–76, apr 2009.

[30] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967*, pages 483–485, 1967.

[31] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, may 1988.

[32] https://www.mpich.org/.

[33] https://mvapich.cse.ohio-state.edu/.

[34] https://www.open-mpi.org/.

[35] https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html.

[36] MPI : A Message-Passing Interface Standard, 2012.

[37] Tarick Bedeir. RDMA Read and Write with IB Verbs. *Manual*, pages 1–8, 2010.

[38] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. The design and implementation of MPI collective operations for clusters in long-and-fast networks. *Cluster Computing*, 11(1):45–55, 2008.

[39] Shigang Li, Torsten Hoefler, and Marc Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing - HPDC '13*, page 85, New York, New York, USA, 2013. ACM Press.

[40] D. Grünewald and Christian Simmendinger. The GASPI API specification and its implementation GPI 2.0. *7th International Conference on ...*, 2013.

[41] Ewing Lusk and Anthony Chan. Early experiments with the openmp/mpi hybrid programming model. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5004 LNCS, pages 36–47, 2008.

[42] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5759 LNCS, pages 94–103, 2009.

[43] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35(12):608–617, 2009.

[44] Georg Hager, Gabriele Jost, and Rolf Rabenseifner. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. IEEE, feb 2009.

[45] Damián A Mallón, Guillermo L Taboada, Carlos Teijeiro, Juan Touriño, Basilio B Fraguela, Andrés Gómez, Ramón Doallo, and J Carlos Mouriño. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5759 LNCS, pages 174–184, 2009.

[46] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming*, 20(2):89–114, 2012.

[47] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, feb 2011.

[48] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, nov 2012.

[49] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX - A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models - PGAS '14*, volume 2014-Octob, pages 1–11, New York, New York, USA, 2014. ACM Press.

[50] Lee Howes and Maria Rovatsou. SYCL Integrates OpenCL devices with modern C++. pages 1–274, 2014.

[51] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, sep 2010.

[52] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. *Tools for High Performance Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[53] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):n/a–n/a, 2009.

[54] Lamia Djoudi and Denis Barthou. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. *Workshop on EPIC architectures and compiler technology*, pages 1–20, 2005.

[55] Andres S Charif-Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. CQA: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Goa, India, dec 2014. IEEE.

[56] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[57] Intel Corporation. VTune Performance Analyzer.

[58] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[59] Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency Computation Practice and Experience*, 22(6):702–719, 2010.

[60] Andreas Knüpfer and Christian Rössel. Score-P - A Joint Performance Measurement Run-Time Infrastructure for. pages 1–12, 2011.

[61] https://tools.bsc.es/extrae.

[62] V Pillet, J Labarta, T Cortes, and S Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. *Proceedings of WoTUG-18: Transputer and occam Developments*, (February):17–31, 1995.

[63] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[64] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. pages 49–49, 2015.

[65] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in MPI: A case study with MPICH2. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6960 LNCS, pages 140–149, 2011.

[66] M.J. Berger, M.J. Aftosmis, D.D. Marshall, and S.M. Murman. Performance of a new CFD flow solver using a hybrid programming paradigm. *Journal of Parallel and Distributed Computing*, 65(4):414–423, apr 2005.

[67] Xiaohu Guo, Michael Lange, Gerard Gorman, Lawrence Mitchell, and Michèle Weiland. Developing a scalable hybrid MPI/OpenMP unstructured finite element model. *Computers and Fluids*, 110:227–234, mar 2015.

[68] Bogdan Satarić, Vladimir Slavnić, Aleksandar Belić, Antun Balaž, Paulsamy Muruganandam, and Sadhan K. Adhikari. Hybrid OpenMP/MPI programs for solving the time-dependent Gross-Pitaevskii equation in a fully anisotropic trap. *Computer Physics Communications*, 200:411–417, mar 2016.

[69] E Martin, F Renac, Parallel Scal, and Performance Measurement. HYBRID MPI / OPENMP PARALLEL STRATEGIES FOR A HIGH ORDER DISCONTINUOUS GALERKIN SOLVER IN. (Wccm Xi):5–6, 2014.

[70] Vincent Moureau, Pascale Domingo, and Luc Vervisch. Une algorithmique optimisée pour le supercalcul appliqué à la mécanique des fluides numérique. *Comptes Rendus - Mecanique*, 339(2-3):141–148, 2011.

[71] Vincent Moureau, P. Domingo, and Luc Vervisch. From Large-Eddy Simulation to Direct Numerical Simulation of a lean premixed swirl flame: Filtered laminar flame-PDF modeling. *Combustion and Flame*, 158(7):1340–1357, jul 2011.

[72] Mathias Malandain. Simulations massivement parallèles des écoulements turbulents à faible nombre de Mach. 2013.

[73] Pierre Benard, Guillaume Balarac, Vincent Moureau, Cecile Dobrzynski, Ghislain Lartigue, and Yves D'Angelo. Mesh adaptation for large-eddy simulations in complex geometries. *International Journal for Numerical Methods in Fluids*, 81(12):719–740, aug 2016.

[74] Manuel Bernard, Ghislain Lartigue, Guillaume Balarac, Vincent Moureau, and Guillaume Puigt. A framework to perform high-order deconvolution for finite-volume method on simplicial meshes. *International Journal for Numerical Methods in Fluids*, M:1–36, 2020.

[75] Mathieu Desbrun, Anil N. Hirani, Melvin Leok, and Jerrold E. Marsden. Discrete Exterior Calculus. pages 1–53, aug 2005.

[76] Mathieu Desbrun, Eva Kanso, and Yiying Tong. Discrete differential forms for computational modeling. In *ACM SIGGRAPH 2006 Courses on - SIGGRAPH '06*, page 39, New York, New York, USA, 2006. ACM Press.

[77] Mathias Malandain, Nicolas Maheu, and Vincent Moureau. Optimization of the deflated Conjugate Gradient algorithm for the solving of elliptic equations on massively parallel machines. *Journal of Computational Physics*, 238:32–47, apr 2013.

[78] W MEIER, P WEIGAND, X DUAN, and R GIEZENDANNERTHOBEN. Detailed characterization of the dynamics of thermoacoustic pulsations in a lean premixed swirl flame. *Combustion and Flame*, 150(1-2):2–26, jul 2007.

[79] Vincent Moureau, P Minot, H Pitsch, and C. Bérat. A ghost-fluid method for large-eddy simulations of premixed combustion in complex geometries. *Journal of Computational Physics*, 221(2):600–614, feb 2007.

[80] P. Benard, G. Lartigue, V. Moureau, and R. Mercier. Large-Eddy Simulation of the lean-premixed PRECCINSTA burner with wall heat loss. *Proceedings of the Combustion Institute*, 37(4):5233–5243, 2019.

[81] https://ark.intel.com/content/www/us/en/ark/products/91754/intel-xeon-processor-e5-2680-v4-35m-cache-2-40-ghz.html.

[82] https://ark.intel.com/content/www/us/en/ark/products/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz.html.

[83] https://www.amd.com/fr/products/cpu/amd-epyc-7h12.

[84] CP-Algorithms, String Hashing.

[85] www.json.org.

[86] J. M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 34(3):249–264, mar 2004.

[87] K. Al-Tawil and C.A. Moritz. LogGP quantified: the case for MPI. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 366–367. IEEE Comput. Soc.

[88] Khalid Al-Tawil and Csaba Andras Moritz. Performance Modeling and Evaluation of MPI. *Journal of Parallel and Distributed Computing*, 61(2):202–223, feb 2001.

[89] Paul R. Eller, Torsten Hoefler, and William Gropp. Using performance models to understand scalable Krylov solver performance at scale for structured grid problems. *Proceedings of the International Conference on Supercomputing*, pages 138–149, 2019.

[90] F. Pellegrini. Distillating knowledge about Scotch. *Combinatorial Scientific Computing*, pages 1–12, 2009.

[91] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, jan 1998.

[92] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9220 LNCS:117–158, nov 2013.

[93] Pavanakumar Mohanamuraly and Gabriel Staffelbach. Hardware locality-aware partitioning and dynamic load-balancing of unstructured meshes for large-scale scientific applications. 2020.

[94] William Jalby, David Kuck, Allen D. Malony, Michel Masella, Abdelhafid Mazouz, and Mihail Popov. The Long and Winding Road Toward Efficient High-Performance Computing. *Proceedings of the IEEE*, 106(11):1985–2003, nov 2018.

[95] Loïc Thébault and Eric Petit. Asynchronous and multithreaded communications on irregular applications using vectorized divide and conquer approach. *Journal of Parallel and Distributed Computing*, 114:16–27, apr 2018.

[96] P Ghysels and W Vanroose. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Computing*, 40(7):224–238, jul 2014.

[97] Mark Frederick Hoemmen. *Communication-avoiding Krylov Subspace Methods*. PhD thesis, 2010.

[98] Paul R. Eller and William Gropp. Scalable Non-blocking Preconditioned Conjugate Gradient Methods. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, volume 0, pages 204–215. IEEE, nov 2016.

[99] James Lin, Minhua Wen, Delong Meng, Xin Liu, Akira Nukada, and Satoshi Matsuoka. Optimizing preconditioned conjugate gradient on taihulight for OpenFOAM. *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018*, pages 273–282, 2018.

[100] Francisco Borges, Albert Gutierrez-Milla, Remo Suppi, and Emilio Luque. A Hybrid MPI+OpenMP Solution of the Distributed Cluster-based Fish Schooling Simulator. *Procedia Computer Science*, 29:2111–2120, 2014.

[101] Dana Akhmetova, Roman Iakymchuk, Orjan Ekeberg, and Erwin Laure. Performance study of multithreaded MPI and Openmp tasking in a large scientific code. *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, (June):756–765, 2017.

[102] M.F. Su, Ihab El-Kady, D.A. Bader, and S.-Y. Lin. A novel FDTD application featuring OpenMP-MPI hybrid parallelization. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 373–379 vol.1. IEEE, 2004.

[103] Pablo D. Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, 37(6-7):316–326, jun 2011.

[104] Yohei Sato, Takanori Hino, and Kunihide Ohashi. Parallelization of an unstructured Navier-Stokes solver using a multi-color ordering method for OpenMP. *Computers & Fluids*, 88:496–509, dec 2013.

[105] F Bassi, A Colombo, A Crivellini, and M Franciolini. Hybrid OPENMP/MPI parallelization of a high-order Discontinuous Galerkin CFD/CAA solver. In *ECCOMAS Congress 2016 - Proceedings of the 7th European Congress on Computational Methods in Applied Sciences and Engineering*, volume 4, pages 7992–8012, 2016.

[106] Charles Henri Bruneau and Khodor Khadra. Highly parallel computing of a multigrid solver for 3D Navier-Stokes equations. *Journal of Computational Science*, 17:35–46, 2016.

[107] Pablo Ouro, Bruño Fraga, Unai Lopez-Novoa, and Thorsten Stoesser. Scalability of an Eulerian-Lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelisation. *Computers and Fluids*, 179:123–136, 2019.

[108] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a Million Processors. pages 20–30. 2009.

[109] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *ACM/IEEE SC 2000 Conference (SC'00)*, pages 12–12. IEEE, 2000.

[110] Ananta Tiwari, Allyson Cauble-Chantrenne, Adam Jundt, Joshua Peraza, Rainald Löhner, Joseph D. Baum, and Laura Carrington. Running large-scale CFD applications on Intel-KNL-based clusters. *International Journal for Numerical Methods in Fluids*, 86(11):699–716, apr 2018.

[111] Gabriele Jost, Haoqiang Jin, Moffett Field, Ferhat F Hatay, Sun Microsystems, High Performance, and Technical Computing. Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster 1. *Structure*, (September), 2003.

[112] E. Yilmaz, R. U. Payli, H. U. Akay, and A. Ecer. Hybrid parallelism for CFD simulations: Combining MPI with openMP. *Lecture Notes in Computational Science and Engineering*, 67 LNCSE:401–408, 2009.

[113] W.J. Bolosky and M.L. Scott. False sharing and its effect on shared memory performance. *USENIX Experiences with Distributed and Multiprocessor Systems, SEDMS 1993*, 1801(14520052):1–15, 1993.

[114] Josep Torrellas, H.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, jun 1994.

[115] J Mark Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[116] J. Mark Bull, Fiona Reid, and Nicola McDonnell. A Microbenchmark Suite for OpenMP Tasks. pages 271–274. 2012.

[117] Michael K. Bane and Graham D. Riley. Extended Overhead Analysis for OpenMP. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2400, pages 162–166. 2002.

[118] Abel Castellanos, Andreu Moreno, Joan Sorribes, and Tomas Margalef. Performance Model for Master/Worker Hybrid Applications on Multicore Clusters. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 210–217. IEEE, nov 2013.

[119] Gerald Schubert, Holger Fehske, Georg Hager, and Gerhard Wellein. Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. *Parallel Processing Letters*, 21(3):339–358, 2011.

[120] Michael Lange, Gerard Gorman, Michele Weiland, Lawrence Mitchell, and James Southern. Achieving Efficient Strong Scaling with PETSc using Hybrid MPI/OpenMP Optimisation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7905 LNCS:97–108, mar 2013.

[121] Nikolaos Drosinos and Nectarios Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, volume 18, pages 193–202, 2004.

[122] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, jul 2002.

[123] Géraud Krawezik and Franck Cappello. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. Technical report, 2003.

[124] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[125] https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-linux/top/additional-supported-features/multiple-endpoints-support/mpi-thread-split-programming-model.html.

[126] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, feb 2009.

[127] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing - ICS '05*, page 253, New York, New York, USA, 2005. ACM Press.

[128] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, sep 2007.

[129] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang H Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5568 LNCS, pages 42–52. 2009.

[130] Hayder Al-Khalissi, Syed Abbas Ali Shah, and Mladen Berekovic. An Efficient Barrier Implementation for OpenMP-Like Parallelism on the Intel SCC. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 76–83. IEEE, feb 2014.

[131] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, sep 1993.

[132] J.E. Burns and N.A. Lynch. Bounds on Shared Memory for Mutual Exclusion. *Information and Computation*, 107(2):171–184, dec 1993.

[133] John M Mellor-Crummey and Michael L Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[134] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, 26(7):106–113, jul 1991.

[135] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015*, volume 2015-Janua, pages 215–226, New York, New York, USA, jan 2015. ACM Press.

[136] https://en.cppreference.com/w/c/language/atomic.

[137] Georgios Georgopoulos. Memory Consistency Models of Modern CPUs.

[138] S.V. Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[139] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Jeff Hammond, Satoshi Matsuoka, and Pavan Balaji. Locking Aspects in Multithreaded MPI Implementations. *Implementations ACM Trans. Parallel Comput. X, X, Article*, X(X):1–27, 2016.

[140] Hoang-Vu Dang, Sangmin Seo, Abdelhalim Amer, and Pavan Balaji. Advanced Thread Synchronization for Multithreaded MPI Implementations. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 314–324. IEEE, may 2017.

[141] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, volume 15-20-Nove, pages 1–12, New York, New York, USA, nov 2015. ACM Press.

[142] David Buettner, Jean Thomas Acquaviva, and Josef Weidendorfer. Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 208–215, 2013.

[143] James Dinan, Ryan E. Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling communication concurrency through flexible MPI endpoints. *The International Journal of High Performance Computing Applications*, 28(4):390–405, nov 2014.

[144] Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar. Enabling Efficient Multithreaded MPI Communication through a Library-Based Implementation of MPI Endpoints. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498. IEEE, nov 2014.

[145] Ron Brightwell, Sue Goudy, and Keith Underwood. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. In *2005 International Conference on Parallel Processing (ICPP'05)*, volume 2005, pages 175–183. IEEE, 2005.

[146] Philip Claude Delhaye. Caplan. Four-dimensional anisotropic mesh adaptation for spacetime numerical simulations. 2019.