SAPIENZA
UNIVERSITÀ DI ROMA

Master di II livello in Calcolo Scientifico
A.A. 2016-2017

High Performance Computing for the Efficient Solution of PDEs on arbitrary domains

Author
Simone Cammarasana

Internal Supervisor
Prof. Elisabetta Carlini

External Supervisor
Dr. Andrea Clematis (CNR-IMATI)
Dr. Antonella Galizia (CNR-IMATI)
Dr. Giuseppe Patanè (CNR-IMATI)

Contents

Abstract

This thesis is the output of a six-months stage performed at CNR-IMATI, in Genova. The goal of this thesis is to evaluate direct and iterative solvers on large sparse linear systems by exploiting available parallel software, in order to study and compare their performances in terms of efficiency and accuracy.

The focus of this study has been on general purpose methods and their high performance computing (HPC) implementations, which will be tested on specific classes of PDEs.

Main scientific outputs have been the presentation and acceptance by CINECA of an ISCRA-C project for performing all the tests on MARCONI cluster, the realization of a technical report [40] for IMATI-CNR and the submission of the technical report to an international journal of parallel computing.

1 Introduction

Partial Differential Equations (PDEs) are used to describe several problems relevant in many fields, such as engineering, physics, biology [12], finance, social science [13] and many others. In all these problems, the input data are a 2D or 3D domain and the PDE to be solved, while the expected output is the solution on the domain. Since most of the PDEs cannot be solved in their analytic form, these are discretized on a finite domain (with regular or irregular grids) and with approximate techniques (such as Finite Element or Finite Differences); the corresponding discrete formulation tipically reduces to the solution of a sparse linear system. Figures 1 and 2 show the workflow and links among these steps.

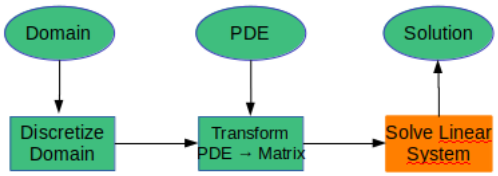

Figure 1: PDE soution workflow.

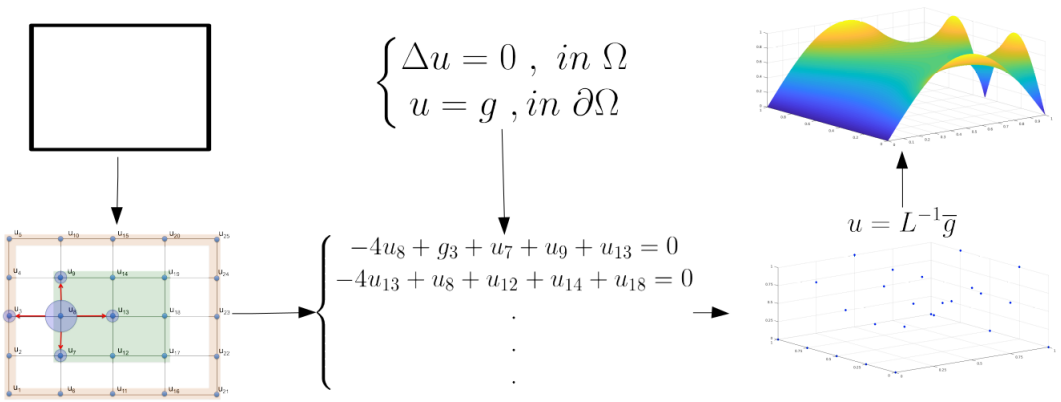

Figure 2: PDE solution workflow - description.

The efficient solution of a large sparse linear system becomes relevant; in fact, several algebraic scientific libraries that exploit HPC resources have been developed since 70's to nowadays, following two main approaches: direct and iterative methods.

Related work
In literature, many studies compare the performances for the solution of linear systems with iterative and direct methods on parallel resources.

In [15] and [43], preconditioned Krylov solvers implemented on GPUs are studied, performing the analysis on a large set of matrices and comparing the performances in terms of execution time and efficiency.

In [45] and [44], the performances of sparse direct solvers (Pastix and SuperLU respectively) have been compared on an extensive set of test probelms, taken from a range of practical applications.

In [41], a new benchmark for the computation of the solution to the Poisson equation on a regular 3D grid with a *High-Performance Conjugate Gradient* (HPCG) and a symmetric Gauss-Seidel pre-conditioner is analysed in terms of computation and data access patterns. The main steps of the HPCG include the conjugate gradient iterations' setup and execu-

tion, the multi-grid and coarse grid solvers, the validation and verification components, an optional optimization step.

According to the comparison [42] of the two packages Amesos2 and Belos of the Trilinos Project, Amesos2 provides a common interface for sparse matrix factorization, enables extended and mixed precision algorithms. Belos includes several iterative methods for the solution of sparse linear systems and least-squares problems, and is oriented towards higher-level problems' solution. Decoupling the algorithms from the implementation of the underlying linear algebra improves efficient the portability of the libraries to a different hardware, supports the independence of the HPC libraries from the linear algebra library, the maximisation of code reuse, the development of applications and architecture-aware algorithms (including mixed precision methods). Since this previous work focuses on software packaging and integration aspects, it further motivates our analysis on numerical solvers of linear systems.

Goal of the thesis

The goal of this thesis is to analyze the performances of these methods and their parallel implementations on Marconi HPC resource, varying input conditions and comparing different metrics and factors, and giving a clear interpretation of results. In particular, a set of input conditions (2D and 3D domain, regular and irregular grids, sparsity pattern of the coefficient matrix) has been defined; after that, iterative and direct methods have been analyzed and compared in terms of scalability, efficiency, solution accuracy and impact of multiple right-hand side terms. Finally, results have been analyzed with detail on the single operations, in order to give an interpretaion of scalability results.

Contribution with respect to previous work

While the benchmark in [41] is focused on the definition of a computational kernel to drive future systems' design, we are interested in the combination of the computational aspects of solvers for large sparse linear systems (e.g., arising from the discretisation of elliptic PDEs) with guarantees on the convergence and accuracy of the underlying numerical solvers. We focus our discussion on the Laplace equation on both regular and irregular 2D/3D grids; however, our approach and analysis are enough general to be applied to the solution of sparse linear systems associated with a generic PDE discretization.

The expected scientific output is a fair comparison of these different methods on the Marconi resource, with a focus on the efficiency performances but also on the interpretation of these results, in order to understand where and why a certain method is preferable.

Thesis structure

This thesis has been organized in the following sections:

- Chapter 2 gives a brief description of PDE properties and discretization methods;
- Chapter 3 describes linear systems, main solver methods (iterative and direct) and their properties;
- Chapter 4 describes HPC resources (software and hardware) and metrics used for tests;
- Chapter 5 describes the PDE equation chosen for tests and the input conditions (domain, grid, discretization techniques);
- Chapter 6 shows results of the analysis performed;
- Chapter 7 shows scientific tools used for the visualization of the solution in both 2D and 3D domain with regular and irregular grids.

2 Partial Differential Equations

A PDE is the definition of one or more equations in the form of $F(u, u_x, u_{xx}...) = 0$ where

- $u = u(x_1, ..., x_n)$ is the unknown function of n variables;
- $x = (x_1, x_2, ..., x_n)$ are the independent variables;
- u_x, u_{xx}, u_{xxx} are partial derivatives of the function u;

The order of the PDE is the maximum derivative order appearing in the equation; furthermore PDE can be classified as:

- Linear: F can be expressed as a linear combination of u and its derivative terms;
- Quasi Linear: F can be expressed as a combination of u and its derivative terms, where the coefficients are dependent from u;
- Non linear: F is non linear with respect to its derivative terms.

Examples of PDE

Some of the classical examples of PDEs, in various fields, are:

- *Transport equation*: defines the flow of a particle through a path: $u_t + v(x,t) \cdot \nabla u = 0$ where u is the function of concentration of mass transfer and v is the speed the quantity is moving;
- *Heat equation*: describes the temperature diffusion on a surface-volume: $u_t - D \cdot \Delta u = 0$ where u is the heat function and D defines the thermal properties of the surface;
- *Wave equation*: describes the propagation of a wave on a surface-volume $u_{tt} c^2 \cdot \Delta u = 0$ where u is the wave amplitude function and c is the propagation speed.

2.1 Numerical method for PDEs

For the discretization of the input domain (2D surface or 3D volume), we distinguishes:

- *regular grids* which allows a simpler generation of the grid and a simpler discretization of the PDE (Figure 3, top);
- *irregular grid* (e.g., triangular and tetrahedral meshes) which can be adapted to the irregularity of the domain and allow a different sampling density on different domain regions (Figure 3, bottom).

The discretization of a PDE on a regular or irregular grid tipically reduces to the solution of a sparse linear system, whose coefficient matrix discretizes the differential operator of the PDE and the right-hand side term is defined by the initial/boundary conditions. Main approaches include finite differente, element, volume and spectral methods [37].

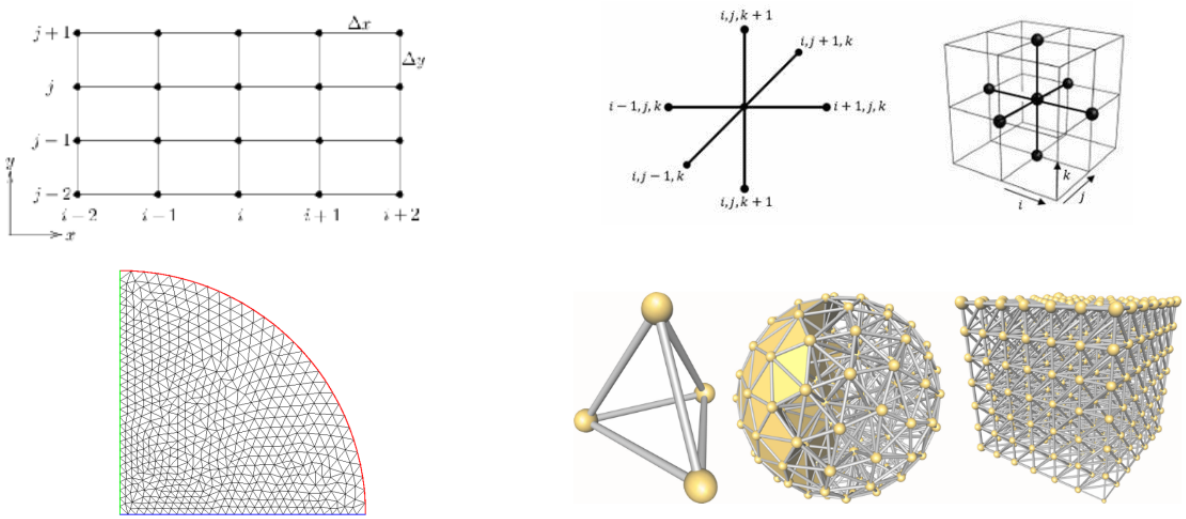

Figure 3: Grid examples for regular/non regular 2D and 3D domains.

2.2 Sparse matrices

A $m \times n$ matrix with k non-zero elements, $k \ll m \times n$ and $n \geq m$, is considered as sparse if it has $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$ non-zero elements. Sparse matrices store only non-zero values and their position; this kind of data-structure allows us to increase the grid size without affecting the amount of stored data.

Among the main methods for the storage of sparse matrices, we mention the *compressed row storage* and the *compressed column storage*, both of which use a three vectors structure. For example, CRS vectors are: vector of values, vector of non-zeros column position and vector of quantity of non-zeros per row.

3 Linear System

Solvers of linear systems can be classified as: (i) *direct methods*, which calculate the exact solution through a decomposition of the coefficient matrix into one or more (triangular, orthogonal) matrices and then calculate the solution by solving the factored linear system [3]; (ii) *iterative methods*, which approximate the solution with a set of iterations that converge to the solution from an initial guess. These methods stop at a certain iteration according to a break criteria; (e.g., maximum number of iterations, convergence/divergence of the solution etc) [2] [7] [8].

3.1 Iterative methods

Iterative methods find an approximate solution of a linear system ,starting from an initial guess and converging to exact solution. Although several algorithms exist, they are all based on the same general theory and techniques.

Projection method
Given a linear system $Ax = b$ where A is a $n \times n$ matrix, projection techniques extract an approximate solution from a subspace of $\mathcal{R}^n$, denominate $\mathcal{K}$. This *search subspace* is m dimensional and, in order to extract the approximate solution, m constraints must be imposed. Tipically, m orthogonality conditions are defined between the residual vector $b-Ax$ and m linearly independent vectors; this defines another subspace $\mathcal{L}$ of dimension m, called *subspace of constraints*.

In analytic terms, a projection technique onto the subspace $\mathcal{K}$ and orthogonal to $\mathcal{L}$ is a process which finds an approximate solution $\widetilde{x}$ by imposing $\widetilde{x} \in \mathcal{K}$ and the new residual vector $b - A\widetilde{x} \perp \mathcal{L}$. This can be expressed as:

$$\begin{aligned} \widetilde{x} &= x_0 + \delta,\ \delta \in \mathcal{K} \\ (r_0 - A\delta, \omega) &= 0,\ \forall \omega \in \mathcal{L} \end{aligned} \tag{1}$$

where $r_0 = b - Ax - 0$ and $\delta = \widetilde{x} - x_0$.

Main iterative methods differ from $\mathcal{K}$ and $\mathcal{L}$ choice.

Krylov subspace
There exists two classes of Krylov methods, on the basis of $\mathcal{K}$ subspace definition:

In the first class, the Krylov subspace of m dimension is $\mathcal{K}_m(A, v) = span(v, Av, A^2v, ..., A^{m-1}v)$; the inverse of the coefficient matrix is approximated with a (m-1)-degree polynomial: $A^{-1}b \approx \widetilde{x} = x_0 + q_{m-1}(A)r_0$. Most algorithms are based on this method, whit the choice of $\mathcal{L}_m = \mathcal{K}_m$ or $\mathcal{L}_m = A\mathcal{K}_m$.

In the second class, two biorthogonal bases are built for the two subspaces

$$\mathcal{K}_m(A, v) = span(v, Av, A^2v, ..., A^{m-1}v) \tag{2}$$

$$\mathcal{K}_m(A^\top, \omega) = span(\omega, A^\top\omega, (A^\top)^2\omega, ..., (A^\top)^{m-1}\omega) \tag{3}$$

Algoritmhs based on second class are projection processes onto Equation 2 orthogonally to Equation 3.

Two algorithms exist for computing orthonormal bases: Arnoldi's method (for first class) and Lanczos method (for second class).

Arnoldi iteration
Arnoldi's procedure is an algorithm for building an orthogonal basis of the Krylov subspace $\mathcal{K}_m$, on the basis of the Modified Gram-Schmidt algorithm:

1. *Choose* v_1 *of norm* 1
2. *For* $j = 1 : m$
3. $\quad \omega_j = Av_j$
4. $\quad$ *For* $i = 1 : j$
5. $\quad\quad h_{ij} = (w_j, v_i)$
6. $\quad\quad w_j = w_j - h_{ij}v_i$
7. $\quad$ *endFor*
8. $\quad$ *if* $(h_{j+1,i} = \|w_j\|_2) == 0$ *Stop*
9. $\quad v_{j+1} = \frac{w_j}{h_{j+1,i}}$
10. *endFor*

(4)

On the basis of this iteration, most of the iterative methods can be derived; for example, given $\beta = \|r_0\|_2$ and $v_1 = r_0 \ / \ \beta$, the approximate solution can be calculated as $y_m = H_m^{-1}(\beta e_1)$ and $x_m = x_0 + V_m y_m$, where H_m is the Hessenberg matrix.

The method just described is called the Full Orthogonalization Method (FOM); some variations of this approach lead to most known methods such as GMRES, Conjugate Gradient etc.

Lanczos iteration

Lanczos Iteration builds a pair of biorhogonal bases, basing on following procedure:

1. *Choose* v_1 *and* $\omega_1 :\ (v_1, \omega_1) = 1$
2. *Set* $\beta_1 = \delta_1 = 0,\ \omega_0 = v_0 = 0$
3. *For* $j = 1 : m$
4. $\quad \alpha_j = (Av_j, w_j)$
5. $\quad v_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6. $\quad \omega_{j+1} = A^\top \omega_j - \alpha_j \omega_j - \delta_j \omega_{j-1}$
7. $\quad If\ (\delta_{j+1} = \sqrt{|(v_{j+1}, \omega_{j+1})|}) == 0$ *Stop*
8. $\quad \beta_{j+1} = \frac{(v_{j+1}, \omega_{j+1})}{\delta_{j+1}}$
9. $\quad \omega_{j+1} = \frac{\omega_{j+1}}{\beta_{j+1}}$
10. $\quad v_{j+1} = \frac{v_{j+1}}{\delta_{j+1}}$
11. *endFor*

(5)

Then, solution can be calculated as $y_m = T_m^{-1}(\beta e_1)$ and $x_m = x_0 + V_m y_m$ where: $T_m = tridia(\delta_2..\delta_m, \alpha_1..\alpha_m, \beta_2..\beta_m)$ is the Hessenberg matrix.

From this approach, several methods can be derived, such as BCG and QMR.

Transpose-free variants

Each step of the Biconjugate Gradient algorithm and QMR requires a matrix-by-vector product with both A and $A^\top$. However, the vectors p_i or w_j generated with $A^\top$ do not contribute directly to the solution; they are used only to obtain the scalars needed in the algorithm (e.g., α_j and β_j for BCG).

Some technical tricks allow us to bypass the use of the transpose of A for computing coefficients; with this approach, BICGSTAB and TFQMR are derived.

3.1.1 Iterative methods used for case study

For our analysis, we have considered several iterative methods (Table 1) working with sparse, not symmetric ($A \neq A^\top$) and positive-definite ($x^\top Ax > 0,\ \forall x \neq 0$) matrices and that allow general approach to the solution of the linear system (e.g., GMRES, BICG, GCR etc.).

The efficiency of a solver mainly depends on the number of operations that it performs for each iteration and on the number of iterations that it needs to converge to the solution under

Method	Acronym	Sparse	Not Symm	Not Pos.Def	Not $A^\top$
Generalized Minimal Residual Method	GMRES	x	x	x	x
Biconjugate Gradient Stabiliezed	BICGS	x	x	x	x
Conjugate Gradient	CG	x	-	-	-
Generalized Conjugate Residual	GCR	x	x	x	x
Improved Biconjugate Gradient Stabilized	IBICGS	x	x	x	x
Transpose Free Quasi Minimal Residual	TFQMR	x	x	x	x
Induced Dimension Reduction	IDR	x	x	x	x
Minimum Residual	MINRES	x	-	x	-
Quasi Minimal Residual	QMR	x	x	x	-
Conjugate Gradient Squared	CGS	x	x	x	x
Biconjugate Gradient	BICG	x	x	x	-

Table 1: Iterative methods comparison.

a certain error threshold. The computational costs of the main methods (not preconditioned) are:

$$\begin{aligned} &GMRES : matvec + 2 \cdot it \cdot vec + vec + it \cdot prod + norm \rightarrow 2m \cdot k_{row} + 4 \cdot m \cdot it + 4 \cdot m \\ &BICGSTAB : 2 \cdot matvec + 11 \cdot vec + 4 \cdot prod \rightarrow 4m \cdot k_{row} + 19 \cdot m \\ &GCR : 2 \cdot matvec + 2 \cdot prod + 1 \cdot it \cdot prod + 5 \cdot vec + 1 \cdot it \cdot vec \rightarrow 4m \cdot k_{row} + 9m + 3 \cdot it \cdot m \\ &TFQMR : matvec + 9 \cdot vec + prod + norm \rightarrow 2m \cdot k_{row} + 14 \cdot m \end{aligned} \tag{6}$$

where $matvec = 2m \cdot k_{row}$ is a *matrix-vector* product, $prod = 2m$ is a *scalar product*, *vec* is a *scalar-vector* or *vector-sum* operation, *norm* is a $3m$ operation, k_{row} are non-zeros per row, m are matrix rows, *it* is the iteration number.

Data distribution

On the parallel implementation, the iterative methods used in this case study distribute data on row blocks. Each process owns a portion of the coefficient matrix with a static assignment.

3.2 Preconditioners

Preconditioners are used to reduce the conditioning number of the coefficient matrix and, consequently, the rate of convergence of iterative solvers. Given a linear system $Ax = b$, applying a preconditioner M means to solve the new equation $AM^{-1}Mx = b$ as:

$$\begin{cases} AM^{-1}y = b \\ Mx = b \end{cases}$$

where $cond(AM^{-1}) << cond(A)$.

3.2.1 Preconditioners used on case study

In our analysis, we have compared the performances of the following preconditioners: (i) *Jacobi* ($M_{ij} = A_{ij}$ $i = j$, 0 *otherwise*); (ii) *Block Jacobi* (block diagonal matrix) [9]; (iii) *Multigrid Methods* [10].

Besides the computation of the matrix M, the use of preconditioners increases the number of operations performed per iteration; however, the number of iterations necessary to converge is highly reduced.

Factorization	LU	Cholesky	QR
Sparse Matrix	x	x	x
Not Symm	x	-	x
Not Pos.Def	x	-	x
Not $A^\top$	x	x	x

Table 2: Factorization methods comparison.

3.3 Direct methods

Direct methods allow us to compute the exact solution of the input linear system and generally involve the:

- *graph partitioning*: the nodes of a graph are reordered in p roughly equal parts, such that the number of edges connecting the nodes in different parts is minimized [4]. Several techniques exist: spectral methods, algebraic methods, multilevel graph. The goal is to optimize the sparsity pattern and to reduce the fill-in after the factorization;
- *matrix permutation*: matrix rows and columns are permuted, according to graph partitioning output. When performing a column permutation, the solution also is permuted;
- *symbolic analysis*: it is a symbolic factorization that does not perform algebraic computation; it allows to determine the non-zero structure of the factor matrices (i.e., triangular matrices used for the solution of Equation (7)) in terms of sparsity pattern and stored memory;
- *factorization*: it computes the factor matrices; there exist several factorization techniques (LU, QR, Cholesky etc.), as shown in Table 2.

In our discussion we focus on the LU decomposition, described in Section 3.3.1.

3.3.1 LU decomposition

This section is based on [3], [25] and [27], where more detailed analyses can be found.

Given a linear system $Ax = b$, an LU factorization refers to the factorization of the coefficient matrix A into two factors, a lower triangular matrix L and an upper triangular matrix U, such that:

$$Ax = LUx = b. \tag{7}$$

This lead to solve two triangular linear systems, $y = L^{-1}x$ and $x = U^{-1}y$, with forward and backward substitution.

LU factors

LU decomposition can be viewed as the matrix form of the Gaussian elimination. Given $A = IA$, Gaussian elimination can be applied on the second matrix in order to get U:

$$A(j,:) = A(j,:) - m_{i,j} \cdot A(i,:),\ i = 1:n\ \ j = (i+1):n \tag{8}$$

where $m_{i,j} = A(j,i)/A(i,i)$ are the multipliers. In order to keep the original matrix product $A = IA$, I has to be updated as

$$I(j,:) = I(j,:) - m_{i,j} \cdot I(i,:),\ i = 1:n\ \ j = (i+1):n$$

and the matrix obtained is the L factor.

In case that one of the multipliers zero, pivoting has to be applied.

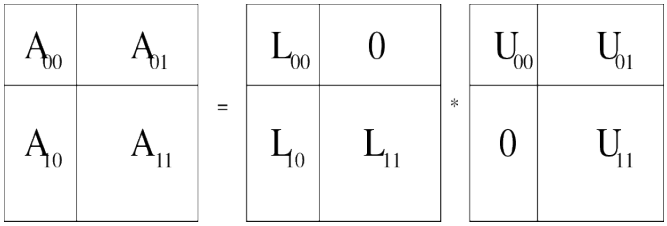

Figure 4: Block LU.

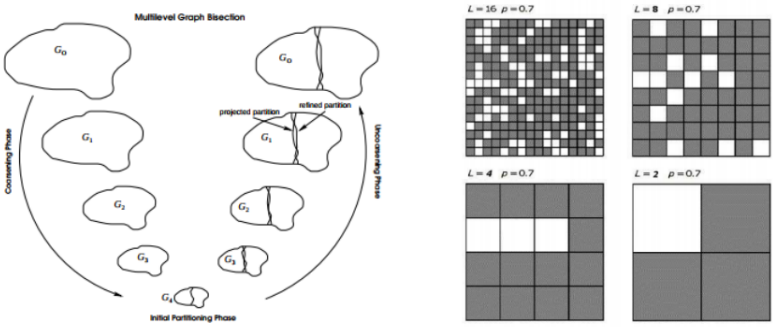

Figure 5: Multirid scheme.

Triangular solver

L and U factors are used for solving triangular systems. If right-hand side is a vector ($Lx = y$), solution can be found with TRSV (TRiangular Solve Vector) as $x_i = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j)/a_{ii}$ for $i = 1 : n$.

If right-hand side is a matrix ($LX = Y$), solution can be found with TRSM (TRiangular Solve Matrix) where basically several TRSV are applied: $y_j = TRSV(L, x_j)$ for each column j.

Block LU

It is possible to organize Gaussian elimination so that matrix multiplication becomes the dominant operation; given $A = LU$ and a block parameter r, decomposition can be organized as per Figure 4, where A_{00} is a $r \times r$ matrix, A_{11} is $(n - r) \times (n - r)$, A_{01} is $r \times (n - 3)$ and A_{10} is $(n - r) \times r$.

Thus, four systems have to be solved:

- $A_{00} = L_{00}U_{00}$ is a LU decomposition of a $r \times r$ matrix
- $A_{01} = L_{00}U_{01}$ is a TRSM
- $A_{10} = L_{10}U_{00}$ is a TRSM
- $A_{11} = L_{10}U_{01} + L_{11}U_{11}$, where $A' = L_{10}U_{01}$ is a matrix-matrix product and thus $A - A' = L_{11}U_{11}$ can be factorized recursively, until full LU decomposition is done.

LU on sparse matrices

LU decomposition on sparse matrices introduces some critical aspects to be managed; if an entry is zero in the original matrix A, the corresponding entry can be non-zero in the factors; this phenomenon is known as *fill-in*. In order to limit the amount of fill-in, the order in which the variables are eliminated in Gaussian elimination is critical; several techniques

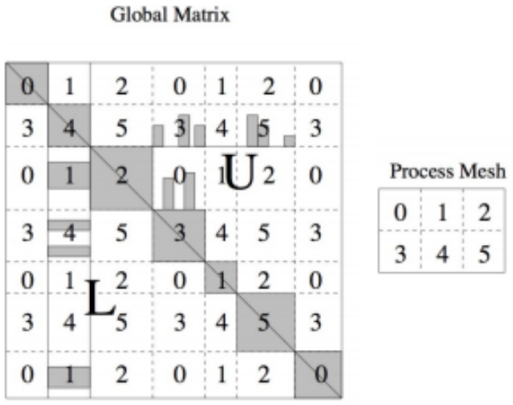

Figure 6: LU block data distribution.

of re-ordering and permutation can be applied, such as Approximate Minimum Degree [26] and Multilevel graph partitioning schemes [4]. Multilevel graph partitioning reduces the size of the graph (coarsen phase) by collapsing nodes and edges, sections the smaller graph (partition phase) and then refines the partition up to the original graph (uncoarsen phase); Figure 5 shows the logical steps of multilevel preconditioning (left) and an example of coarsed grid (right), where nodes are collapsed into a simpler structure. The goal of multilevel graph technique is to partition the nodes of a graph in p roughly equal parts, such that the number of edges connecting the nodes in different parts is minimized.

Furthermore, due to fill-in phenomenon, non-zero structure of factors matrices is unknown a priori; thus, symbolic factorization is performed in order to define the sparsity pattern of the LU decomposition with reduced computing complexity (with respect to the numerical factorization). The basic idea of symbolic factorization is that, given a M sparse matrix:

$$Struct(M_{i*}) := k < i | m_{ik} \neq 0$$

$$Struct(M_{*j}) := k > j | m_{kj} \neq 0$$

and the function:

$$p(j) := \begin{cases} min\{i \in Struct(L_{*j})\}, \ if \ Struct(L_{*j}) \neq 0 \\ j, \ otherwise \end{cases}$$

where $p(j)$ is the row index of the first off-diagonal non-zero in j column, if present.
It can be shown that the structure of column j of L can be characterized as:

$$Struct(L_{*j}) := Struct(A_{*j}) \cup \left(\bigcup_{i<j} \{Struct(L_{*i}) | p(i) = j\} \right) - \{j\}$$

that is, the structure of column j of L is given by the structure of the lower triangular portion of column j of A, together with the structure of each column of L whose first off-diagonal non-zero is in row j. In this way, it is possible to forecast the non-zero entries of each column of L.

Finally, when performing the LU decomposition, only non-zero elements have to be considered in order to exploit the sparsity of the coefficient matrix A. Several techiques exist, which can mainly be classified in left-looking (used by SuperLU), right-looking and multifrontal (used by MUMPS), as described in [27] and [28].

Left-looking starts from Equation 8 as a triple-nested loop $a_{ij} = a_{ij} - (a_{ik}a_{kj})/a_{kk}$; taking j in the outer loop, successive columns of L are computed one by one and the inner

loops computes a matrix-vector product; the column of the inner loop is affected by the columns on the left in the matrix.

High level algorithm can be described as:

$$\begin{array}{l} For\ j = 1 : n \\ \quad For\ k \in Struct(L_{i*}) \\ \quad\quad cmod(j,k) \\ \quad cdiv(j) \end{array} \tag{9}$$

where $cmod(j,k)$ is the modification of column j by column k with $k < j$ and $cdiv(j)$ is the division of column j by a scalar.

Data distribution

On the parallel implementation, the direct methods used in case study distribute data on cyclic block matrices; data are assigned so that each process communicates only with its neighbours. Blocks can be of different size, on the basis of non-zero patterns; Figure 6 shows an example of block distribution among processes.

4 High Performance Computing

Parallelism allows us to distribute data and computation, exceeding limits of memory and reducing computation time. In order to take advantage of parallelism, several parallel scientific libraries offer basic routines of linear algebra; in this way, the user can focus on analysis topics, assigning low level operations to the libraries.

As shown in Figure 7, scientific libraries for the solution of linear systems can be structured on three logical levels, based on the user interaction and the level of operations performed:

- user operates at high level, selecting a solver and passing the coefficient matrix and the right-hand side array to the library;
- scientific libraries implement the algorithm and sets the data indices to be distributed;
- low level libraries perform basic linear algebra operation (BLAS) and data communication (MPI).

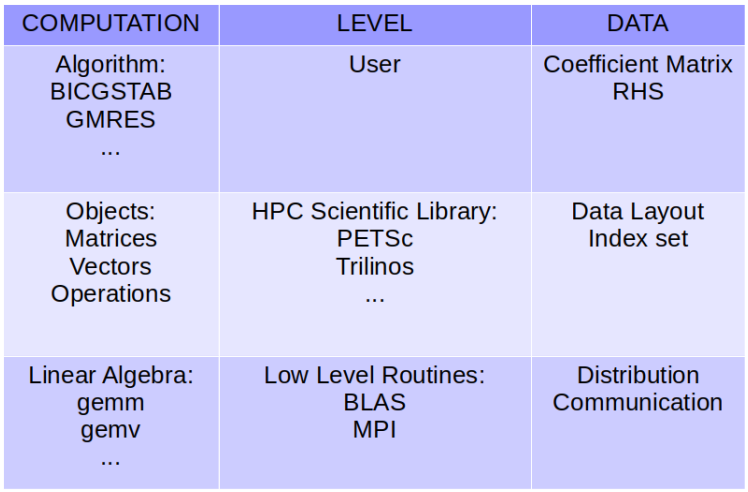

Figure 7: Logical vision of parallel scientific libraries.

This chapter is divided in two parts: Section 4.1 gives a short introduction to parallel computation and scientific libraries while Section 4.2 describes software, hardware and metrics used for the analysis of the case study.

4.1 Parallel scientific libraries for algebraic operations

Numerical linear algebra is often the heart of many engineering and computational science problems; the scientific libraries provide the *building blocks* for the efficient implementation of more complex algorithms, such as the solution of linear systems of equations, linear least squares problems and eigenvalue problems. In order to exploit hardware improvements and parallel resources, the implementation of parallel software libraries have been evolved considering also the specific features of the architectures established during the years. For example, important aspects are a proper exploitation of the memory hierarchy and the arithmetic density, which can have a significant impact on the execution time [33].

In the following paragraphs a short introduction to parallel computing, memory hierarchy, basic linear algebra libraries and an explanation of parallel matrix computation will be described.

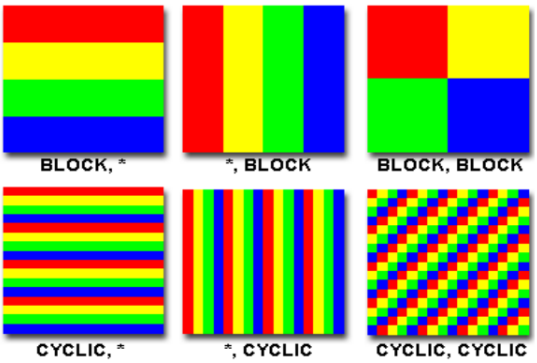

Figure 8: Data Distribution.

Parallel computation

Parallel computation is a type of computation in which many calculations or the execution of processes are carried out concurrently; large problems can be divided into smaller ones, which can then be solved at the same time. Parallelism can involve only data distribution (Single Instruction Multiple Data) or data and tasks (Multiple Instruction Multiple Data); in this paragraph, data distribution will be described.

Two main paradigms are considered: shared and distributed memory; these two paradigms follow the architectural evolution of hardware resources. A hybrid approach consists in distributing computation among several processes and then each of these exploit shared memory computation.

In distributed memory systems, each processor has a local memory and executes its own program; the program can alter values in the executing processor's local memory and can send data in the form of messages to the other processors in the network. The interconnection of the processors defines the network topology (ring, 2D mesh, torus, tree etc.) In shared memory systems, communication among processors is achieved by reading and writing to global variables that reside in the global memory [3]. Hybrid paradigm uses distributed memory approach among nodes and shared memory inside each node.

Distributed memory paradigm allows high scalability and portability, but the explicit communication can create load balancing problems; shared memory paradigm allows implicit communication and dynamic load balance, but it can work only on shared memory resources and data access consistency has to be managed [29]. Furthermore, while distributed paradigm can be used in shared memory resource, it is not possible the opposite.

The choice of shared, distributed or hybrid paradigm depends on the problem to be solved and on the available resources (hardware and software) and it affects performances. In tested algorithms, the parallelism is applied at data level; this means that the data are splitted among processes. The way data are distributed affects communications and load balance; there exist several ways of distributing data, as reported in Figure 8.

Scientific libraries used in this work use row block distribution (PETSc) and cyclic block distribution (SuperLU).

Memory hierarchy

The memory hierarchy divides computer storage into a hierarchy based on the response time. The main advantage of this memory configuration is that those data needed for computation can be allocated into contigous memory location near the processing unit, in order to access them very quickly. Memory hierarchy affects performances in computer architectural design, algorithm predictions and lower level programming constructs; in most computers, performances of algorithms can be dominated by the amount of memory traffic, rather than the number of floating-point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on the data; for this reason

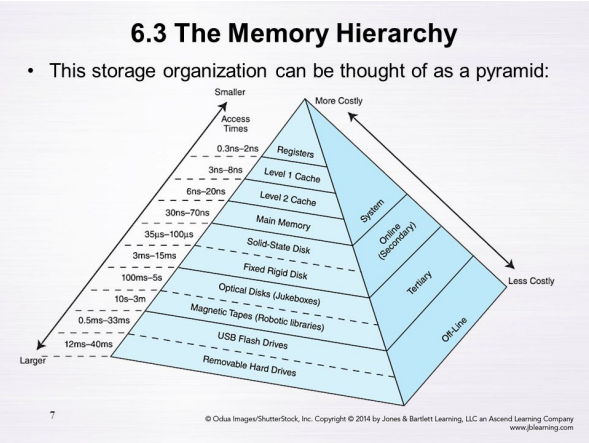

Figure 9: Memory Hierarchy Pyramid.

an important parameter is represented by the arithmetic density that may provide a metric to data access efficiency of the algorithm. Figure 9 shows the different levels of memory:

access time decreases as we go closer to CPU register, i.e. where the data is needed. Quicker memories are more expensive; for example, fixed rigid disk has an approximate cost of 0.02-2\$/GB, solid-state disk has 4-12\$/GB, main memory has 20-75\$/GB [35].

Typical memory sizes are summarized on Table 3.

Memory	Size
CPU register	few thousands byte
L1 cache	128 KB
L2 cache	1 MB
Main memory	few GB
Solid-state disk	several GB
Fixed rigid disk	few TB

Table 3: Memory sizes.

Basic Linear Algebra Subprograms

BLAS is a very successful example of software library and it is used in a wide range of softwares. It is an aid to clarity, portability, modularity, and maintenance of software; it has become a *de facto* standard for elementary vector and matrix operations.

BLAS identifies the frequently occurring operations of linear algebra, i.e. the *building blocks*, and specifies a standard interface for them, thus promoting software modularity. To improve performances, the optimization of BLAS subroutines can be done without modifying the higher-level code that may employ them. Other peculiar features of BLAS code are robustness, portability and readability. It is possible to identify three Levels of BLAS, depending on the software organization; this structure is aimed to obtain a better exploitation of the underlying architecture and to improve performances: BLAS1 (scalar operations between two vectors), BLAS2 (matrix-vector operations) and BLAS3 (matrix-matrix operations). [31].

Sparse-BLAS are the counterpart of BLAS, providing computational routines for unstructured sparse matrices. Sparse BLAS also contains the three levels of operations as in the dense case:

- Level 1: sparse dot product, vector update;

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} & \cdots & \mathbf{B}_{1r} \\ \mathbf{B}_{21} & \mathbf{B}_{22} & \cdots & \mathbf{B}_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{s1} & \mathbf{B}_{s2} & \cdots & \mathbf{B}_{sr} \end{bmatrix},$$

Figure 10: Block Matrix.

$$\begin{aligned} C \;=\; AB = & \left(\begin{array}{c} A_{00} \\ \hline A_{10} \\ \hline \vdots \\ \hline A_{(N-1)0} \end{array}\right) \left(B_{00} \mid B_{01} \mid \cdots \mid B_{0(N-1)} \right) \\ + & \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline \vdots \\ \hline A_{(N-1)1} \end{array}\right) \left(B_{10} \mid B_{11} \mid \cdots \mid B_{1(N-1)} \right) + \cdots \\ + & \left(\begin{array}{c} A_{0(N-1)} \\ \hline A_{1(N-1)} \\ \hline \vdots \\ \hline A_{(N-1)(N-1)} \end{array}\right) \left(B_{(N-1)0} \mid B_{(N-1)1} \mid \cdots \mid B_{(N-1)(N-1)} \right) \end{aligned}$$

Figure 11: Matrix-Matrix product.

- Level 2: sparse matrix-vector multiply and triangular solver;
- Level 3: sparse matrix-dense matrix multiply and triangular solver with multiple right-hand sides.

Parallel matrix computation
Matrix operations are central in both direct and iterative methods and their efficient parallel implementation is relevant in order to achieve good performances.

For parallel matrix-matrix operation, block-cyclic data decomposition can be assumed, as showed in Figure 10, where B_{ij} is a $n_r \times n_c$ submatrix. Matrix is distributed to nodes so that B_{ij} is assigned to process $P_{i \bmod r,\ j \bmod c}$.

Thus, assuming both A and B matrices distributed as per Figure 10, matrix-matrix product $C = AB$ can be performed as described in Figure 11, where each process can perform its portion of computation.

The parallel computation can be implemented, as shown with a high-level description in Figure 12; communications have to be performed in order to exchange data at each iteration among processes and to collect local results into global C matrix (if needed).

Scientific libraries for parallel computation
In conclusion, the goal of parallel scientific libraries for linear algebra is to provide efficient basic algebra routines, dealing with data distribution and communications in parallel environment.

They offer to users a high level use of low level functionality (such as BLAS, MPI), in order to get best performances without "tedious programming tasks"; parallelism is offered in transparent way and the user does not need to call routines for data distribution or communications because everything is managed by the scientific library.

4.2 HPC on case study

For the case study presented in this report, scientific libraries, hardware resources and metrics used for performances evaluation are presented.

in parallel $C = 0$
for $i = 0, N-1$
broadcast A_{*i} **within rows**
broadcast B_{i*} **within columns**
in parallel

$$C = C + \left(\begin{array}{c} A_{0i} \\ \hline A_{1i)} \\ \hline \vdots \\ \hline A_{(N-1)i} \end{array} \right) \left(\begin{array}{c|c|c|c} B_{i0} & B_{i1} & \cdots & B_{i(N-1)} \end{array} \right)$$

endfor

Figure 12: Parallel Matrix-Matrix product.

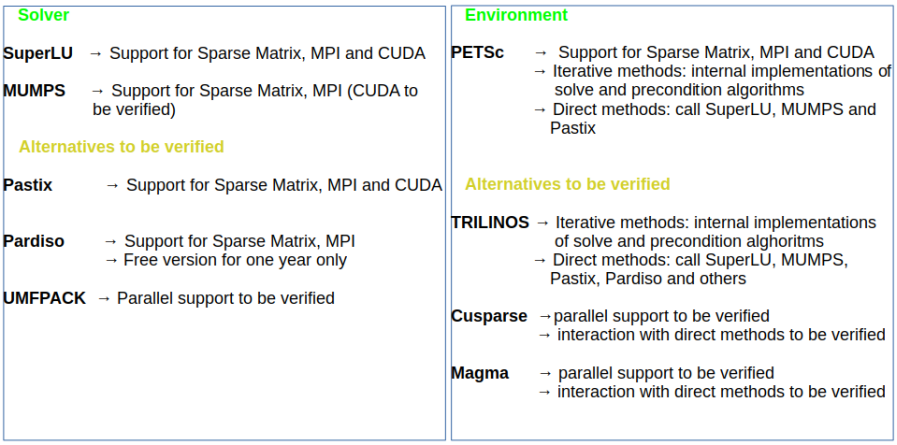

Figure 13: Scientific library selection.

Software

Several parallel scientific libraries and software have been selected to evaluate performances of solvers of sparse linear systems; these have been classified according to the following criteria (Figure 13):

- open-source library;
- support for matrices and algorithms, as described in Table 1;
- technical support and documentation;
- interaction with libraries for direct solvers;
- support for distributed and accelerated computation (MPI, CUDA etc.);

Libraries used for solving linear systems are:

- PETSc [18]: Linear Algebra library, with iterative methods and call routines to direct methods;
- SuperLU [6] and MUMPS [5]: direct methods libraries;

Furthermore, other libraries have been used in order to support the analysis:

- parMetis [21] and Hypre [20]: graph permutation libraries;
- ParaView [19]: 2D and 3D scientific visualization library;

- freeFem++ [23], dealii [24]: used for converting PDE into linear system;
- Gmsh [22]: creates mesh from CAD;

Metrics
In order to evaluate performances of scientific libraries and operations performed, several metrics have been considered:

- *Execution time (T)*: time of an operation;
- *FLOPS*: floating operations per second performed;
- *SpeedUp(n)*: $T(1)/T(n)$ where $T(1)$ is time for serial execution and $T(n)$ is time for parallel execution with n processes;
- *Efficiency*: calculated as $SpeedUp(n)/n$;
- *Granularity*: the ratio of computation time to communication time.

$$G = \frac{T_{computation}}{T_{communication}}$$

- *Efficiency per iteration*: efficiency calculated with an average time per iteration

$$E_{it} = \frac{\overline{T}(1)/\overline{T}(n)}{n},$$

where $\overline{T} = T/n_{iter}$

These metrics have been evaluated by taking into account several factors, such as the solution accuracy, the matrix size and the sparsity pattern.

Resources
Tests have been performed on CINECA cluster Marconi, based on Intel Xeon product family, on Broadwell partition; this is composed by:

- 2 x 18-cores Intel Xeon E5-2697 v4 (Broadwell) at 2.30 GHz;
- 1512 nodes, 36 cores/node. Total cores = 54432;
- 128 GB/node of RAM.

At June 2017, Marconi was in 14th position on Top500 rankning [16].

5 Case Study - Laplace Equation Discretization

This chapter introduces the Laplace equation and its discretization used for the case study. Section 5.1 introduces the Laplace operator while Section 5.2 describes the case study with the discretization of the Laplace equation on 2D/3D domains with regular/irregular grids and the corresponding coefficient matrices used for tests.

5.1 Laplace-Beltrami operator

This section is based on [32], where more detailed description can be found.

The Laplace-Beltrami operator is a differential operator given by the divergence of the gradient of a function on Euclidean space. Given a twice-differentiable real-valued function f, Laplace-Beltrami operator is expressed by:

$$-\Delta(f) = \ div(grad(f)) = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}$$

Given a domain Ω equipped with a Riemannian metric and the *scalar product*

$$\langle f, g \rangle_2 := \int_\Omega f(p)g(p)dp$$

defined on the space $\mathcal{L}^2(\Omega)$ of square integrable functions on Ω and the corresponding norm $\|\cdot\|_2$, the *Laplace-Beltrami* operator satisfies the following properties:

- *self-adjointness*: $\langle \Delta f, g \rangle_2 = \langle f, \Delta g \rangle_2$, $\forall f, g$;
- *positive semi-definiteness*: $\langle \Delta f, f \rangle_2 \geq 0$, $\forall f$. In particular, the Laplacian eigenvalues are positive;
- *null eigenvalue*: the smallest Laplacian eigenvalue is null and the corresponding eigenfunction ϕ, $\Delta\phi = 0$, is constant;
- *locality*: the value $\Delta f(p)$ does not depend on $f(q)$, for any couple of distinct points p, q;
- *linear precision*: if Ω is planar and f is linear, then $\Delta f = 0$.

Discrete Laplacian: Given a (triangular, polygonal, volumetric) mesh $\mathcal{M} := (\mathcal{P}, T)$, which discretizes a domain Ω, where $\mathcal{P} := \{p_i\}_{i=1}^n$ is the set of n vertices and T is the connectivity graph. On $\mathcal{M}$, a piecewise linear scalar function $f : \mathcal{M} \to \mathbb{R}$ is defined by linearly interpolating the values $f := (f(p_i))_{i=1}^n$ of f at the vertices using barycentric coordinates. For point sets, f is defined only at $\mathcal{P}$ and T is the k-nearest neighbor graph.

We represent the Laplace-Beltrami operator on surface and volume meshes in a unified way as $\tilde{L} := B^{-1}L$, where B is a sparse, symmetric, positive definite matrix (*mass matrix*) and L is sparse, symmetric, and positive semi-definite (*stiffness matrix*). We also assume that the entries of B are positive and that the sum of each row of L is null. In particular, we consider the B-scalar product $\langle f, g \rangle_B := f^\top Bg$ and the induced norm $\|f\|_B^2 := f^\top Bf$. Analogously to the continuous case, the Laplacian matrix satisfies the following properties.

- *self-adjointness*: $\tilde{L}$ is adjoint with respect to the B-scalar product; i.e.,

  $$\langle \tilde{L}f, g \rangle_B = \langle f, \tilde{L}g \rangle_B = f^\top Lg$$

  If $B := I$, then this property reduces to the symmetry of L;
- *positive semi-definiteness*: $\langle \tilde{L}f, f \rangle_B = f^\top Lf \geq 0$. In particular, the Laplacian eigenvalues are positive;
- *null eigenvalue*: by construction, we have that $\tilde{L}1 = 0$;
- *locality*: since the weight $w(i, j)$ is not null for each edge (i, j), the value $(\tilde{L}f)_i$ depends only on the f-values at p_i and its 1-star neighbor $\mathcal{N}(i) := \{j : (i, j) \text{ edge}\}$.

Harmonic functions The *harmonic function* $h : \Omega \to \mathbb{R}$ is the solution of the Laplace equation $\Delta h = 0$ with Dirichlet boundary conditions $h|_{\mathcal{S}} = h_0$, $\mathcal{S} \subset \Omega$. We recall that a harmonic function

- minimizes the *Dirichlet energy* $\mathcal{E}(h) := \int_{\mathcal{N}} \|\nabla h(p)\|_2^2 dp$;
- satisfies the *locality property*; i.e., if p and q are two distinct points, then $\Delta h(p)$ is not affected by the value of h at q;
- verifies $h(p) = (2\pi R)^{-1} \int_\Gamma h(s)ds = (\pi R^2)^{-1} \int_{\mathcal{B}} h(q)dq$, where $\mathcal{B} \subseteq \mathcal{N}$ is a disc of center p, radius R, and boundary Γ (*mean-value theorem*).

According to the *maximum principle*, a harmonic function has no local extrema other than at constrained vertices. In the case that all constrained minima are assigned the same global minimum value and all constrained maxima are assigned the same global maximum value, all the constraints will be extrema in the resulting field.

The problem of finding a harmonic function with Dirichlet boundary condition on the domain Ω is a standard PDE problem and it has several applications:

- Electromagnetism: electrostatic potential;
- Thermodynamic: heat diffusion in steady-state.

5.2 Laplace operator on case study problem

For our case study, we have selected the Laplace equation with Dirichlet boundary conditions:

$$\begin{cases} \Delta u = 0 \ in \ \Omega \\ u = f \quad in \ \partial\Omega \end{cases} \tag{10}$$

Given a discretizaion $\mathcal{D}$ of domain Ω and the associated connectivity graph $\mathcal{G}$, the Laplace-Beltrami operator is discretized with the Laplacian matrix:

$$L(i,j) := \begin{cases} w(i,j) & j \in \mathcal{D}, \\ -\sum_{k \in \mathcal{D}} w(i,k) & i = j, \end{cases}$$

which is sparse, symmetric and positive semi-definite.

On regular grids, weights w are costant and the boundary condition is set by imposing boundary values; the coefficient matrix $\widetilde{L}$ of the linear system can be defined as

$$\widetilde{L}(i,j) := \begin{cases} 1 & i = j, i \in \partial\mathcal{D}, \\ w(i,j) := 1 & (i,j) \in \mathcal{G}, i \neq j, \\ -\sum_{k \in \mathcal{D}} |(i,k) \in \mathcal{G}| & i = j, \end{cases}$$

and it is sparse and positive definite.

On irregular grids, weights w depend from grid geometry and the boundary condition is imposed with penalization techique; the coefficient matrix $\widetilde{L}$ of the linear system can be defined as

$$\widetilde{L}(i,j) := \begin{cases} 1 & j \in \partial\mathcal{D}, i = j, \\ w(i,j) := -\frac{\cot \alpha_{ij} + \cot \beta_{ij}}{2} & (i,j) \in \mathcal{G}, i \neq j, \\ -\sum_{k \in \mathcal{D}} w(i,k) & i = j, \end{cases}$$

and it is sparse, symmetric and positive definite.

The linear system $\widetilde{L}u = g$, where g depends on both the r.h.s. and the boundary values of Equation 10, has been used for all the tests of this casy study.

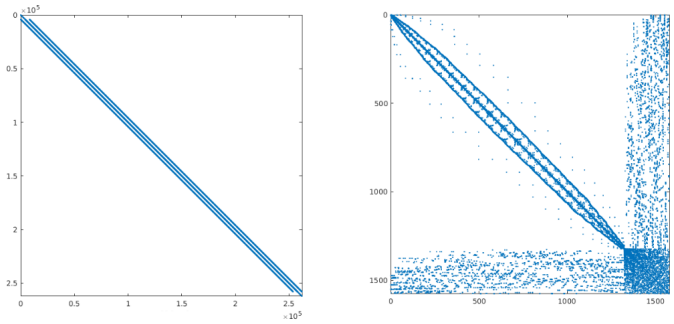

Figure 14: Sparsity patterns.

5.3 Domain and grid

2D and 3D domains have been discretized with regular grids and meshes (i.e., irregular grids). The domain discretization affects the sparsity pattern and the non-zero fill-in of the coefficient matrix.

As shown in Table 4, several matrices have been used for tests; elements represent the number of nodes used for the regular grid and Fill% represents the ratio between non-zero numbers and total elements.

Domain	Elements	Matrix Rows	Matrix Non-zeros	Fill %
Square	2048 x 2048	4 194 304	20 938 768	0.00012
Cube	128 x 128 x 128	2 097 152	14 099 408	0.0298
Cube	512 x 512 x 512	134 217 728	930 123 728	0.000005
Circle	-	4 152 441	29 053 205	0.00017
Sphere	-	2 094 834	33 225 967	0.00076

Table 4: Domain and grid examples.

Figure 14 shows the sparsity pattern for a regular grid on a cube domain (on the left) and for a tetrahedalization on a sphere domain (on the right). Sparsity pattern has an impact on data distribution among processes, MPI communications and processes work load; more regular matrices will have good balanced processes and better scalability performances.

6 Results

We present the scalability analysis of direct methods with two different libraries (SuperLU and MUMPS), a comparison between several preconditioned iterative methods, in order to identify which one offers the best performances; we also provide an analysis on large matrices, performed only for iterative methods on regular grids due to memory limits of FreeFem++ and SuperLU. Fixed a class of iterative solvers, we perform scalability analysis with iterative methods, an error analysis of iterative methods (compared to direct methods) and an analysis with multiple right-hand side terms. Finally, we analyze the impact of the discretization properties on the sparsity pattern and solvers performances (on irregular grids, exploiting FreeFem++ features) and the granularity varying with the number of processes (on regular grids).

Scalability analysis has been performed at high level on regular grids (Section 6.1) and on irregular grids (Section 6.2); a more detailed analysis has been performed on most expensive operations, through profiling tools of libraries (Section 6.3).

6.1 Regular grids

Direct methods scalability

Scalability analysis has been performed with direct solvers, comparing two libraries: SuperLU and MUMPS. Input data are summarized in Table 5.

Grid	$128 \times 128 \times 128$
Matrix rows	2 097 152
Matrix non-zeros	14 099 408

Table 5: Input data for direct methods.

Figure 15 shows the performances of both libraries, up to 576 processes (16 nodes); MUMPS shows worse performances in terms of execution time and scalability, thus SuperLU has been selected for further investigation.

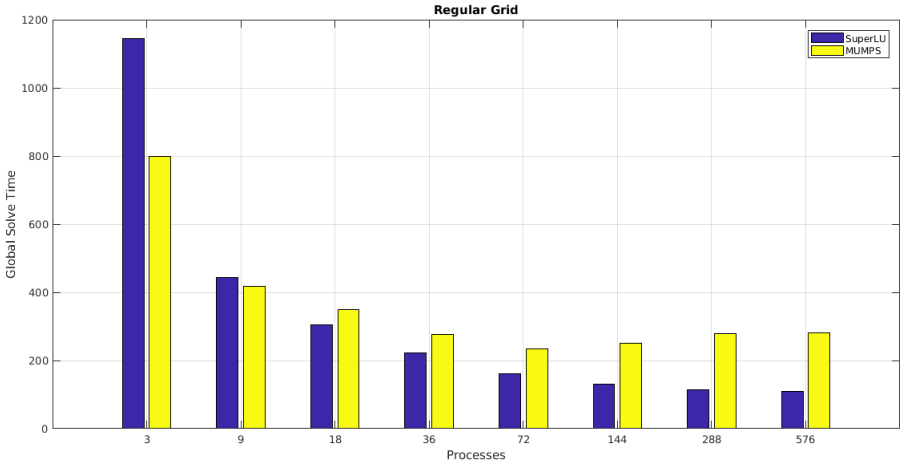

Figure 15: SuperLU and MUMPS comparison.

Figure 16 shows the scalability results for SuperLU. Scalability can be divided in: performances inside the node (1-3-9-18-36 processes) and performances among the nodes (36-72-144-288-576 processes, corresponding to 1-2-4-8-16 nodes). SuperLU has a good efficiency

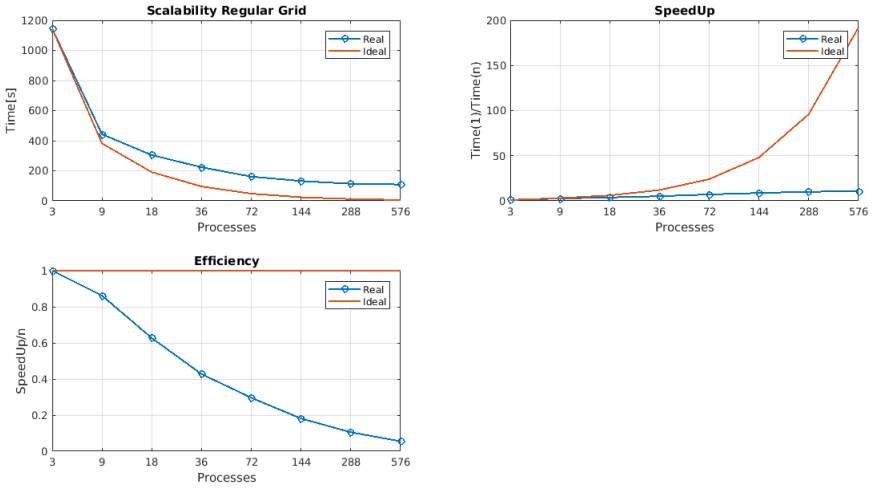

Figure 16: SuperLU scalability analysis.

of 42% on one node and an efficiency of 5% on 576 processes. The main reason is that some operations are not parallelized in the SuperLU library; since they do not scale on processes, their impact on global computation becomes heavier when increasing the number of processes. For further details, we refer the reader to Section 6.3.

Direct methods have a high global execution time, due to the factorization operations; indeed, in case of a single linear system, direct methods do not have comparable performances with respect to iterative methods. However, on multiple r.h.s. systems, they become valuable since the factorization is performed only once and only the numerical solver is applied to each system. Note that for direct methods, speed-up and efficiency are calculated with respect to $T(3)$ instead of $T(1)$, since serial data is not available for SuperLU: the reason is that serial approach of SuperLU is implemented in a different library and results are not comparable.

Iterative algorithms comparison

Preconditioners and iterative solvers have been compared on the input data reported in Table 6. These data have been selected for several reasons: the coefficient matrix has a number of rows and non-zeros comparable with common applications (computer graphics, engineering problems etc) [17]; with this error value, the approximation of the solution is good; the number of processes (72 on 2 nodes) allows us to distribute the computation on more than one node (so that intra and inter performances are both evaluated).

Grid	128 x 128 x 128
Matrix rows	2 097 152
Matrix non-zeros	14 099 408
Processes	72 (2 nodes)
Error	1e-12

Table 6: Matrix and condition for test.

Results (Figure 17) show the computation timings of five solvers coupled with three preconditioners plus a not preconditioned case (called *NONE*), in order to identify the best combination. For each solver (x-axis), all the preconditioners have been tested (columns) and bars show the time for the solver (blu) and for the preconditioning operations (yellow).

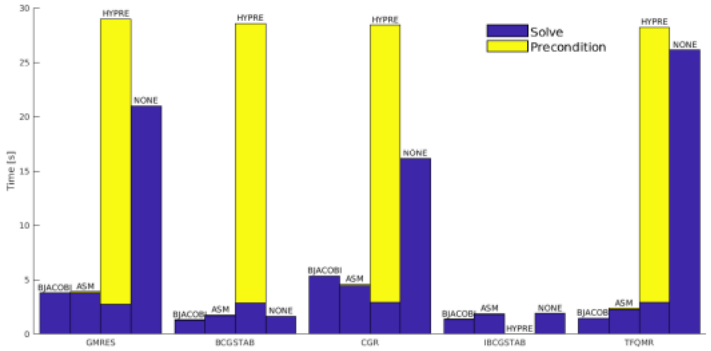

Figure 17: Iterative methods comparison.

As for the preconditioners, Block Jacobi and Additive Schwartz Method (ASM) have a very low computation time compared to iterative solvers; for this reason, their contribution (in yellow) is very small.

As for the solvers, excluding "NONE" case , GMRES and CGR appear as the worst iterative solvers in terms of solver time, while BICGSTAB, IBICGSTAB and TFQMR have similar results.

On the basis of this empirical experience, BICGSTAB has been selected as iterative solver and *Block Jacobi* as preconditioner for the solution of a single system; a further comparison based on an error analysis is presented in Section 6.1. Finally, *Hypre* has been considered for multiple r.h.s. tests and error analysis.

As preconditioners, *Block Jacobi* is a generic preconditioner that does not exploit geometric properties of the grid and it is suited for general purpose analysis. *Hypre* is a multigrid preconditioner which, after an initial time of preconditioning, can have good performances on solver routines; it has an impact on the number of iteration and on the solution accuracy.

As solver, BICGSTAB is an iterative method based on Lanczos iteration (Appendix A) and it is one of the most used iterative solver on literature [15].

Iterative methods scalability

Scalability analysis has been performed on the same conditions (Table 6) and the results are reported in Figure 18.

BICGSTAB shows good performances and scalability results up to 288 processes; a poor scalability at 576 processes is due to the increase of MPI messages and to the reduction of MPI messages' length, as detailed in Section 6.3. Furthermore, the overhead of communications leads to a reduction of granularity; due to all these factors, this algorithm is not able to scale efficiently when 576 processes are reached. As described in paragraph below, increasing the matrix dimension allows us to improve performances and to have good scalability results up to 64 Marconi nodes (2304 processes).

Scalability can be divided in: performances inside the node (1-3-9-18-36 processes) and performances among the nodes (36-72-144-288-576 processes, corresponding to 1-2-4-8-16 nodes). Efficiency inside a single node is 24%, at 8 nodes (288 processes) is 19%, while performances reduces with 16 nodes (576 processes) with an efficiency of 9%. Compared to ideal curves (red line in all graphs), the decrease of performances can be observed at the increase of processes used; the main reason is the increase of MPI communications on some operations of the algorithm, which will be detailed in Section 6.3. Furthermore, *Efficiency per iteration* extracts the impact of convergence (i.e., the increase of the number of iterations), since it considers an average time per iteration. In this case, efficiency with 36 processes is 35% while efficiency with 576 processes (16 nodes) is 19%. This reduction of

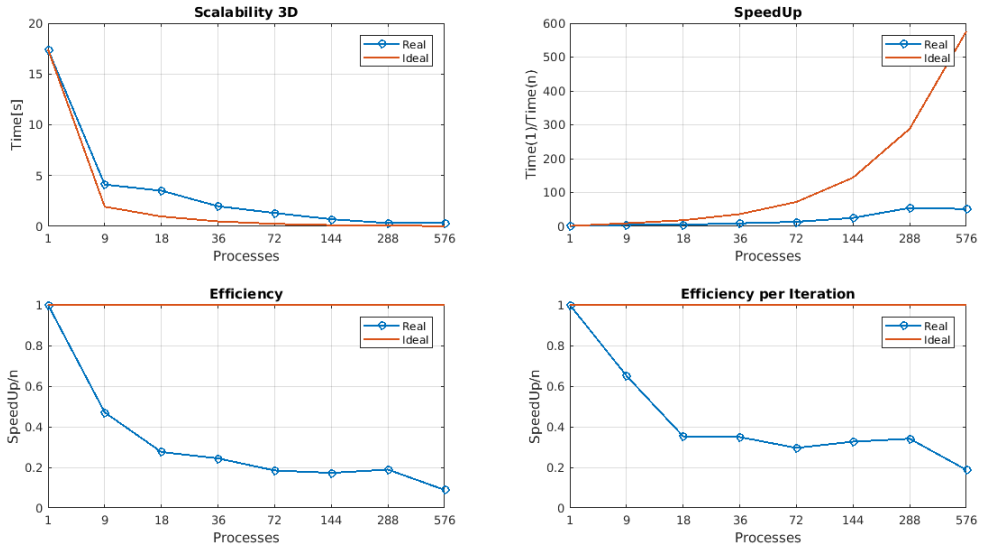

Figure 18: Iterative methods performances.

efficiency is due only to finer-grain granularity, whereas the overhead of MPI communications over computation causes cache inefficiency and leads to poor scalability performances.

Analysis on large matrices

In order to stress algorithms and scalability, we have analyzed the performances on large matrices (2D and 3D) with a high number of processes (up to 2304 processes, 64 processes).

Matrices have been chosen according to the maximum dimension reached by common applications (as described in section 6.1-Direct methods scalability). Results (Table 7) show good performances and scalability, even with a high number of processes. BICSTAB can solve a 1 billion non-zeros matrix in less than one minute with 1152 processes; furthermore, it scales very well passing from 32 to 64 nodes. As previously anticipated, this test confirms that the previous inefficiency with 576 processes was caused by a too fine-grained granularity and that, increasing the matrix size, the algorithm still scales on 64 nodes.

Domain	Grid	Matrix Rows	Matrix non-zeros	Process	Solver Time	MFLOPS
Square	4096 x 4096	16 777 216	83 820 560	1152	16.06	328 414
				2304	11.89	535 280
Cube	512 x 512 x 512	134 217 728	930 123 728	1152	51.23	167 534
				2304	19.5	453 828

Table 7: Large matrices.

Error analysis

In order to compare direct and iterative methods, we have analyzed the approximation error by comparing the exact solution with the computed solution as

$$x_{norm} = \frac{\|x_{ground\_truth} - x_{computed}\|_2}{\|x_{ground\_truth}\|_2}$$

The solution of direct methods is considered as a baseline, since it computes exact solution (unless approximation errors). As exit condition of itherative methods (e.g., maximum

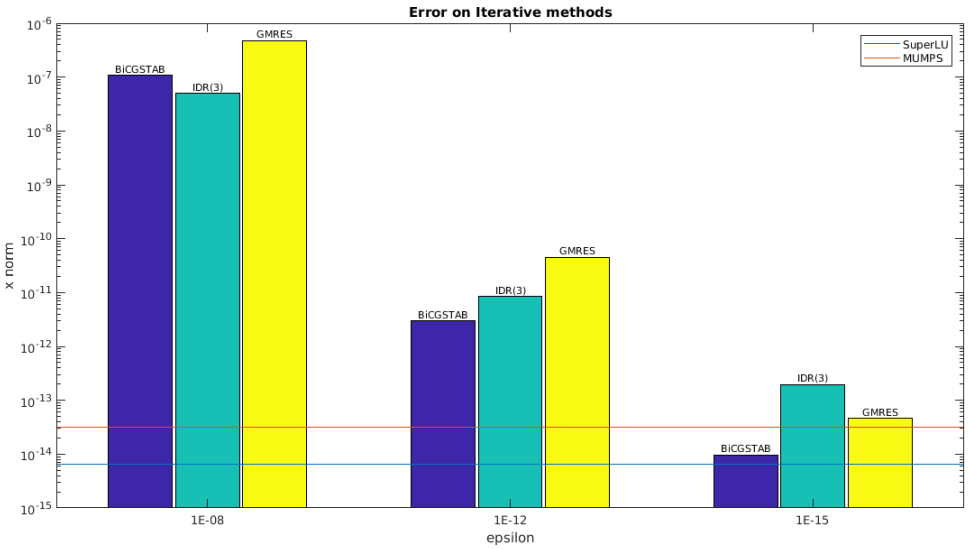

Figure 19: Error on iterative methods.

number of iterations, solution divergence), we have considered the relative error

$$\frac{\|Lu - g\|_2}{\|g\|_2} < \epsilon.$$

Decreasing ϵ, we expect that iterative methods reach an accuracy comparable with direct methods in terms of approximation accuracy, with an impact on the number of iterations and the computational time.

Tests have been performed on the matrices reported in Table 5 with one process only, since IDR(s) parallel implementation was not available. Figure 19 compares the approximation accuracy of several solvers without preconditioning; horizontal lines are the x_{norm} of direct methods (for both SuperLU and MUMPS), which can be considered as baseline. Iterative methods have similar results in terms of error on solution, even if BICGSTAB has shown best results.

Figure 20 shows a comparison of BICGSTAB with two preconditioners (Block-Jacobi and Hypre) and without preconditioner, for different values of ϵ (10^{-8}, 10^{-12}, 10^{-15}). The baseline is the horizontal line, which is reached at $\epsilon = 10^{-15}$. Preconditioners are represented in order to show different behaviours, not only in terms of solver time, but also in terms of approximation accuracy. In particular, Hypre, which has a higher time due to a strong preconditioning phase, allows us to reach a very accurate solution. However, Block-Jacobi offers a good performance in terms of approximation accuracy, considering that it is a general purpose preconditioner and its execution time is negligible. On the basis of this analysis, the choice of the preconditioner depends on the required solution accuracy.

Figure 21 shows the relationship between the approximation error on the solution (circle mark) and solver time (bar) for different ϵ; with ϵ varying from 10^{-8} to 10^{-15}, solution error reaches the order of direct method but the solver time is doubled. In this graph, only Block Jacobi and BICGSTAB are showed, but all the other solvers/preconditioners have a similar behaviour.

To conclude, the final choice of ϵ can be based on the required precision on the computed solution, taking into account that a precision comparable with direct methods is payed on the solver time.

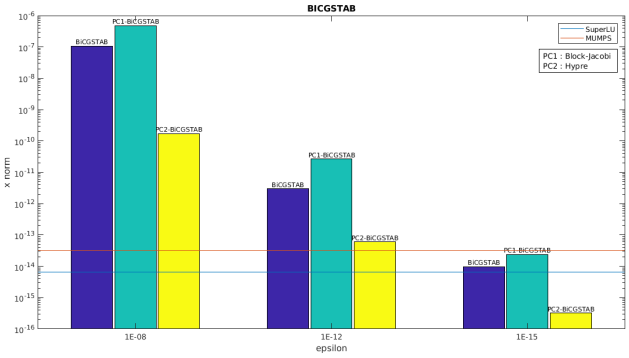

Figure 20: Preconditioners comparison.

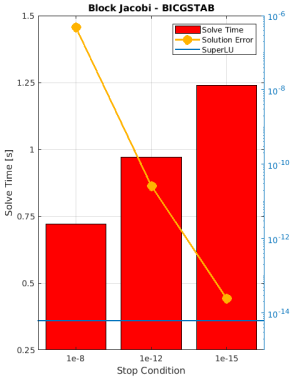

Figure 21: Solver time and iterations comparison.

Multiple r.h.s.

Direct methods become valuable in case of systems with multiple right-hand side terms (e.g., in a time-depending problem), since most of the operations (factorization, column permutation of the coefficient matrix) are performed only once while the solution is computed for each r.h.s.. On the contrary, iterative methods solve every time a new linear system, even if the coefficient matrix is not changed; preconditioning is the only operation which is performed once. The Block-Jacobi preconditioning is negligible in terms of time while if a strong preconditioner is chosen, execution time for preconditioning becomes relevant.

Both methods have been compared with a set of r.h.s. terms: iterative methods have been compared with two different preconditioners (Block Jacobi and Hypre) and with two ϵ values, passing one r.h.s. term per time; direct methods have been compared with two approaches: r.h.s. arrays passed individually or as a matrix (i.e., passed only once.) Furthermore, iterative methods have been evaluated also by solving t linear systems at the same time, placing t matrices in a global block matrix and finding as result a vector of t solution, as showed in Equation 11.

$$\begin{bmatrix} A_1 & 0 & ... & 0 \\ 0 & A_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & A_t \end{bmatrix} \cdot \begin{bmatrix} \overrightarrow{x_1} \\ \overrightarrow{x_2} \\ ... \\ \overrightarrow{x_t} \end{bmatrix} = \begin{bmatrix} \overrightarrow{b_1} \\ \overrightarrow{b_2} \\ ... \\ \overrightarrow{b_t} \end{bmatrix} \tag{11}$$

Figure 22 shows results, with a different number of r.h.s., of each approach; iterative methods have good performances if ϵ is low ($\epsilon = 10^{-8}$), giving results comparable with direct methods (passing r.h.s. terms once per time). With this error, iterative methods (blu line) have a smaller slope than direct methods (red line); the reason is that even if the solver phase (in direct methods) performs less operations than an iterative method, direct method works on factorized matrices, which have more non-zero elements. Thus, there is an ϵ limit where iterative methods become more performing.

Otherwise, when increasing the solution accuracy of iterative methods, the slope of direct methods becomes lower than iterative one and direct methods become more efficient after a certain number of iterations.

The second option for the direct methods (passing all r.h.s. terms once, as a matrix) solves the block linear system $AU = B$ where the solution is a matrix (each column is the solution to the linear system with a different r.h.s.). With this approach, direct methods have the best performances with respect to all the other methods; however, this approach has some limits in terms of maximum r.h.s. passed together, as this vector is dense; an empirical limit, for this test, is 500 r.h.s..

Since direct methods can have memory limits with large matrices, iterative methods are

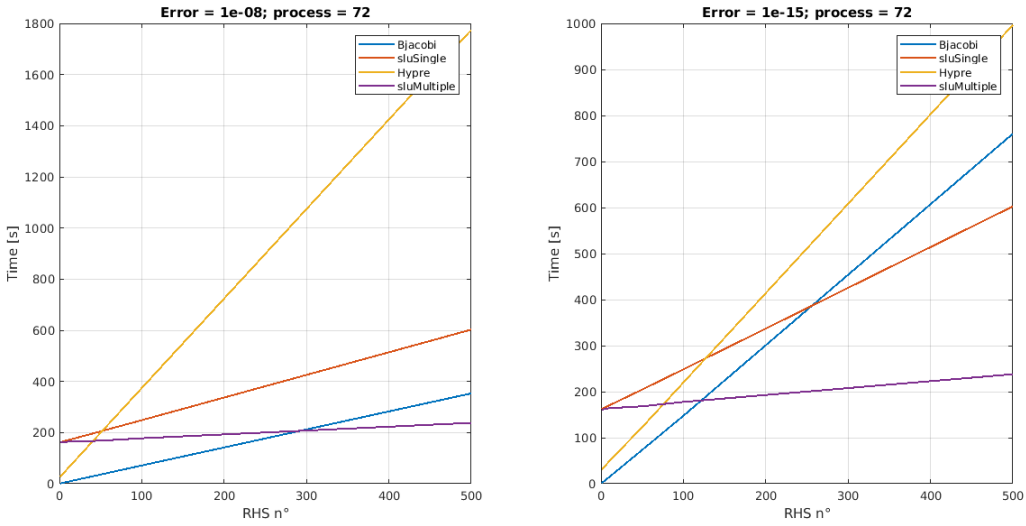

Figure 22: Multiple r.h.s..

a very good alternative, if a lower precision on the result is acceptable.

Finally, Figure 23 shows a comparison between iterative methods (Block Jacobi + BICGSTAB) of one system per time and iterative methods (Block Jacobi + BICGSTAB) of a block matrix as previously explained in Equation 11. For two different ϵ values, both approaches have similar performances; anyway, the block method shows a memory limit at 100 linear systems simultaneously; thus, first approach should be used.

6.2 Irregular grids

Irregular grids have been constructed with an external software (FreeFem++); a sphere has been selected as input domain, then the Laplace equation has been discretized on the tetrahedralization of the sphere and the coefficient matrix (Table 8) and r.h.s. have been extracted and passed to PETSc.

Domain	Sphere
Matrix rows	2 094 977
Matrix non-zeros	33 225 967
Fill in	0.0007%

Table 8: Irregular domain.

As already described in Section 5.3, this matrix (Figure 14, right) has several differences with respect to the regular case: the number of rows is basically the same but the number of non-zero elements is more than the double; the matrix sparsity pattern is irregular due to the arbitrary connectivity of the underlying mesh and also to FreeFem++ node ordering. All these elements have an impact on performances and scalability.

Direct methods scalability

Tests have been performed with the following direct solvers: SuperLU and MUMPS. On the basis of the results (Figure 24), SuperLU has a better performance than MUMPS, also on irregular grids.

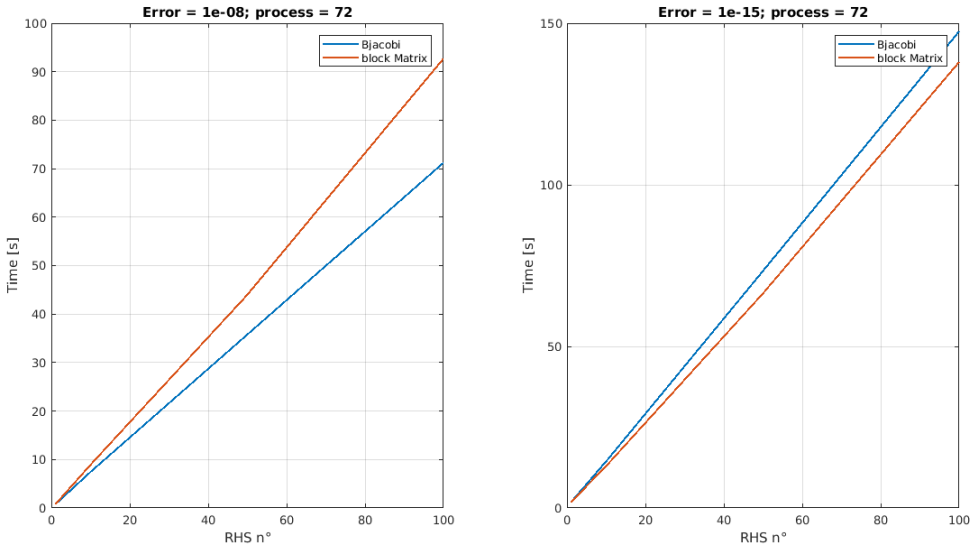

Figure 23: Multiple r.h.s. - block matrix comparison.

Figure 25 shows the results of SuperLU, which has a good performances on one node (64% efficiency) and 8% with 576 processes (16 nodes). The reason of this lower efficiency is that some operations (Section 6.3) are not parallelized, as already described for regular grids.

Global execution time is greater than on regular grids, but scalability intra/inter nodes is even better, due to increased number of non-zeros to be computed. Indeed, direct solvers have similar performances in terms of MPI behaviour and load balance between regular and irregular grids; column permutation and block data distribution allow us to reduce inefficiencies caused by irregular grids.

Iterative algorithms comparison

In Figure 26, preconditioners and solvers have been compared with 72 processes and an accuracy of order 10^{-12}. Only two preconditioners have been analyzed for this comparison, on the basis of previous results on regular grids: Block-Jacobi and Hypre. Block-Jacobi has a very low execution time and its contribution is not visible on the graph, indeed it has been selected as preconditioner (for the "generality and performance" properties already described in the analogous section for regular grids). Hypre, as for regular grids case, has a high precondition time; its use is not relevant on single system but becomes valuable on multiple r.h.s..

BICGSTAB, IBICGSTAB and TFQMR solvers (all based on Lanzcos Bi-orthogonalization and transpose free) have a similar execution time; BICGSTAB has been used as iterative method (in order to have results comparable with those ones on regular grids), but all the solvers are valuable for this kind of analysis. CGR method was not able to converge to the solution with both Block-Jacobi and Hypre preconditioners.

Iterative methods scalability

Concerning scalability analysis (input data reported in Table 8, results in Figure 27), BICGSTAB shows an efficieny of 9% on one node and an efficiency of 8% with 144 processes (4 nodes). Then, with 288 and 576 processes (respectively 8 and 16 nodes), there is a heavy loss of performances (0.1% efficiency). *Efficiency per iteration* with 36 processes is 28% while efficiency with 576 processes (16 nodes) is 0.4%. This efficiency at 36 processes shows an improvement, since it extracts the contribution of the increased iteration numbers.

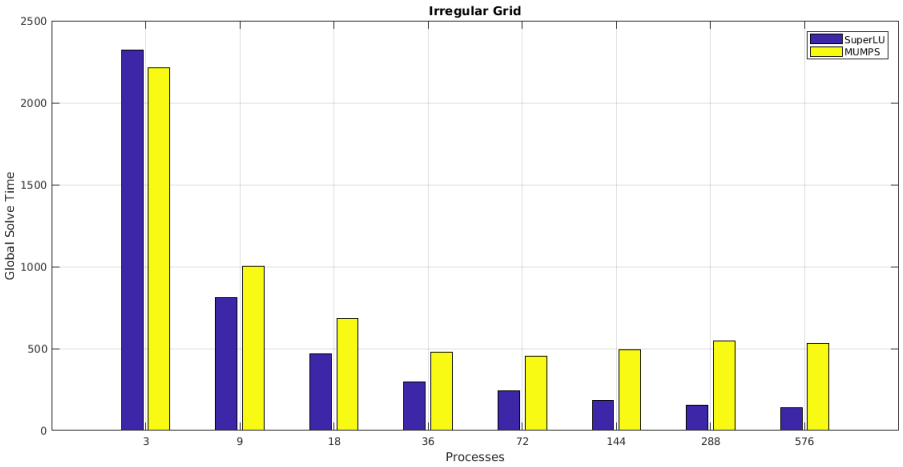

Figure 24: SuperLU and MUMPS comparison on irregular grids.

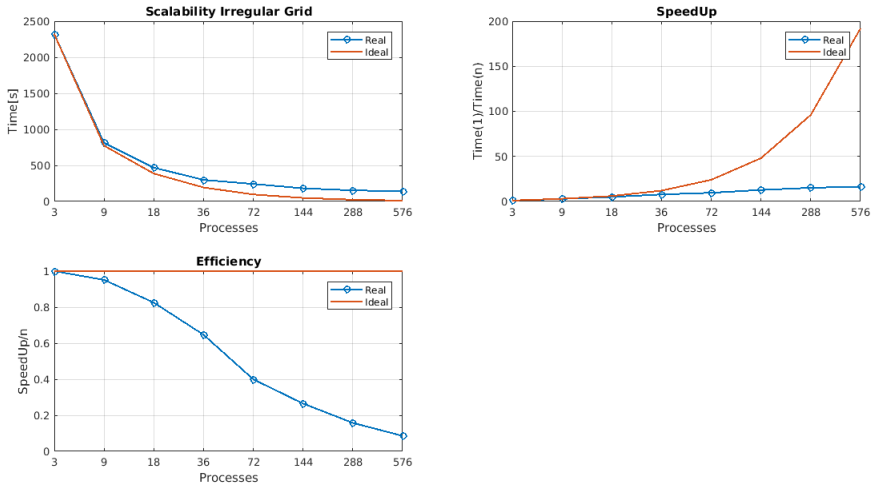

Figure 25: SuperLU scalability analysis.

Otherwise, with 576 processes the increase of MPI messages and the reduction of MPI messages' length, combined with the fine-grained granularity, lead to inefficiency; in Section 6.3, we will analyze how single operations contribute to efficiency decrease.

Comparing scalability results of regular and irregular grids, it is possible to affirm that:

- Solver time is higher with respect to regular grids, as the increased number of non-zero elements implies more operations to be performed;
- efficiency is reduced (comparing 288 nodes, regular grids have 19% while irregular grids have 9%). This dissimilarity is caused by a different sparisty pattern of the coefficient matrix, which implies an unbalanced load and an increase of MPI communications.

Analysis on large matrices

This analysis has not been performed on irregular grids, due to some limits of FreeFem++ for the generation of large irregular grids on the chosen domain.

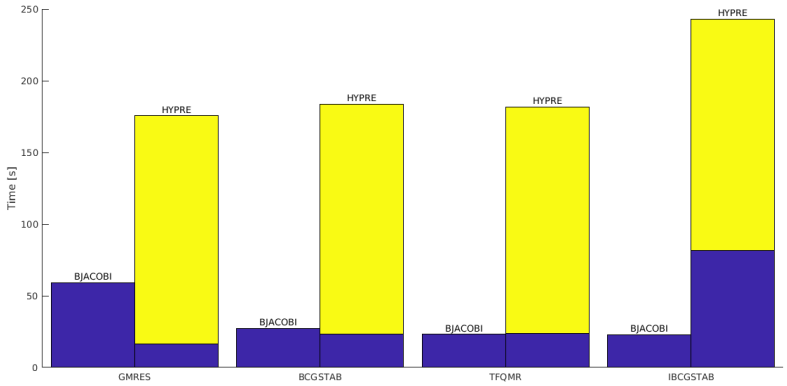

Figure 26: Iterative solvers and preconditioner.

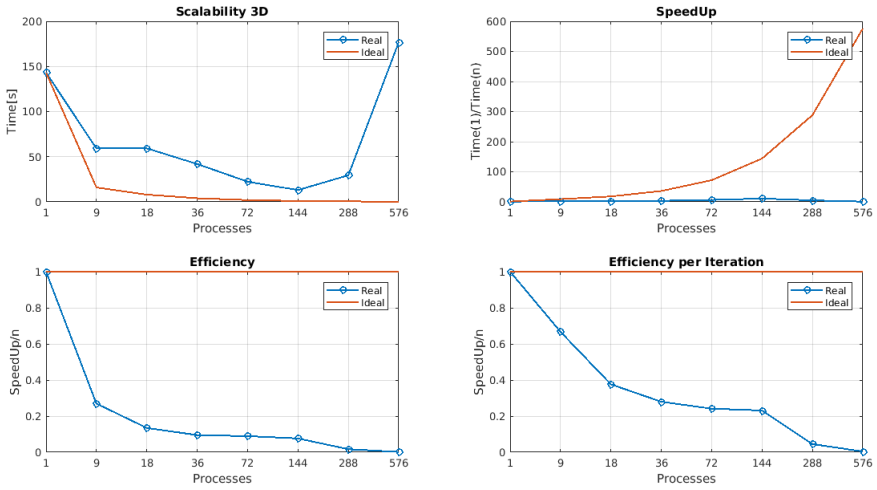

Figure 27: Irregular grids scalability.

Multiple r.h.s.
Iterative methods have been compared with two different preconditioners (Block Jacobi and Hypre) and $\epsilon = 10^{-8}$; direct methods have been compared with two r.h.s.: a set of vectors (passed once per time) and a matrix (passed only once). Results (Figure 28) show that iterative methods are not competitive with respect to direct methods on multiple r.h.s., even with low ϵ values. This aspect is due to a higher *Solve Time* of iterative methods on irregular grids (with respect to regular grids), while *Solve Time* of direct methods is basically the same. However, direct methods have memory limits on high-dimensional problems due to increased number of non-zero elements after the factorization; indeed, the choice between direct/iterative methods has to be based on the matrix size and if its sparsity allows direct methods. If we apply iterative methods, then a strong preconditioner allows us to reduce the number of iterations and the computation time.

Finally, the use of r.h.s. passed as a matrix is the best solution in terms of performances; however, analogously to the case of regular grids (Section 6.1), some size limits may occur with this approach.

The block approach for iterative method has not been used on irregular grids, due to memory limits on FreeFem++.

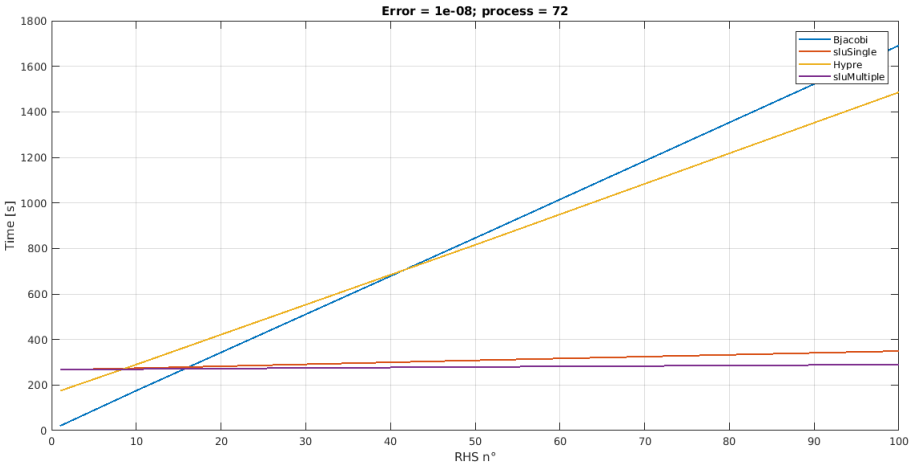

Figure 28: Multiple r.h.s. with irregular grids.

Analysis on matrix sparsity pattern

FreeFem++ offers several possibilities, such as changing the input domain, PDE, discretization methods. In particular, the polynomial degree of the FEM discretization affects the sparsity pattern of the coefficient matrix while increasing the approximation accuracy and, consequently, the solver time. FreeFem++ implements several types of basis functions [23]; most used basis functions are $P<n>$ where n is the order of the polynomial approximation of the solution to the PDE. As shown in Figure 29, increasing the polynomial order affects the number and the structure of non-zeros of the coefficient matrix, besides its size.

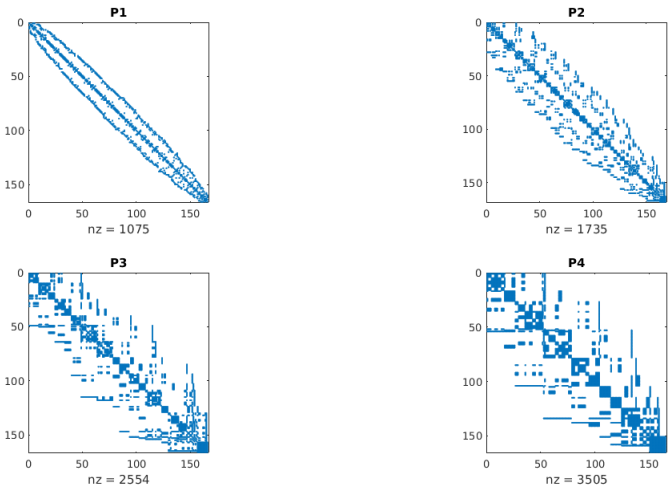

Figure 29: Different polynomial basis.

Polynomial Basis	Nodes	Elements	Non-zeros	Solver time [s]	Iterations	Solution accuracy
P1	86 425	86 425	602 969	0.11	358	$5.0 \cdot 10^{-6}$
P2	86 425	344 697	3 894 474	0.52	689	$2.3 \cdot 10^{-10}$
P3	86 425	774 817	13 148 592	3.32	938	$6.0 \cdot 10^{-10}$
P4	86 425	1 376 785	32 319 425	10.21	1260	$1.7 \cdot 10^{-9}$

Table 9: Polynomial basis performances.

Tests (Table 9) performed with fixed input parameters (Circle domain, BICGSTAB + Block-Jacobi solver, $\epsilon = 10^{-12}$, 72 processes) show that, fixed the nodes of the grid, the coefficient matrix size and its non-zero numbers increase when passing from degree 1 to degree 4; this leads to an increase of the number of iterations and solver time. The approximation accuracy of the solution (computed for nodes elements only) improve significantly when passing from degree 1 to degree 2, while it remains unchanged with degree 3 and 4.

The choice of the polynomial degree depends on the target approximation accuracy of the solution, taking into account the increased computational cost and the execution time needed.

6.3 Profiling

PETSc and SuperLU offer the possibility to go deeper into the analysis of the alghoritms, in order to better understand the scalability properties in terms of performed operations. Our analysis has considered regular and irregular grids for both methods and Table 10 shows the matrices used for the profiling.

With iterative methods, the whole program is divided into three sections: *Main Stage* is the data load, *Preconditioning* calculates the preconditioned matrix and *Solve* is the solver algorithm (Figure 30). Each section is divided into functions (*MatMult*, *MatSolve* etc.); for each function we report: the number of calls to each function, the time spent and the FLOPS (with the ratio between maximum and minimum among processes), the number of messages (MPI communications), the average length of messages and the number of reduction operation. Percentages refer to these five data, divided into global percentage (the whole program) and stage percentage. Finally, MFLOPS are calculated.

With direct methods, SuperLU offers a less detailed profiling; statistics provided by this library include execution time and MFLOPS for main operation (i.e. solver, factorization, column permutation, matrix distribution).

Domain	Sphere	Cube
Grid	-	128x128x128
Matrix rows	2 094 977	2 097 152
Matrix non-zeros	33 225 967	14 099 408

Table 10: Domain for Profiling.

6.3.1 Iterative methods

Operations for iterative methods could be divided into three phases, as shown in Figure 30; only Solve stage will be analyzed, since it holds most of the time (85 % in the example). Results refer to 72 processes.

For regular grids, Pareto chart for the solver operations, in percentage on total solver time (Figure 31), shows that most of the time is spent in *MatMult* and *MatSolve* operations, that are called twice per iteration. A minor but significant impact is given also by scalar product operations (*VecDot* and *VecDotNorm2*), which are called respectively twice and once per iteration.

For irregular grids, *MatMult* takes most of the time, with a significant impact on scalability (Figure 31).

Data

Functions

Stage	Event	Count		Time (sec)		Flops					Global					Stage					
		Max	Ratio	Max	Ratio	Max	Ratio	Mess	Avg Length	reduction	%T	%F	%M	%L	%R	%T	%F	%M	%L	%R	Mflop/s
Main Stage	MatAssemblyBegin	1	1	1.55E+00	486.3	0.00E+00	0	0.00E+00	0.00E+00	2.00E+00	2	0	0	0	0	12	0	0	0	11	0
	MatAssemblyEnd	1	1	6.65E-02	1.2	0.00E+00	0	2.80E+02	1.30E+05	8.00E+00	0	0	0	0	1	1	0	50	2	44	0
	MatLoad	1	1	4.18E+00	1	0.00E+00	0	5.00E+02	3.00E+06	1.30E+01	13	0	0	3	1	90	0	88	92	72	0
	VecSet	1	1	2.28E-04	2.9	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	0	0	0	0	0	0
	VecAssemblyBegin	1	1	2.37E-01	35.7	0.00E+00	0	0.00E+00	0.00E+00	3.00E+00	0	0	0	0	0	7	0	0	0	17	0
	VecAssemblyEnd	1	1	2.60E-05	13.6	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	0	0	0	0	0	0
	VecLoad	1	1	2.92E-01	1	0.00E+00	0	7.10E+01	1.90E+06	4.00E+00	1	0	0	0	0	6	0	12	8	22	0
Preconditioning	MatILUFactorNum	1	1	7.59E-07	1.5	4.34E+06	1.4	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	73	100	0	0	0	11953
	MatILUFactorSym	1	1	2.25E-02	1.6	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	22	0	0	0	0	0
	MatGetRowIJ	1	1	7.61E-05	63.8	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	0	0	0	0	0	0
	MatGetOrdering	1	1	2.06E-03	2	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	2	0	0	0	0	0
	KSPSetUp	2	1	8.24E-02	6.4	0.00E+00	0	0.00E+00	0.00E+00	2.00E+00	0	0	0	0	0	53	0	0	0	100	0
	PCSetUp	2	1	5.04E-02	1.4	4.34E+06	1.4	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	47	100	0	0	0	6135
	PCSetUpOnBlocks	1	1	5.04E-02	1.4	4.34E+06	1.4	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	47	100	0	0	0	6137
Solve	MatMult	738	1	1.12E+01	1.6	2.21E+09	1.4	1.00E+05	5.20E+05	0.00E+00	31	36	99	97	0	36	36	100	100	0	14054
	MatSolve	739	1	8.09E+00	1.4	2.02E+09	1.3	0.00E+00	0.00E+00	0.00E+00	23	33	0	0	0	27	33	0	0	0	17807
	VecDot	738	1	5.03E+00	2.9	3.44E+08	1	0.00E+00	0.00E+00	7.40E+02	10	6	0	0	49	11	6	0	0	50	4924
	VecDotNorm2	369	1	4.61E+00	3.4	3.44E+08	1	0.00E+00	0.00E+00	3.70E+02	8	6	0	0	25	10	6	0	0	25	5370
	VecNorm	370	1	7.36E-01	1.8	1.72E+08	1	0.00E+00	0.00E+00	3.70E+02	2	3	0	0	25	2	3	0	0	25	17101
	VecCopy	3	1	4.37E-03	1.6	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	0	0	0	0	0	0
	VecSet	742	1	5.28E-01	1.5	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	1	0	0	0	0	1	0	0	0	0	0
	VecAXPBYCZ	738	1	1.66E+00	1.2	6.88E+08	1	0.00E+00	0.00E+00	0.00E+00	5	11	0	0	0	6	11	0	0	0	29898
	VecWAXPY	738	1	1.74E+00	1.2	3.44E+08	1	0.00E+00	0.00E+00	0.00E+00	5	6	0	0	0	6	6	0	0	0	14237
	VecScatterBegin	738	1	7.90E-01	2.6	0.00E+00	0	1.00E+05	5.20E+05	0.00E+00	2	0	99	97	0	3	0	100	100	0	0
	VecScatterEnd	738	1	2.48E+00	3.3	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	3	0	0	0	0	4	0	0	0	0	0
	KSPSolve	1	1	2.63E+01	1	6.12E+09	1.2	1.00E+05	5.20E+05	1.50E+03	85	100	99	97	99	100	100	100	100	100	16650
	PCSetUpOnBlocks	1	1	6.99E-05	0	0.00E+00	0	0.00E+00	0.00E+00	0.00E+00	0	0	0	0	0	0	0	0	0	0	0
	PCApply	739	1	8.54E+00	1.4	2.02E+09	1.3	0.00E+00	0.00E+00	0.00E+00	24	33	0	0	0	28	33	0	0	0	16881

Figure 30: Profiling stages.

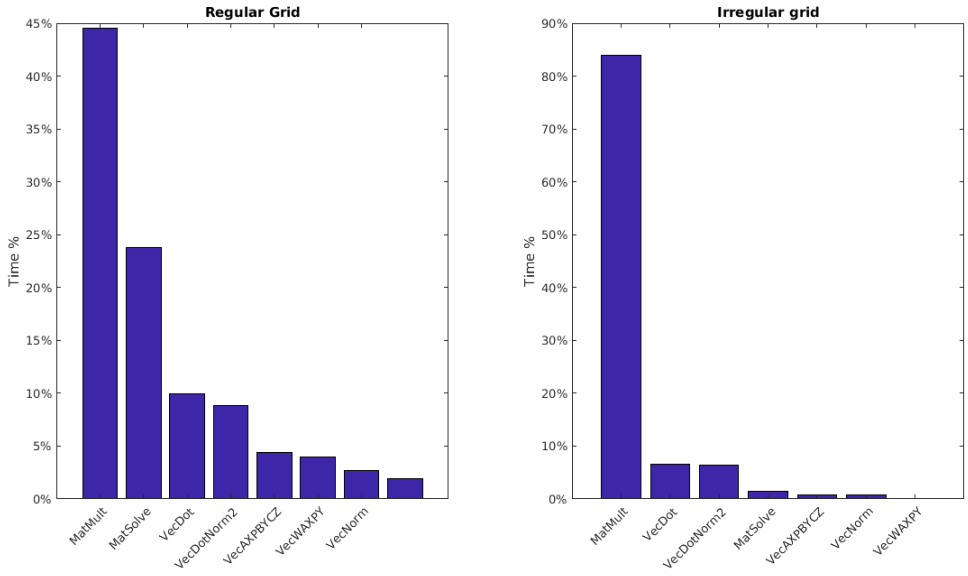

Figure 31: BICGSTAB Pareto.

These operations have been analyzed in terms of scalability, MPI communications and reduction operations. It is also analyzed the *Ratio*, i.e. the ratio between maximum and minimum process (in terms of execution time); this metric is relevant because unbalanced operations have an impact on the granularity and, consequently, on the efficiency; infact high Ratio implies an increase of processes' idle time and a finer-grain granularity.

MatMult is the matrix-vector product and it is called twice in the BICGSTAB algorithm. *MatMult* has both computational and communication parts.

Time (max among processes) and max/min ratio between processes are shown in Table 11.

As shown in Figure 32 on left side, on regular grids *MatMult* has an efficiency of 12% on 36 processes and of 8% with 576 processes (16 nodes); efficiency per iteration is 17.8%

	Regular Grids		Irregular Grids	
Process	Time(max)	Ratio	Time(max)	Ratio
1	5.23	1	66.2	1
3	2.66	1	61.6	1
9	1.54	1.1	52.1	1.2
18	1.46	1.1	56.0	1.2
36	1.16	1.5	38.5	1.1
72	0.615	1.3	20.6	1.3
144	0.351	2.2	12.3	1.3
288	0.117	4.3	15.9	1.7
576	0.113	4.4	175.0	12.6

Table 11: MatMult.

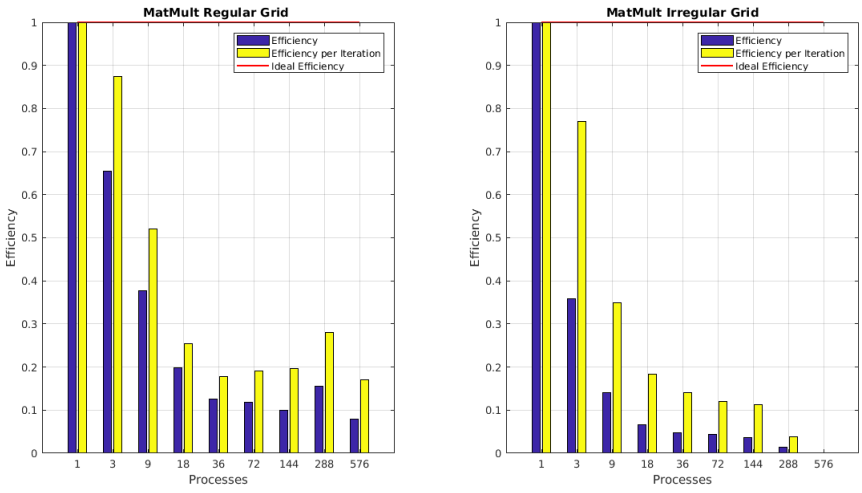

Figure 32: MatMult efficiency.

on 36 processes and 16.9% on 576 processes. Efficiency is improved if slower convergence is not considered (i.e., efficiency per iteration metric).

This efficiency result is mainly due to the increase of MPI messages and to the reduction of messages length, as shown in Figure 33. On the left, it is reported the scalability of *MatMult* operation compared with ideal scalability; on the right, it is reported the number of MPI communications (blu bar) and Average Message Length (yellow Bar). The ratio remains well balanced and does not have a significant impact on efficiency.

As shown in Figure 32 on right side, on irregular grids *MatMult* has an efficiency of 4.8% on 36 processes and of 1.4% with 576 processes (16 nodes); even efficiency per iteration has poor efficiency, when a high number of processes is used.

With 576 processes, *MatMult* has a huge increase of time, mainly due to the number of MPI communications and message length (Figure 34); in fact, the number of messages is two order of magnitude greater than the regular grids case, while the average message length is one order of magnitude lower. Furthermore, there is a significant increase of the ratio value, which has an impact on granularity. This result can be identified as the main cause of poor performances, as previously shown in Figure 27.

MatSolve solves a linear system with factored matrix and it is called twice in the BICGSTAB algorithm. Time (max between processes) and ratio (max/min between processes) are shown

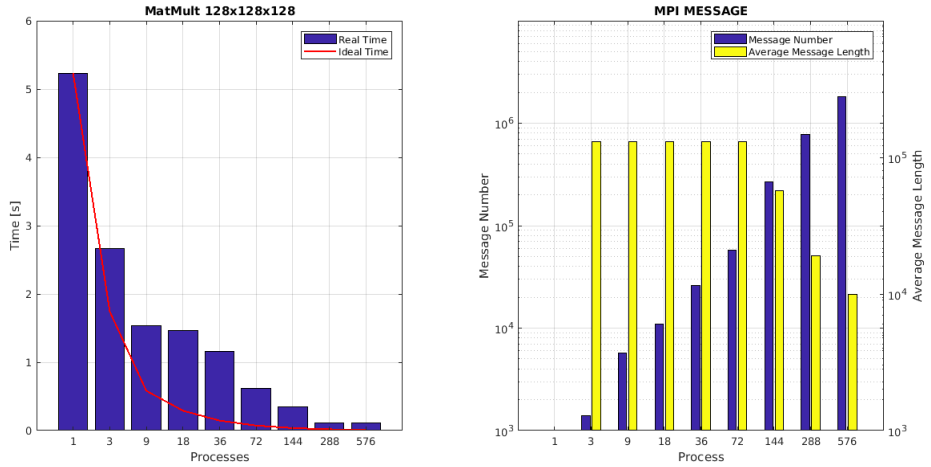

Figure 33: MatMult scalability on regular grids.

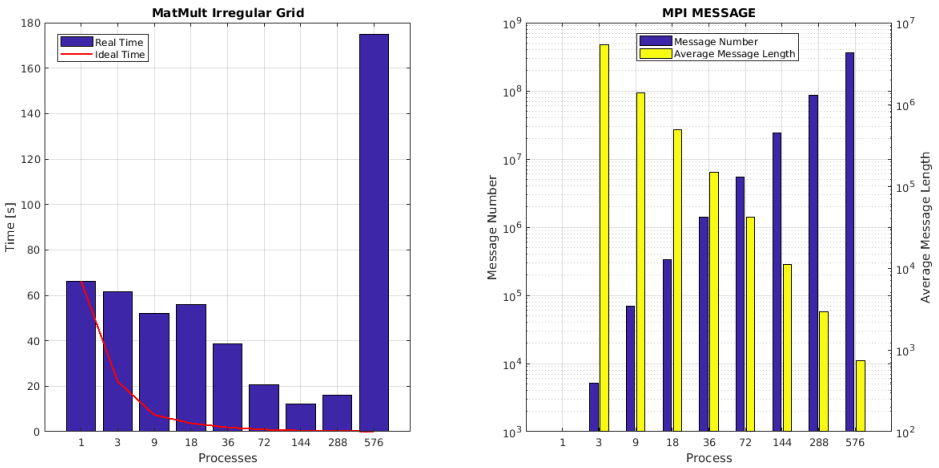

Figure 34: MatMult scalability on irregular grids.

in Table 12. *MatSolve* has very good scalability performances for both regular (Figure 35, left side) and irregular (Figure 35, right side) grids, due to the absence of MPI communications and to the reduction of computing time; *MatSolve* can be considered as a massively parallel operation.

As shown in Figure 36, on regular grids, efficiency is 25.4% on 36 processes and 29.1% with 576 processes (16 nodes); performances are very good, specially on scalability results among processes. On irregular grids, efficiency is 125% on 36 processes and 113% on 16 nodes; in this case, there is a superlinear scalability up to 576 processes. Results are even better when considering efficiency per iteration, with 61% on regular grids with 576 processes and 313% on irregular grids with 576 processes.

The reasons of such good performances depends on the kind of operation (massively parallel) and very high performing resources, which both lead to cache efficiency [38].

It is noted that the communication part is all managed by *MatMult* operation, letting to *MatSolve* the computational part only.

	Regular Grids		Irregular Grids	
Process	Time(max)	Ratio	Time(max)	Ratio
1	7.68	1	67.8	1
3	3.54	1	24.7	1.2
9	1.52	1.1	5.26	1.1
18	1.28	1.2	2.9	1.5
36	0.838	1.7	1.5	2.3
72	0.351	2.1	0.56	2.9
144	0.148	4.7	0.354	3.3
288	0.0575	4.8	0.186	5
576	0.0458	5.8	0.104	5.2

Table 12: MatSolve.

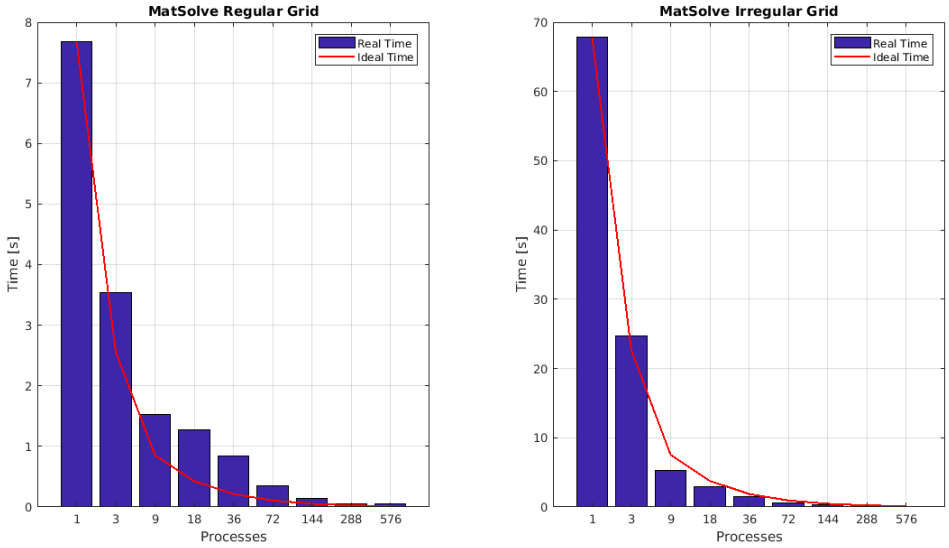

Figure 35: MatSolve scalability.

VecDot is the scalar product between two vectors and it is called twice in the BICGSTAB, plus a call to *VecDotNorm2*, which also calculates the norm of the vector w. *VecDot* has a computational part (product and sum operation) and a communication part (reduction of each process's result). Time (max between processes) and ratio (max/min between processes) are shown in Table 13

On regular grids, *VecDot* has a poor efficiency and efficiency per iteration (respectively 0.7% and 1.6% with 576 processes, as shown in Figure 37 on left side), and execution time is mainly affected by the load distribution on processes more than reduction numbers (which depends only on the number of iterations). As empirical analysis, there is an increase of execution time with 36 processes due to the increased ratio, while the execution time reduction from 144 to 288 processes is due to the improved load balance and the reduction of *Reduction* operations; however, there is not a clear correlation between these three metrics. Figure 38 reports VecDot scalability on regular grids.

On irregular grids, efficiency and efficiency per iteration are very low (both below 0.01% with 576 processes, as shown on Figure 37 on right side); *VecDot* execution time is almost costant till 144 processes, has a slight increase up to 288 processes and at 576 unbalanced load has a huge impact on time, as reported in Figure 39. Also in this case, excluding the 576 processes case, there is not a clear correlation between reductions, ratio and execution

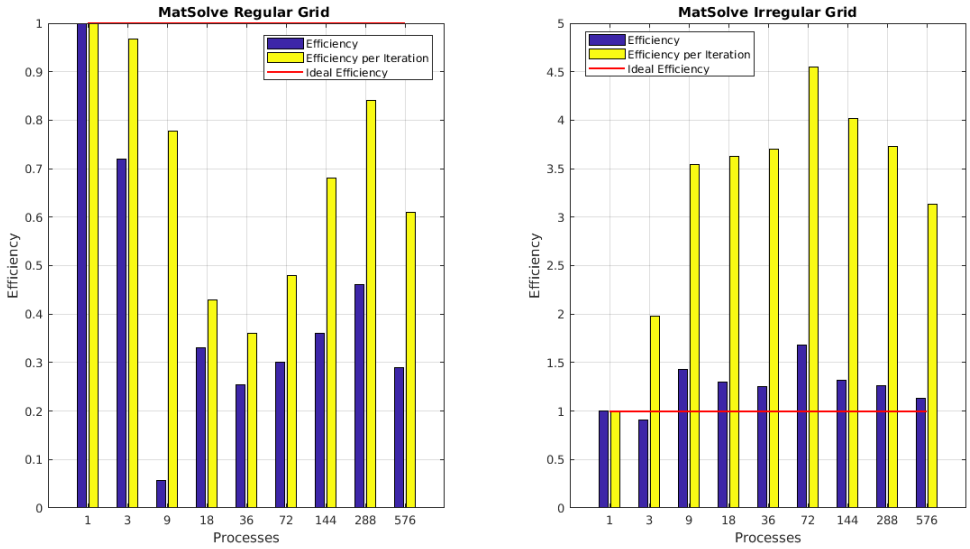

Figure 36: MatSolve efficiency.

	Regular Grids		Irregular Grids	
Process	Time(max)	Ratio	Time(max)	Ratio
1	0.544	1	0.784	1
3	0.312	1.2	2.85	4.8
9	0.228	1.8	3.55	12.6
18	0.235	2.3	3.7	14.2
36	0.419	6.7	3.25	1.9
72	0.219	3.1	2.34	11.4
144	0.223	3.2	1.21	11.9
288	0.113	2.7	10.9	1.5
576	0.125	2.3	81.5	176

Table 13: VecDot.

time.

Granularity Figures 40 and 41 show scalability of regular and irregular grids respectively, divided into operations with MPI Call (in blu) and operation with computing only (in yellow).

For both regular and irregular grids, the ratio between communication and computing time increases with the processes, leading to a reduction of granularity. Communication includes vector scattering operation (called by *MatMult*) and reduction operation (called by *VecDot* and *VecNorm* operations).

Granularity becomes coarser as matrix dimension is increased or when, with a fixed matrix dimension, a lesser number of processes is used; indeed, Figure 42 shows percentage balance of computing only operations (in yellow) and operations with communications (in blu). Results are shown for three matrix dimensions and each column shows the number of processes used (144, 288 and 576).

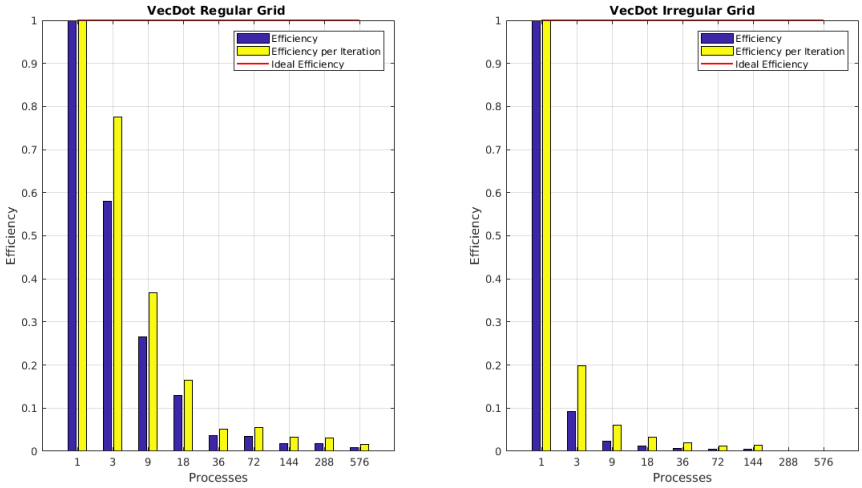

Figure 37: VecDot efficiency.

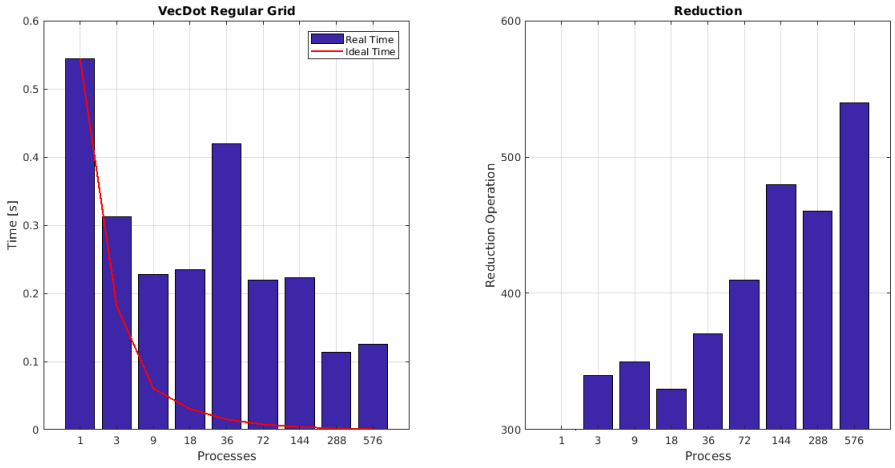

Figure 38: VecDot scalability on regular grids.

6.3.2 Direct methods

PETSc environment calls external libraries (SuperLU and MUMPS), whose level of detail for single algebraic operations is minor.

Figure 43 and Figure 44 show, for regular and irregular grids respectively, performances and scalability results of both libraries, with five principal steps stacked:

- *distribution* performs a cyclic block data distribution among processes (Figure 6);
- *column permutation* reorder the columns of the coefficient matrix, in order to optimize the number of non-zero elements after the factorization;
- *symbolic factorization* provides the number (and the position) of non-zero elements on LU matrices;
- *factorization* performs the numeric factorization of the coefficient matrix, in order to compute numerically the factor matrices;

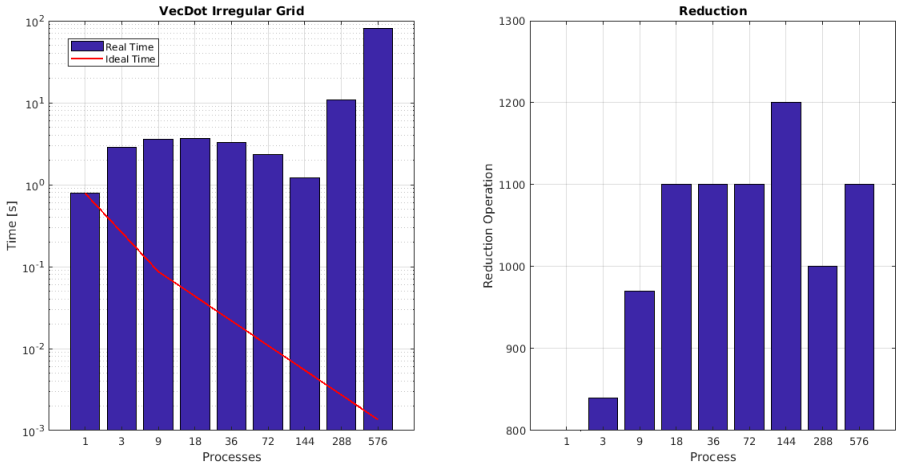

Figure 39: VecDot scalability on irregular grids.

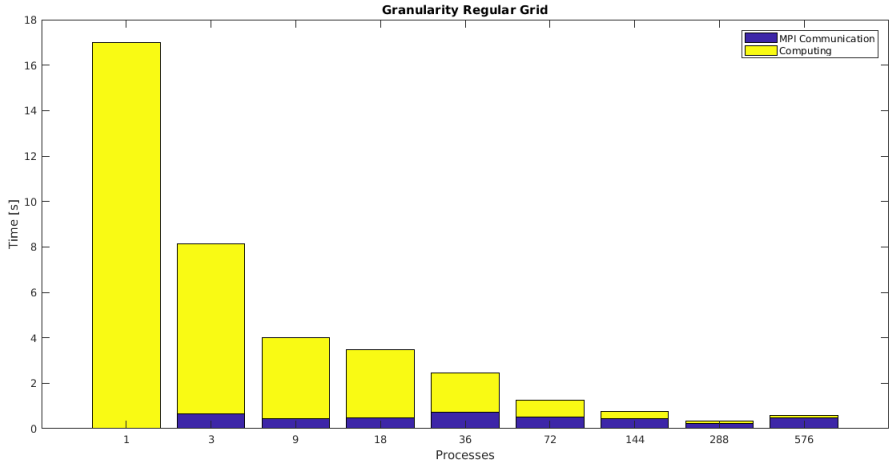

Figure 40: Granularity on regular grids.

- *solver* computes the solution of the triangular systems.

However, since now on only SuperLU will be commented, since it achieves better performances (as described on Figure 43 and Figure 44).

Two operations are not parallelized: symbolic factorization and column permutation. In fact, their time is constant with processes increase. Data distribution impact is not significant since it is performed once and its execution time is much lower than factorization.

Analizying *Solver* and *Factorization*, with regular grids the factorization scalability has a good efficiency (56% with 36 processes, 12% with 576 processes); Figure 45 shows time scalability compared to ideal time (left) and efficiency decrease (right): scalability results are good up to 576 processes.

Solver operation has 24% efficiency with 36 processes, 6% efficiency with 288 processes (8 nodes) and a time increase with 576 processes (with an efficiency of 1.8%, as reported in Figure 46).

Table 14 reports the increase of non-zero elements between A and $L + U$, with growth factor of 297. Even if multigraph analysis and column permutation are performed, the increase of non-zeros is significant; this has some consequence:

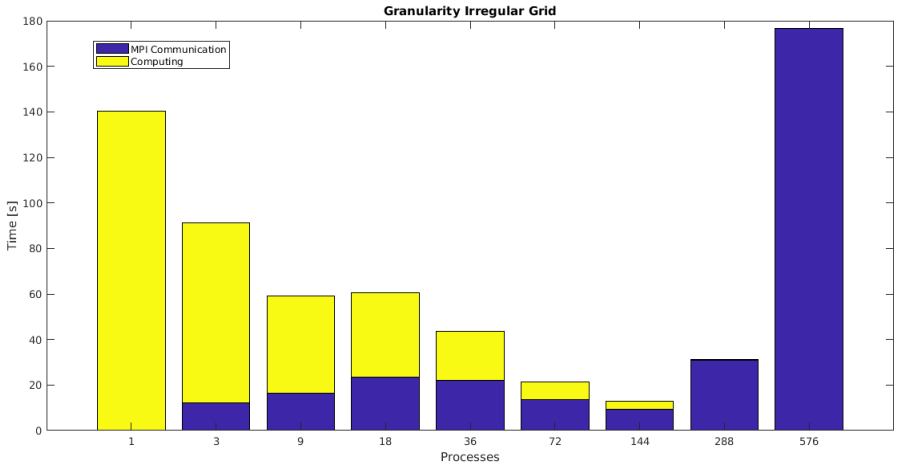

Figure 41: Granularity on irregular grids.

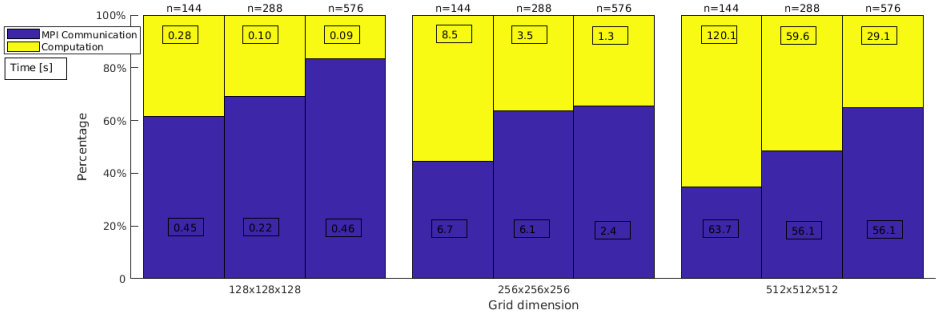

Figure 42: Granularity on large matrices.

- Data space for LU matrices increases and this could lead to memory limits with direct methods. For example, SuperLU was not able to factorize a coefficient matrix coming from a $256 \times 256 \times 256$ regular grid;
- The *Solve* operation is performed on a bigger matrix (in terms of non-zeros) than iterative methods. For this reason, even if the *Solve* operation has a lower computation than iterative methods, it performs more floating operations due to a larger number of non-zero elements. This is why it could happen that, as explained previously in Figure 22, iterative methods have a minor slope with respect to direct method.

A	L+U	Ratio
14 099 408	4 209 973 954	297

Table 14: LU elements.

On irregular grids, factorizazion has also very good performances (90% efficiency on the node, 30% among 16 nodes), as shown in Figure 47. Like regular grids, factorization has both a phase of computation and communication and it gives good scalability results up to 576 processes. Solver operations have similar performances to regular grids case, with a 32% efficiency on the node, 12% efficiency among 8 nodes and a time increase with 16 nodes (Figure 48), with an efficiency of 3% on 576 processes. Table 15 reports the increase of non-zero elements, with a growth factor of 228.

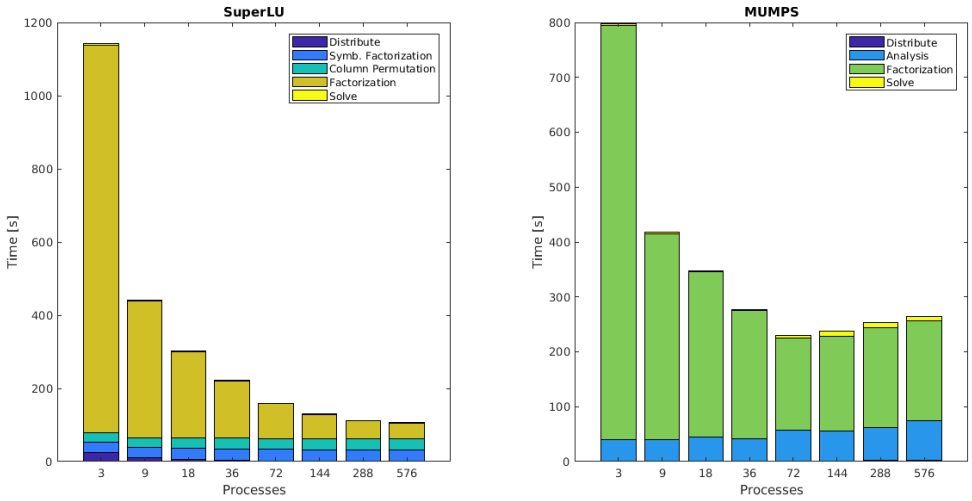

Figure 43: Direct methods performances on regular grids.

A	L+U	Ratio
33 225 967	7 615 661 965	228

Table 15: LU elements.

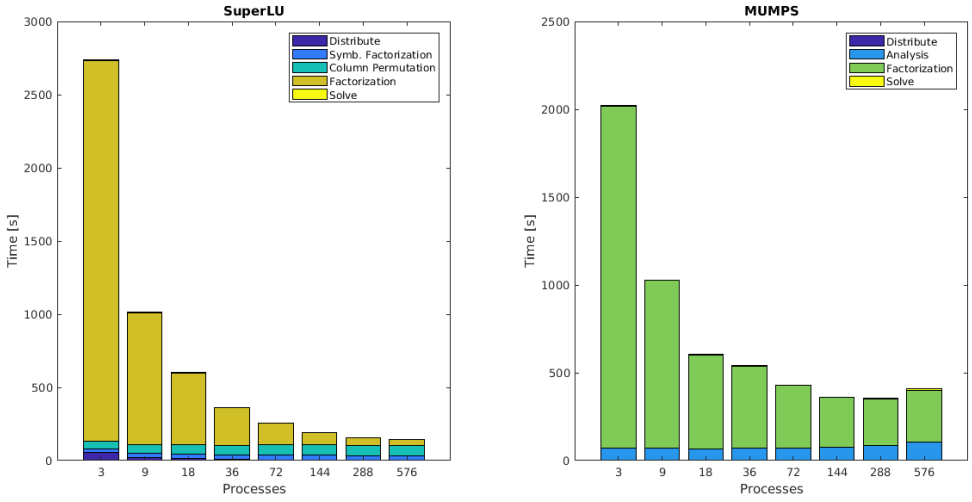

Figure 44: Direct methods performances on irregular grids.

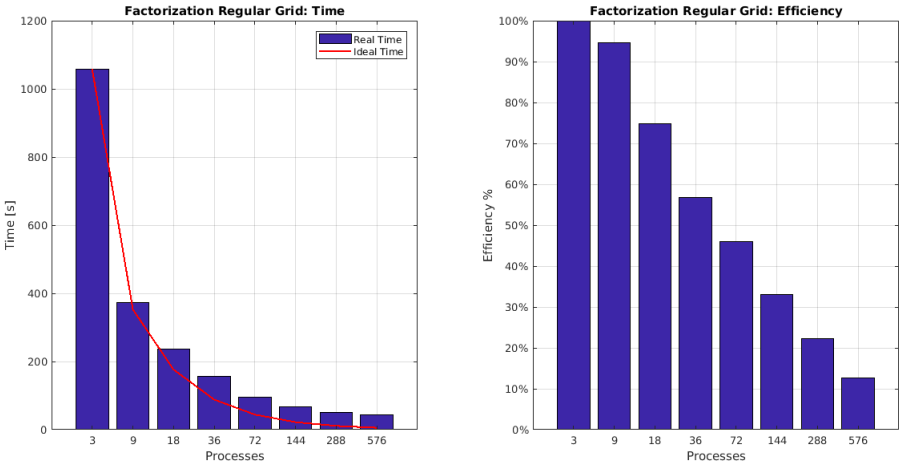

Figure 45: Regular grids factorization.

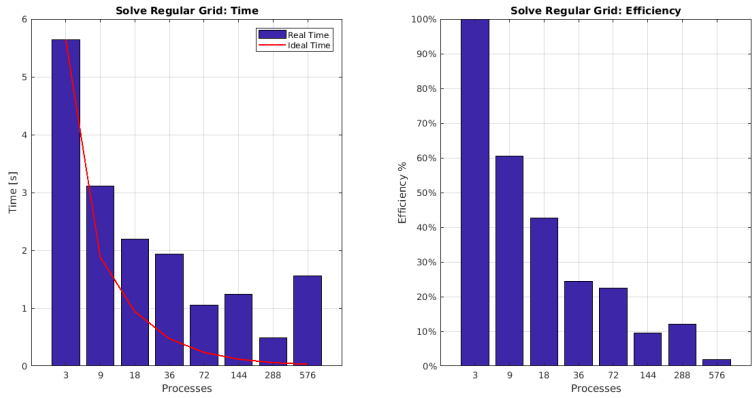

Figure 46: Regular grids solver.

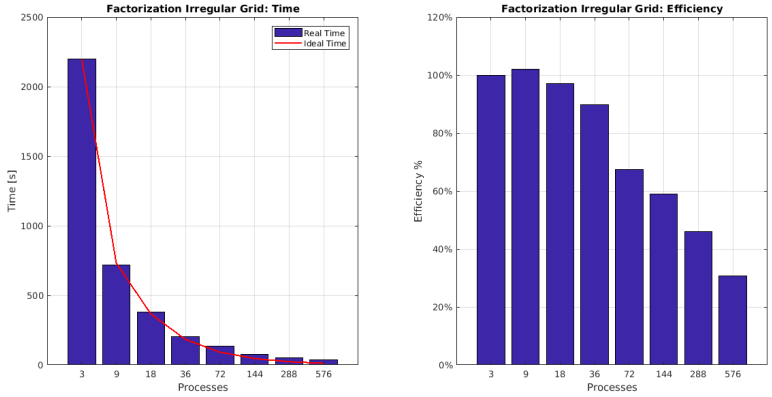

Figure 47: Irregular grids factorization.

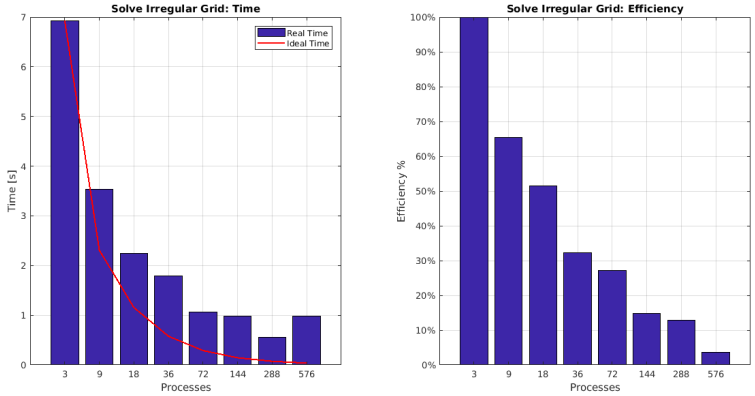

Figure 48: Irregular grids solver.

7 Scientific visualization

Once a linear system is solved, PETSc prints out data in different format: ASCII, VTK etc. Data visualization can be performed with several tools and functions.

7.1 Matlab data Visualization

Matlab can print results on regular grids (2D and 3D), with different functions:

- 2D domain: curve level and image with scaled color (Figure 49);
- 3D domain: isosurfaces and slice function (Figure 50).

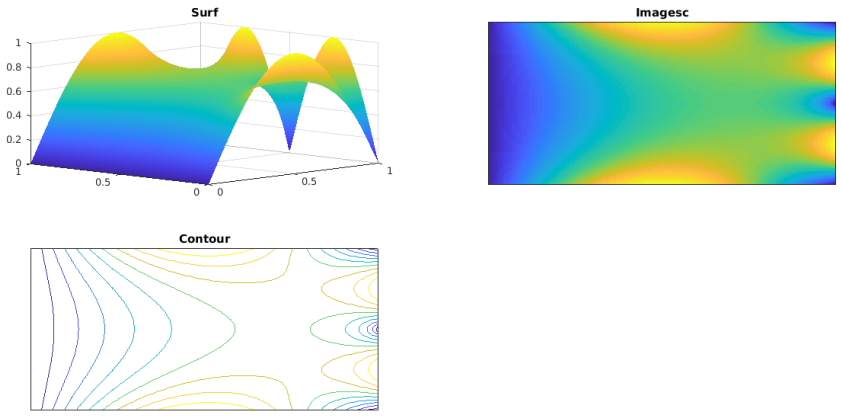

Figure 49: 2D domain visualization.

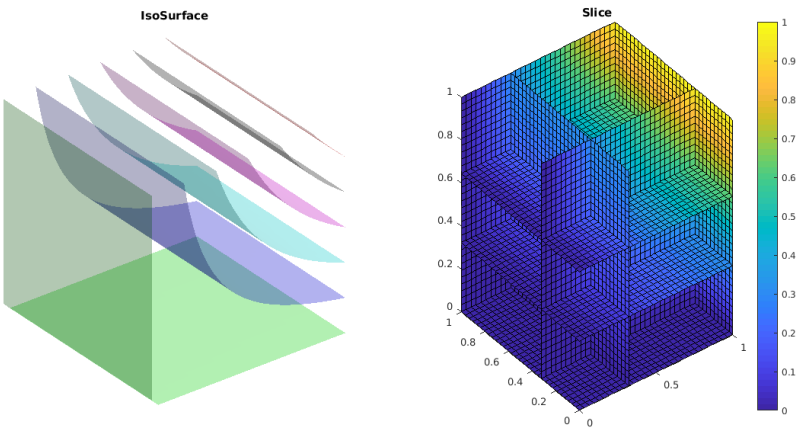

Figure 50: 3D domain visualization.

7.2 Workflow and Paraview visualization

PETSc can exploit other libraries in order to have a global environment where to solve and visualize PDEs.

Workflow presented in Figure 1 can be reviewed as Figure 51, where:

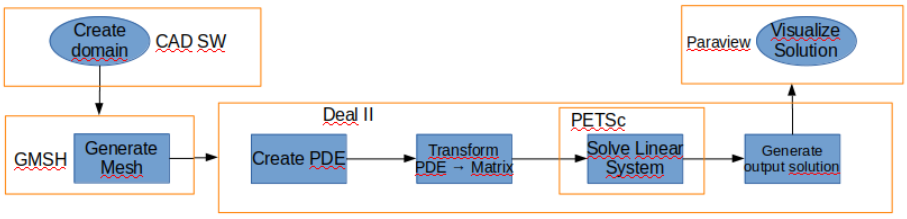

Figure 51: Workflow with SW.

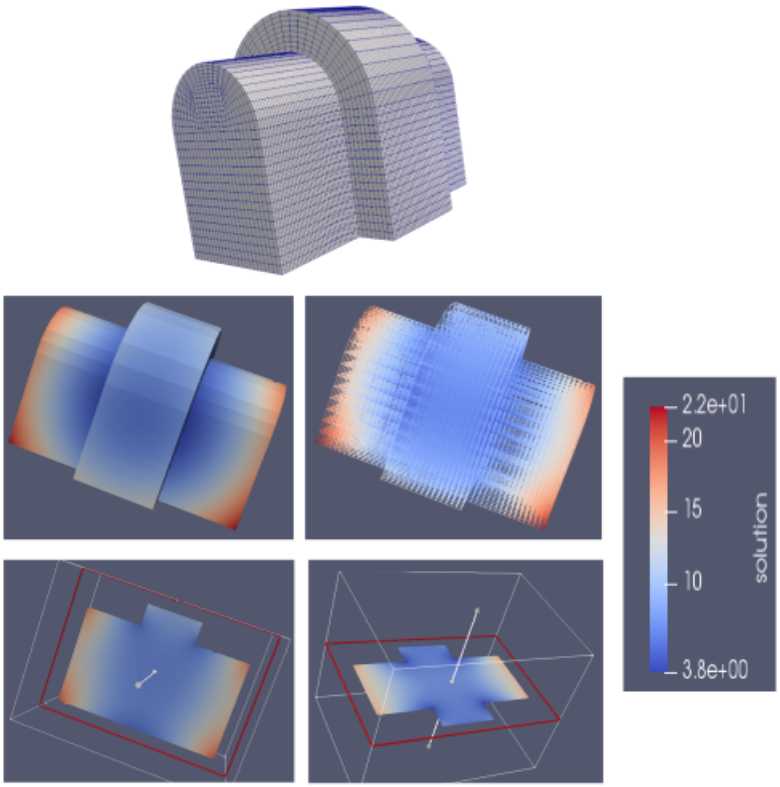

Figure 52: Laplace solver on a CAD model.

- *Gmsh*: reads a cad file and create a mesh on it;
- *dealii*: defines a PDE in variational form, transforms it into a linear system and solves it; then, prints the solution to a datafile;
- *ParaView*: reads a .vtk file printed by *dealii* and visualize the solution.

Following this workflow, Laplace equation can be solved and visualized on a CAD model, as shown in Figure 52.

8 Conclusion

Parallel implementations of direct and iterative methods for solving sparse linear systems have been compared on Marconi cluster and their performances have been analyzed in terms of efficiency and accuracy on different input conditions.

On 2D/3D regular grids, direct methods (SuperLU) have very good efficiency results; however, the elevated execution time for factorization makes this method not applicable for a single linear system; furthermore, LU matrices can involve memory limits. Only two operations are parallelized: factorization has very good efficiency results, thanks to data distribution and column permutation performed upstream; solver operation has a very small execution time but scalability performances are slightly worse, due to the overhead of the communications with respect to the computation.

For iterative methods, from a comparison of several preconditioner-solver combinations, *Block Jacobi - BICGSTAB* have resulted the most performing; they are general purpose preconditioner-solvers with very good performances in terms of execution time. Scalability results are moderately good: *Block Jacobi - BICGSTAB* efficiency decrease quickly, also due to the increased number of iterations.

Most expensive operations have been identified in *MatMult*, *MatSolve* and *VecDot*. The former has both computation and communication parts and its efficiency decrease with number of MPI communications increase and message length reduction; the latter is a massively parallel operation, with good efficiency results. Finally, *VecDot* has shown bad scalability results, only in part affected by the number of iterations and load balance among processes.

On large matrices, *Block Jacobi - BICGSTAB* still scales with 2304 processes (64 nodes) on Marconi cluster: granularity becomes coarser as coefficient matrix size is increased or number of processes is reduced.

In case of multiple r.h.s., direct methods are preferable only if high precision is required, otherwise iterative methods have still better performances (with both Block Jacobi and Hypre preconditioners); this result is interesting because, even if *Solve* operation of direct method has less computational operations than an iterative method, it works on more dense matrix and this leads to results just described. Two additional approaches also have been evaluated: column arrangement of r.h.s. in a matrix and block approach of coefficient matrix; only first approach has shown improvement on results while both have memory limits that have to be taken into account.

The error analysis comparison between direct and iterative methods has shown that iterative methods can reach the solution accuracy of direct methods, at the cost of number of iterations and consequently execution time. From a comparison of preconditioners in terms of accuracy, Hypre has given best results even if Block Jacobi has comparable results.

On irregular grids, the increased number of non-zeros leads, for both direct and iterative methods, to an increase of execution time.

Direct methods have similar results as regular grids, in terms of efficiency. Factorization has even better results, due to increased density of coefficient matrix; solver operation has similar results to regular grids case: execution time is very low but efficiency result is poor due to the overhead of communications with respect to computation.

Iterative methods have shown good scalability properties up to 144 processes and an efficiency decrease at 576 processes; the comparison among algorithms and preconditioners has given similar results to the regular grids one. Also on irregular grids, most expensive operations have been identified in *MatMult*, *MatSolve* and *VecDot*. First one has very poor scalability results at 576 processes due to number of MPI communications and message length, which are respectively three order more and one order less than regular grids ones; second one has superlinear scalability results, due to the kind of operation (massively parallel) and resources used (cache memory size). Finally, *VecDot* has shown very poor scalability

results, due to the number of reduction operations, load balance among processes and also the kind of operation itself. Granularity is well balanced till 36 processes, after that the overhead of MPI communications leads to bad scalability performances described.

The comparison with multiple r.h.s. has shown that direct methods have better performances than iterative ones, even if low error and *Hypre* preconditioner are used.

The analysis on matrix sparsity pattern has shown the impact of non-zero structure in terms of execution time (almost linear), iterations and approximation accuracy.

In conclusion, on regular grids iterative methods have shown good results in terms of scalability, accuracy and multiple r.h.s.; the use of a strong preconditioner, according to these results, does not seem necessary. Direct methods have shown very good results in terms of efficiency; however, the high execution time for the factorization makes this method valuable only if many linear systems have to be solved and a high accuracy on the solution is required, also considering that memory limits could be encountered. On irregular grids, iterative methods have worse performances, due to the irregular structure of the matrix that leads to an overhead of communication in some of the operations; in this case, when multiple systems have to be solved, direct methods should be used if memory size allows it. Otherwise, a strong preconditioner should be applied. It is anyway interesting that direct methods on irregular grids (compared to regular grids results) have an increased execution time but also benefits on efficiency; this is because the increased number of non-zero elements leads to a coarser granularity.

Section 7 has shown some tools (with *MATLAB* and *Paraview*) used for the visualization of the solution; in particular, *Paraview* handles irregular grids and allows performing and very customizable visualization.

PETSc library has shown very good performances, flexibility and easiness-of-use, since it offers many routines for iterative methods (and preconditioners) and the possibility to call some direct solvers libraries. All routines are available and accessible. PETSc performs data reading, MPI distribution and solution gathering and it also offers the possibility to print output in several formats. Furthermore, it gives the possibility to have both high and low level profiling on used methods. Finally, it can call some other libraries that allow to define PDE and solve it on domain.

Future directions

Some future developments of this work could be divided into four categories:

- *Numerical analysis*: study of the correlation between number of processes and iterations, in case of different preconditioners (Block-Jacobi, Hypre or None);
- *Performances analysis*: time-depending equation could be analyzed (through FreeFem++ or dealii) on different domains; furthermore, other libraries (such as Trilinos) offer different implementation of numeric algorithms for preconditioning and solving; these could be compared with PETSc performances;
- *High Performance Computing*: several HPC resources could be investigated (KNL on CINECA cluster, GPU-based resources etc.)
- *Algorithm development*: IDR(s) algorithm is not available on PETSc; thus, starting from Appendix B, it could be developed a parallelized version of this algorithm and exported in PETSc.

Appendix A BICGSSTAB

BICGSTAB algorithm is the transpose free version of the BCG, based on the Lanczos biorhogonalization.

The algorithm, implemented by PETSc routine KSPBICGS, is reported in Equation 12:

$$
\begin{array}{ll}
1. & r_0 = b - Ax_0 \ ; \ r_0^* \ arbitrary; \ r = r_0 \\
2. & initialize(p_0, \rho_{old}, \alpha, \omega_{old}, v, p) \\
while & (convergence) \\
3. & \rho = (r, r_0) \\
4. & \beta = (\rho/\rho_{old})/(\alpha/\omega_{old}) \\
5. & p = r - \omega_{old}\beta v + \beta p \\
6. & v = AK^{-1}p \\
7. & \alpha = \rho/(v, r_0) \\
8. & s = r - \alpha v \\
9. & t = AK^{-1}s \\
10. & d_1 = (s, t) \ ; \ d_2 = \|t\| \\
11. & x = x + \alpha p \\
12. & \omega = d_1/d_2 \\
13. & x = \alpha p + \omega s + x \\
14. & r = s - \omega t \\
15. & \|r\| \\
16. & \omega_{old} = \omega \\
17. & \rho_{old} = \rho \\
end & while
\end{array}
\tag{12}
$$

As reported in Equation 12, there are two *MatMult* and two *Matsolve* operation in line 6 and 9.

These operations, called PCApplyAB, solve a triangular system $y = K^{-1}p$ where K is the preconditioned matrix; then perform a matrix-vector product $v = Ay$.

VecDot operation (scalar product between two vectors) is performed twice, in line 3 and 7.

Then, line 10 performs the *VecDotNorm2* operation (a VecDot and a Norm operation).

Line 15 performs a norm operation (used for updating the residual and verifying the exit condition).

Line 14 and Line 8 performs a WAXPY ($W = aX + Y$) operation.

Line 11 performs an AXPY ($Y = aX + Y$) and Line 13 performs an AXPBYPCZ ($Z = aX + bY + cZ$) and both are collected in AXPBYCZ profiling voice.

Appendix B IDR

Induced Dimension Reduction (IDR) is an iterative algorithm for solving linear systems, based on Lanczos biorthogonalization; it can be considered as a generalization of BICGSTAB algorithm. It has been introduced by [8] who has developed the code in MATLAB and FORTRAN90.

PETSc has not implemented yet this algorithm, thus a preliminary version has been written. This code uses PETSc routines (when possible), otherwise it uses Blas routines. The code is working but it is not parallelized yet; anyway it is fully reported since it could be a baseline for a future development of IDR algorithm on PETSc.

```
void idrSolve(Mat A, Vec b, Vec xReal) {

PetscErrorCode ierr;
PetscInt       one = 1.0, zero = 0.0;
PetscViewer    viewer;
int            i,j,k;
PetscInt       sizeMat;
PetscScalar    alfa = 1, beta = 0;
PetscInt       s = 3; //shadows
PetscReal      tol = 1e-12;
PetscReal      angle = 0.7;
PetscReal      om = 1;
PetscReal      Alpha,Beta;
PetscInt       iter = 0;
PetscInt       maxIteration = 10000;
Mat            P,G,U,M;
PetscScalar    *aP, *aG, *aU, *aM, *ar, *af, *aGapp,
               *aUapp, *ax, *at, *axReal; //scalar from array
PetscScalar    *ac, *av;
PetscScalar    *deltaX;
Vec            x0; //initial solution
Vec            menox;
Vec            r,f; //residual
Vec            D; // Diagonal ones
Vec            Gapp, Uapp;
Vec            t;
Vec            v, vcopy;
Vec            c;
Vec            x;
KSP            ksp;     //Solver Context
PC             pc;      //Preconditioner Context
PetscReal      normb;
PetscReal      normr;
PetscReal      tolb;


KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetType(ksp,KSPPREONLY);
KSPSetOperators(ksp,A,A);
KSPGetPC(ksp,&pc);
PCSetType(pc,PCBJACOBI);
KSPSetUp(ksp);
KSPSetUpOnBlocks(ksp);

//Read sizeMat
ierr = VecGetSize(b,&sizeMat);

//Create Matrix and Vectors
MatCreateDense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,sizeMat,s,NULL,&P);
MatSetFromOptions(P);
VecCreate(PETSC_COMM_WORLD, &x0); VecSetSizes(x0,PETSC_DECIDE,sizeMat);
```

```
VecSetFromOptions(x0); VecSet(x0,zero);
VecCreate(PETSC_COMM_WORLD, &r); VecSetSizes(r,PETSC_DECIDE,sizeMat);
VecSetFromOptions(r);
VecCreate(PETSC_COMM_WORLD, &menox); VecSetSizes(menox,PETSC_DECIDE,sizeMat);
VecSetFromOptions(menox);
MatCreateDense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,sizeMat,s,NULL,&G);
MatSetFromOptions(G);
MatCreateDense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,sizeMat,s,NULL,&U);
MatSetFromOptions(U);
MatCreateDense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,s,s,NULL,&M);
MatSetFromOptions(M);
VecCreate(PETSC_COMM_WORLD, &D); VecSetSizes(D,PETSC_DECIDE,s);
VecSetFromOptions(D); VecSet(D,one);
VecCreate(PETSC_COMM_WORLD, &f); VecSetSizes(f,PETSC_DECIDE,s);
VecSetFromOptions(f); VecSet(f,zero);
VecCreate(PETSC_COMM_WORLD, &Gapp); VecSetSizes(Gapp,PETSC_DECIDE,sizeMat);
VecSetFromOptions(Gapp);
VecCreate(PETSC_COMM_WORLD, &Uapp); VecSetSizes(Uapp,PETSC_DECIDE,sizeMat);
VecSetFromOptions(Uapp);
VecCreate(PETSC_COMM_WORLD, &t); VecSetSizes(t,PETSC_DECIDE,sizeMat);
VecSetFromOptions(t);
VecCreate(PETSC_COMM_WORLD, &v); VecSetSizes(v,PETSC_DECIDE,sizeMat);
VecSetFromOptions(v);
VecCreate(PETSC_COMM_WORLD, &vcopy); VecSetSizes(vcopy,PETSC_DECIDE,sizeMat);
VecSetFromOptions(vcopy);
VecCreate(PETSC_COMM_WORLD, &c); VecSetSizes(c,PETSC_DECIDE,s);
VecSetFromOptions(c);
VecCreate(PETSC_COMM_WORLD, &x); VecSetSizes(x,PETSC_DECIDE,sizeMat);
VecSetFromOptions(x);

//Create P
ierr = PetscMalloc1(sizeMat*s,&aP);
for(i = 0; i < sizeMat*s;i++)
 aP[i] = rand()/(double)RAND_MAX;

//initialize x to x0
x = x0;

//norm and tolerance
ierr = VecNorm(b,NORM_2,&normb);
tolb = tol; /*tolb = tol*normb;*/

//Residual Calculation
menox = x;
ierr =  VecScale(menox, -1);

ierr = MatMultAdd(A,menox,b,r);
ierr = VecNorm(r,NORM_2,&normr);

//Create M
ierr = MatZeroEntries(M);
ierr = MatDiagonalSet(M,D,INSERT_VALUES);

//Extract Arrays
ierr = VecGetArray(r,&ar);
ierr = VecGetArray(f,&af);
ierr = VecGetArray(x,&ax);
ierr = MatDenseGetArray(M,&aM);
ierr = MatDenseGetArray(U,&aU);
ierr = MatDenseGetArray(G,&aG);
ierr = VecGetArray(Uapp,&aUapp);
ierr = VecGetArray(t,&at);
ierr = VecGetArray(v,&av);
ierr = VecGetArray(c,&ac);
ierr = VecGetArray(xReal,&axReal);

//Core CODE
while( (iter < maxIteration) && (normr > tolb) ) {
```

```
 cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
             1, s, sizeMat, alfa, ar,1, aP, sizeMat, beta, af, 1);
 for(k = 0; k < s-1; k++) {
 //Initialize ac
  for(i = k; i< s; i++)
   ac[i] = 0;
 //Solve M\f
  ac[k] = af[k]/aM[k*s+k];
  for(i = k+1; i < s; i++) {
   for(j = k; j < i; j++)   {
    ac[i] = ac[i]+(aM[j*s+i]*ac[j])/aM[i*s+i];
   }
   ac[i] = af[i]/aM[i*s+i]-ac[i];
  }
//Calculate v
  cblas_dcopy(sizeMat,ar,1,av,1);

  cblas_dgemv(CblasColMajor, CblasNoTrans,
              sizeMat, s-k,  -1, aG+sizeMat*k,sizeMat, ac+k,1, 1, av, 1);

//PRECONDITIONING
  ierr = VecRestoreArray(v,&av);
  VecCopy(v,vcopy);
  ierr = PCApply(pc,vcopy,v);
  ierr = VecGetArray(v,&av);

//Compute new U
  cblas_dgemv(CblasColMajor, CblasNoTrans,
              sizeMat, s-k,  +1, aU+sizeMat*k,sizeMat, ac+k,1, om, av, 1);

  cblas_dcopy(sizeMat,av,1,aU+sizeMat*k,1);
//Compute new G
  cblas_dcopy(sizeMat,aU+k*sizeMat,1,aUapp,1);
  VecRestoreArray(Uapp,&aUapp);
  MatMult(A,Uapp,Gapp);
  VecGetArray(Gapp,&aGapp);
  VecGetArray(Uapp,&aUapp);
  cblas_dcopy(sizeMat,aGapp,1,aG+sizeMat*k,1);
//BiOrthogonalise the new basis vectors
  for(i = 0; i < k; i++) {
   Alpha = cblas_ddot(sizeMat,aP+sizeMat*i,1,aG+sizeMat*k,1)/aM[i*s+i];
   cblas_daxpy(sizeMat,-Alpha,aG+i*sizeMat,1,aG+k*sizeMat,1);
   cblas_daxpy(sizeMat,-Alpha,aU+i*sizeMat,1,aU+k*sizeMat,1);
  }
//New column of M
  cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
              1, s-k, sizeMat, alfa, aG+k*sizeMat,1,
              aP+k*sizeMat, sizeMat, 0, aM+k*s+k, 1);
//Make r ortoghonal
  Beta = af[k]/aM[k*s+k];
  cblas_daxpy(sizeMat,-Beta,aG+k*sizeMat,1,ar,1);
  cblas_daxpy(sizeMat,Beta,aU+k*sizeMat,1,ax,1);
  normr = calcNorm(ar,sizeMat);
//New f = P'*r
  if(k < s-1)
   cblas_daxpy(s-k-1,-Beta,aM+k*s+k+1,1,af+k+1,1);
 }
//Calculate t = A*v
 cblas_dcopy(sizeMat,ar,1,av,1);
 VecRestoreArray(v,&av);

//PRECONDITIONING
 VecCopy(v,vcopy);
 PCApply(pc,vcopy,v);

//Calculate
 MatMult(A,v,t);
 VecGetArray(v,&av);
```

```
//Calculate Omega
 om = omega(at,ar,angle,sizeMat);
//Calculate r and x
 cblas_daxpy(sizeMat,-om,at,1,ar,1);
 cblas_daxpy(sizeMat,+om,av,1,ax,1);
 normr = calcNorm(ar,sizeMat);
 iter++;
 }
return;
}

PetscReal omega(PetscScalar *at, PetscScalar *ar,
                PetscReal angle, PetscInt sizeMat) {
 PetscReal ns,nt,ts;
 PetscReal rho,om;
 ns = calcNorm(ar,sizeMat);
 nt = calcNorm(at,sizeMat);
 ts = cblas_ddot(sizeMat,at,1,ar,1);
 rho = fabs(ts/(nt*ns));
 om = ts/(nt*nt);
 if(rho < angle)
  om = om*angle/rho;
 return om;
}

PetscReal calcNorm(PetscScalar *array, PetscInt size) {
 int i;
 PetscReal somma = 0;
 for(i = 0; i < size; i++)
  somma = somma + array[i]*array[i];
 return sqrt(somma);
}
```

Some considerations on the code and its performances:

- Preconditioning is applied with PETSc routines;
- Operations with A matrix are applied with PETSc routines;
- Operations with dense matrices (G, U, M) are applied with BLAS routines, due to the fact that only some portions of these matrices are used each time;
- Shadow space (s variable) has been fixed to 3 and omega angle has been fixed to 0.7;
- A preliminary profiling has been performed on input data defined in Table 6 with one process only. Total solver time is 21 seconds, compared to 17 seconds of BICGSTAB (with the same Block-Jacobi preconditioner); number of iterations is 56 compared to 128 of BICGSTAB; error on solution are comparable.

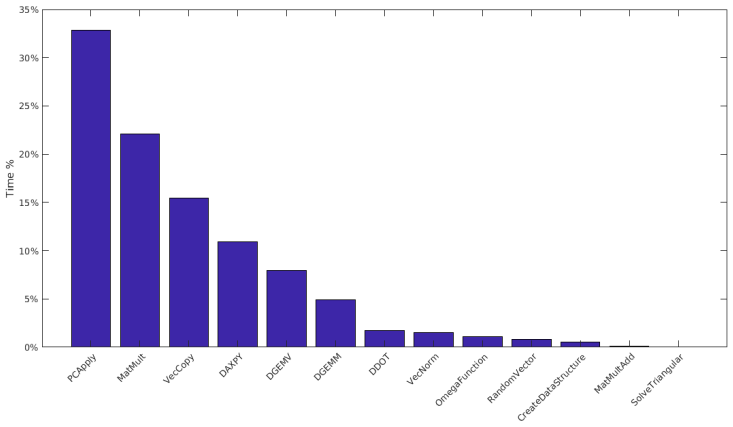

Figure 53: IDR Profiling.

A simple profiling of the operations give results as reported in Figure 53; *PCapply* and *MatMult* are, as expected, most expensive operations. *VecCopy* has 15 % and this could be a point to be optimized.

References

[1] Lawrence C. Evans *Partial Differential Equations.* American Mathematical Society, 1998

[2] Youseef Saad *Iterative Methods for Sparse Linear Systems.* American Mathematical Society, 1998

[3] Gene H. Golub, Charles F. Van Loan *Matrix Computation* The Johns Hopkins University Press, 1983

[4] George Karypis and Vipin Kumar *Multilevel graph partitioning schemes.* 10th Intl. Parallel Processing Symposium, pp. 314 - 319, 1996.

[5] `http://mumps.enseeiht.fr/index.php?page=doc`

[6] `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/ug.pdf`

[7] Laura Grigori *Sparse linear solvers: iterative methods, sparse matrix-vector multiplication, and preconditioning.* March 2015

[8] Peter Sonneveld, Martin B. Van Gijzen *IDR(s): A family of simple and fast algorithms for solving large nonsymmetric system of linear equations.* 2008 Society for Industrial and Applied Mathematics, SIAM J. SCI. COMPUT.Vol. 31, No. 2, pp. 10351062

[9] Zhu, Yao and Sameh, Ahmed H. *How to Generate Effective Block Jacobi Preconditioners for Solving Large Sparse Linear Systems*, Advances in Computational Fluid-Structure Interaction and Flow Simulation: New Methods and Challenging Computations, 2016, Springer International Publishing, 231–244

[10] Alfio Borz, *Introduction to Multigrid Methods*

[11] Sandro Salsa *Equazioni a derivate parziali - Metodi, modelli e applicazioni.* Springer-Verlag Italia 2016

[12] Leah Edelstein-Keshet *Mathematical Models in Biology.* Society For Industrial & Applied Mathematics,u.s., 2005.

[13] Martin Burger, Luis Caffarelli and Peter Markowich *Partial differential equation models in the socio-economic sciences.* Royal Society Publishing, 2014

[14] J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, A. YarKhan, *Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems*, 2015

[15] H. Anzt, M. Gates, J. Dongarraa, M. Kreutzer, G. Welleind, M. Khler, *Preconditioned Krylov solvers on GPUs*, 2017.

[16] `https://www.top500.org/list/2017/06/`

[17] `https://math.nist.gov/MatrixMarket/`

[18] `https://www.mcs.anl.gov/petsc/`

[19] `https://www.paraview.org/`

[20] `https://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods`

[21] `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

[22] `http://gmsh.info/`

[23] `http://www.freefem.org/`

[24] http://www.dealii.org/

[25] Yogish Sabharwal, *Introduction to parallel programming in OpenMP*,
https://www.youtube.com/channel/UCNp-uk36t-bnvHr3A_snQtg

[26] P. R. Amestoy, T. A. Davis, and I. S. Duff., *An approximate minimum degree ordering algorithm.*, SIAM Journal on Matrix Analysis and Applications, 17:886905, 1996

[27] M. Heath, E. Ng, and B. Peyton., *Parallel algorithms for sparse linear systems.*, SIAM Review, 33:420460, 1991.

[28] E. Rothberg., Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization., PhD thesis, Dept. of Computer Science, Stanford University, December 1992.

[29] CINECA, *Introduction to MPI+OpenMP hybrid programming*, Summer School on Parallel Computing.

[30] R. A. van de Geijn, A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, *Parallel Implementation of BLAS: General Techniques for Level 3 BLAS**,

[31] http://www.netlib.org/blas/

[32] G. Patané, *Laplace-Beltrami operator and harmonic equation*, 2016

[33] A. Galizia, *Design of Parallel Image Processing Libraries for Distributed and Grid Environments*, 2008.

[34] L. Null, J. Lobur, *Computer organization and architecture*, 2014

[35] K. McKinley, *Lectures/Storage*.

[36] https://en.wikipedia.org/wiki/Memory_hierarchy

[37] A. Quarteroni, *Modellistica Numerica per Problemi Differenziali*, 2016

[38] S. Ristov, R. Prodan, M. Gusev, K. Skala *Superlinear Speedup in HPC Systems: why and when?*, 2016

[39] N. Gould, Y. Hu, J. Scott, *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems*, 2005

[40] S. Cammarasana, A. Clematis, A. Galizia, G. Patané, *High Performance Computing for the Efficient Solution of PDEs on arbitrary domains*, 2018.

[41] Jack J. Dongarra and Michael A. Heroux and Piotr Luszczek, *HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems?*, 2015.

[42] Bavier, Eric and Hoemmen, Mark and Rajamanickam, Sivasankaran and Thornquist, Heidi, *Amesos2 and Belos: Direct and Iterative Solvers for Large Sparse Linear Systems*, 2012.

[43] Hartwig Anzt, Jack Dongarra, Moritz Kreutzer, Gerhard Wellein, Martin Khler, *Efficiency of General Krylov Methods on GPUs – An Experimental Study*, 2016

[44] Xiaoye S. Li, James W. Demmel, *SuperLU DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, 2003

[45] OlafSchenk, Klaus Grtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, 2003