Parallel meshing, discretization and computation of flow in massive discrete fracture networks

Original

Availability:
This version is available at: 11583/2734797 since: 2019-12-19T17:09:06Z

Publisher:
SIAM

Published
DOI:10.1137/18M1228736

Terms of use:
openAccess
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)
PARALLEL MESHING, DISCRETIZATION AND COMPUTATION OF FLOW IN MASSIVE DISCRETE FRACTURE NETWORKS * 
S. BERRONE†, S. SCIALÔ†, AND F. VICINI† 
June 6, 2019

Abstract. In the present work an MPI parallel implementation of an optimization-based approach for the simulation of underground flows in large discrete fracture networks is proposed. The software is capable of parallel execution of meshing, discretization, resolution and post-processing of the solution. We describe how optimal scalability performances are achieved combining high efficiency in computations to an optimized use of MPI communication protocols. Also, a novel graph-topology for communications, called multi-Master approach, is tested, allowing for high scalability performances on massive fracture networks. Strong scalability and weak scalability simulations on random networks counting order of $10^5$ fractures are reported.

Key words. Parallel scalability, MPI, Discrete Fracture Networks, Single-phase flows, PDE constrained optimization

AMS subject classifications. 65N30, 65N15, 65N50, 65J15, 68U20, 68W10, 68W40, 86-08

1. Introduction. It is well known that underground flow simulations are particularly demanding from a computational point of view, mainly as a consequence of the size and geometrical complexity of computational domains of interest for practical applications. Moreover, due to the lack of direct measurements of subsoil characteristics, input data for simulations are typically given as probability distributions, thus demanding a large number of costly computations to derive reliable statistics on quantities of interest [17, 21, 24, 7].

The subsoil can be regarded as a porous material crossed by a network of intersecting fractures. Among the different models to describe the subsoil (see e.g. [40]), Discrete Fracture Network (DFN) models only represent underground fractures, neglecting the contribution of the surrounding rock matrix, such that flow can only occur through fractures and fracture intersections. Differently from homogenization techniques, DFN models explicitly represent the fractures and are thus capable of reproducing the topology of the fracture network and preferential pathways for the flow, thus being particularly suitable for the simulation of dispersion phenomena [37, 32, 41, 10]. Fractures in a DFN are modeled as intersecting planar polygons forming an extremely challenging computational domain, usually characterized by an intricate system of intersections. Further, DFNs present geometrical features at different scales, as e.g. small fractures intersecting with large faults, fractures intersecting with narrow angles or co-existence of very small and very large fracture intersections. This geometrical complexity and multi-scale nature poses severe constraints on the meshing strategies. Standard discretization techniques, based on the Finite Element Method (FEM), require a mesh conforming to the intersections between fractures in order to correctly enforce matching conditions at the interfaces. Meshes conforming to all the geometrical features in a DFN can not be generated for networks of practical interest, or would introduce such a large number of elements to make the resolution...
of the corresponding discrete problem unaffordable. Further, a single simulation for a given geometry is not sufficient for applications, as it is the case for time dependent simulations or for uncertainty quantification purposes.

Next to recent works proposing advanced tools to produce conforming meshes for discrete fracture networks, [31, 41, 26], a large number of different approaches has been presented, aimed at relaxing the geometrical constraints on the mesh, thus overcoming the complexity of DFN simulations. We will define as partially conforming those DFN meshes in which fracture intersections do not cross mesh elements, as in a conforming mesh, but there is no matching at the intersections among element vertexes of the mesh of different fractures (see [9, Figs. 1,2]); and as non-conforming those meshes in which elements are free to arbitrarily cross fracture intersections (see [9, Figs. 3,17]). In [29] a survey on several unconventional numerical schemes for stationary DFN flow simulations is presented. In [20, 42] the DFN flow problem is dimensionally reduced to a mono-dimensional problem, whereas in [44] mixed finite elements are used on meshes partially conforming at fracture intersections and mortaring is used to enforce the required matching conditions. Polygonal methods have also been suggested as a way to easily generate conforming or partially conforming meshes: in [2, 3, 28, 27] the Virtual Element Method (VEM) is used for fracture networks and in [4] for fracture/matrix coupling, whereas [1] proposes the use of mimetic finite differences and [18] of Hybrid High Order Methods for fracture/matrix flow coupling. To completely overcome mesh related complexities in DFN simulations some authors suggest the use of non-conforming meshes. In this context, the use of the eXtended Finite Element Method (XFEM) has been proposed by many authors, see [25], to allow for irregular solutions within mesh elements, along with other techniques based e.g. on the cutFEM [16] or on Lagrange multipliers [35] to enforce conditions at the interfaces for fracture/matrix coupling. In Embedded Discrete Fracture Matrix (EDFM) models [36, 38], only the fractures exceeding a certain threshold size are explicitly represented, the others being homogenized, and the fractures and the porous matrix are represented on different computational meshes, adding suitable fracture/matrix connections for the coupling. Different approaches aimed at reducing cost and complexity of DFN simulations are based on a graph representation of the network of fractures, as proposed by [46, 33], also using graph theory tools [30] or machine learning [51] to estimate the flow.

The present work focuses on a novel approach for flow computations in large DFNs, first proposed in [11] and based on numerical optimization. The method allows the use of non-conforming meshes and the coupling conditions at fracture intersections are enforced through the minimization of a properly designed cost functional. The functional expresses the error in the fulfillment of the interface conditions, and the solution is obtained as the minimum of this functional constrained by the equations describing the flow on each fracture [12, 14]. One of the key aspects of the method resides in the fact that the minimization process requires to iteratively solve local and almost independent linear systems, each defined on a fracture of the network, and these fracture-local problems only need to share information at the interfaces. This allows a natural parallel implementation, with a high scalability potential.

A parallel implementation of this optimization-based method is reported in [15]. The algorithm is based on the MPI protocol for the communications among parallel processes and uses a Master/Slave topology to handle communications. According to this, the computing processes (Slaves) can not directly share the required data, as all communications flow through the Master process. The mentioned reference shows how this topology can reduce the number of communications in large networks. The
efficiency of the parallel algorithm is tested on networks with a number of fractures of the order of $10^3$, envisaging the use of an improved communication topology to handle larger networks. For this approach “a posteriori” error estimates have been derived in [5] and applied for adaptive mesh refinement to large scale DFNs in [6]. Another parallel code for flow simulations in DFNs is proposed in [39], based mixed-hybrid finite elements on conforming meshes. Due to the use of conforming meshes, the main focus is on the parallel mesh generation and assembly of the discrete matrices, thus differing from both the approach in [15] and the one described here.

The present work proposes a new implementation of the optimization-based approach for flow simulations in large DFNs. In this code, the meshing of the domain, the assembly and the resolution of the discrete problem, as well as other pre-resolution and post-resolution tasks are implemented allowing for parallel computing on distributed memory architectures using MPI, taking full advantage of the peculiarities of the optimization approach. An effective organization of linear algebra operations and the use of functions for MPI communications capable of reducing the overhead related to communications at each iteration of the resolution process, yield optimal scalability performances to the proposed code. Further a multi-Master topology is implemented with a hierarchy of Master processes in order to remove possible bottlenecks caused by the overloading of a single process handling the communication phases of a large number of parallel computing processes. This extends optimal scalability properties to a large range of numbers of parallel processes, thus allowing to effectively and efficiently solve problems on extremely large networks using a strong parallel approach. Numerical examples on networks counting order of $10^5$ fractures and using up to 128 parallel processes are used to document the performances of the code.

The structure of the manuscript is as follows: Section 2 describes the optimization based formulation of the DFN problem, which is presented in detail in the following Section 3. Parallel performances of the code are shown in Section 4, and finally some conclusions are reported in Section 5.

2. Optimization formulation. Let us briefly recall the optimization based formulation of the DFN problem; full details can be found in [14] and references therein mentioned. Let $\Omega$ denote a fracture network, $\Omega := \bigcup_{i=1,...,I} F_i$, with each of the $F_i$ representing a planar polygon in the three dimensional space, resembling one of the fractures in the network. Fractures are surrounded by an impervious matrix, such that flow only occurs in the fractures and through fracture intersections. Fracture intersections, also called traces, are denoted by $S_m$, $m = 1, \ldots, M$, and we assume that each trace is given by the intersection of exactly two fractures, i.e. $S_m = F_i \cap F_j$. There is a map between each trace index and the couple of fracture indexes, denoted by $I_S : [1, \ldots, M] \mapsto [1, \ldots, I]^2$ and defined by $I_S(m) = (i, j)$ with $i < j$ such that $F_i \cap F_j = S_m$. We also define $L_S(m) := i$ and $I_S(m) := j$. Let us further indicate by $S$ the set of all the traces in the network and by $S_i$ the set of traces on $F_i$. The boundary $\partial \Omega$ of $\Omega$ is split in a Dirichlet part $\Gamma_D \neq \emptyset$ and a Neumann part $\Gamma_N = \partial \Omega \setminus \Gamma_D$. Functions $b^D$ and $b^N$ prescribe Dirichlet and Neumann boundary conditions on $\Gamma_D$ and $\Gamma_N$, respectively. The boundary of each fracture $\partial F_i$ can be consequently split in a Dirichlet part $\Gamma_i^D := \partial F_i \cap \Gamma_D$ and a Neumann part $\Gamma_i^N := \partial F_i \cap \Gamma_N$ with boundary conditions given by functions $b_i^D := b^D|_{\Gamma_i^D}$ and $b_i^N := b^N|_{\Gamma_i^N}$, respectively.

2.1. The continuous problem. The distribution of the hydraulic head $H$ in a DFN is governed by the Darcy’s law, which can be stated in weak form as follows:
for \( i = 1, \ldots, I \), given the functional spaces
\[
V_i = H^1_0(F_i) = \{ v \in H^1(F_i) : v|_{\Gamma_{iD}} = 0 \}
\]
and
\[
V^D = H^1_0(F_i) = \{ v \in H^1(F_i) : v|_{\Gamma_{iD}} = b^D_i \},
\]
find \( H_i := H_{|F_i} \in V^D_i \) such that, \( \forall v \in V_i \),
\[
(2.1) \quad \int_{F_i} K \nabla H_i \nabla v \, d\Omega = \int_{F_i} q_i v \, d\Omega + \int_{\Gamma_{iN}} b^N_i v \, d\gamma + \sum_{S_m \in S_i} \int_{S_m} \left[ \frac{\partial H_i}{\partial \nu_{S_m}} \right] v|_{S_m} \, d\gamma
\]
and, \( \forall m = 1, \ldots, M \), with \( (i, j) = I_S(m) \)
\[
(2.2) \quad H_i|_{S_m} - H_j|_{S_m} = 0,
\]
\[
(2.3) \quad \left[ \frac{\partial H_i}{\partial \nu_{S_m}} \right]_{S_m} + \left[ \frac{\partial H_j}{\partial \nu_{S_m}} \right]_{S_m} = 0,
\]
where \( K_i \) is a uniformly positive definite tensor representing the fracture hydraulic conductivity, \( q_i \) is a source term and \( \left[ \frac{\partial H_i}{\partial \nu_{S_m}} \right]_{S_m} \) is the jump of the co-normal derivative along the unit vector normal to \( S_m \) with a fixed direction on \( F_i \). Equations (2.2)-(2.3) represent the matching conditions at fracture intersections, imposing the continuity of the solution and the balance of fluxes, respectively.

Let us introduce the quantity
\[
U^m_i := \left[ \frac{\partial H_i}{\partial \nu_{S_m}} \right]_{S_m} + \alpha H_i|_{S_m}, \quad U^m_i \in H^{-\frac{1}{2}}(S_m)
\]
for \( \alpha > 0 \), and the quadratic functional \( J \) defined as:
\[
J(H, U) = \sum_{m=1}^M \| H_i|_{S_m} - H_j|_{S_m} \|_{H^1(S_m)}^2 + \left\| U^S_m + U^J_m - \alpha (H_i|_{S_m} + H_j|_{S_m}) \right\|_{H^{-\frac{1}{2}}(S_m)}^2,
\]
being \( U \) the function given by the cartesian product of functions \( U^m_i \), for \( i = 1, \ldots, I \) and \( S_m \in S_i \), \( U \in [H^{-\frac{1}{2}}(S)]^I \). Then problem (2.1)-(2.3) is equivalent to the following PDE-constrained optimization problem [12]:
\[
(2.5) \quad \min_{U \in [H^{-\frac{1}{2}}(S)]^I} J(H, U)
\]
such that, \( \forall i = 1, \ldots, I \), \( \forall v \in V_i \):
\[
(2.6) \quad \int_{F_i} K_i \nabla H_i \nabla v \, d\Omega + \alpha \sum_{S_m \in S_i} \int_{S_m} H_i|_{S_m} v|_{S_m} \, d\gamma = \int_{F_i} q_i v \, d\Omega + \int_{\Gamma_{iN}} b^N_i v \, d\gamma + \sum_{S_m \in S_i} \int_{S_m} U_i^m v|_{S_m} \, d\gamma.
\]

**2.2. The discrete problem.** Let us now introduce on each fracture \( F_i \) of the DFN \( \Omega \) a triangulation \( T_i \) and also, on each trace of each fracture \( F_i \), a discretization \( T_i^m \). We remark that each of these meshes can be defined independently of all the
The functional (2.9) is re-written in compact algebraic form as follows:

\[ A_i h_i = q_i + B_i u_i, \quad i = 1, \ldots, I \]

where \( A_i \in \mathbb{R}^{N_i \times N_i} \) is defined by

\[
(A_i)_{k\ell} = \int_{F_i} K \nabla \varphi_{i,k} \nabla \varphi_{i,\ell} \, d\Omega + \alpha \sum_{m=1}^{M_i} \int_{S_m} \varphi_{i,k|S_m} \varphi_{i,\ell|S_m} \, d\gamma,
\]

matrix \( B_i \in \mathbb{R}^{N_i \times N_S} \), collects the integrals of the product of basis functions \( \{ \varphi_{i,k|S_m} \} \), \( k = 1, \ldots, N_i \), with \( \{ \psi_{i,k}^m \} \), \( k = 1, \ldots, N^m_i \), and \( q_i \in \mathbb{R}^{N_i} \) is the vector deriving from the discretization of forcing terms and boundary conditions. The block-diagonal matrix \( A = \text{diag}(A_1, \ldots, A_I) \in \mathbb{R}^{N^F \times N^F} \) is then formed from matrices \( A_i \), whereas matrix \( B \in \mathbb{R}^{N^F \times N^S} \), is defined as \( B := (B_1 R_1; \ldots; B_I R_I) \), i.e. is obtained collecting column-wise matrices \( B_i, R_i \) \( \forall i = 1, \ldots, I \), where matrix \( R_i \) acts extracting from \( u \) the DOFs corresponding to \( u_i \); now equations (2.7) can be compactly re-written as

\[
h h = q + B u
\]

where \( q \) is obtained grouping vectors \( q_i \) columnwise.

The functional is obtained replacing, in equation (2.4) functions \( H \) and \( U \) with their discrete versions and using \( L^2(S_m) \) norms in place of the \( H^1(S_m) \) and \( H^{-1}(S_m) \) norms:

\[
J(h,u) = \frac{1}{2} \sum_{i=1}^{I} \sum_{S_m \in s_i} \left( \int_{S_m} \sum_{k=1}^{N_i} h_{i,k} \varphi_{i,k|S_m} - \sum_{k=1}^{N_i} h_{j,k} \varphi_{j,k|S_m} \right)^2 \, d\gamma + \int_{S_m} \sum_{k=1}^{N^m_i} u_{i,k}^m \psi_{i,k}^m + \sum_{k=1}^{N^m_i} u_{j,k}^m \psi_{j,k}^m - \alpha \sum_{k=1}^{N^m_i} h_{i,k} \varphi_{i,k|S_m} - \alpha \sum_{k=1}^{N^m_i} h_{j,k} \varphi_{j,k|S_m} \right)^2 \, d\gamma.
\]

The functional (2.9) is re-written in compact algebraic form as follows:

\[
J(h,u) := \frac{1}{2} \left( h^T G^h h - \alpha h^T B^h u - \alpha u^T B^u h + u^T G^u u \right)
\]

where matrix \( G^h \in \mathbb{R}^{N^F \times N^F} \) collects integrals on the traces of the product between functions \( \{ \varphi_{i,k|S_m} \}, k = 1, \ldots, N_i, i = 1, \ldots, I \); matrix \( G^u \in \mathbb{R}^{N^S \times N^S} \) collects integrals of the product between functions \( \{ \psi_{i,k}^m \}, k = 1, \ldots, N^m_i \), and matrices \( B^h \) and
\( B^u, \ B^h = (B^u)^T \in \mathbb{R}^{N_F \times N_S} \) collect integrals of the product between basis functions of \( h \) and of \( u \).

The discrete optimization problem is then stated as follows:

\[
\begin{align*}
\min & \quad J(h, u) \\
\text{s.t.} & \quad A h - B u = q.
\end{align*}
\]

Optimality conditions for this problem are given by the system of equations

\[
\begin{align*}
\mathcal{A} h &= q + B u \\
\mathcal{A}^T p &= G^h h - \alpha B^h u \\
0 &= B^T p + G^u u - \alpha B^u h = \nabla J(u),
\end{align*}
\]

which corresponds to the following saddle point problem:

\[
\mathcal{A} = \begin{pmatrix}
G^h & -\alpha B^u \\
-\alpha B^h & G^u \\
\mathcal{A} & B \\
\end{pmatrix} : \quad \mathcal{A} \begin{pmatrix}
h \\
u \\
-p
\end{pmatrix} = \begin{pmatrix}
0 \\
0 \\
q
\end{pmatrix}
\]

**Proposition 2.1.** Matrix \( \mathcal{A} \) in equation (2.15) is non singular, and the unique solution \((h^*, u^*, p^*)\) of (2.15) is the minimizer of (2.11).

The proof of Proposition 2.1 can be easily found as a particular case of the proof given in [13].

It is possible to obtain an unconstrained minimization problem equivalent to (2.11) by formally replacing \( h = \mathcal{A}^{-1} (B u + q) \) in (2.10). To this end, we set:

\[
\hat{J}(u) = \frac{1}{2} u^T (B \mathcal{A}^{-T} G^h \mathcal{A}^{-1} B + G^u - \alpha B^T \mathcal{A}^{-T} B - \alpha B^T \mathcal{A}^{-1} B) u
\]

\[
+ q^T \mathcal{A}^{-T} (G^h \mathcal{A}^{-1} B + G^u - \alpha B) u
\]

\[
= \frac{1}{2} u^T \hat{G} u + \hat{q} u,
\]

where \( \hat{B} := B^h = (B^u)^T \), and \( \hat{G} \) is symmetric positive definite, see [15] for more details. Then, solving problem (2.11) is equivalent to solve equation \( \hat{G} u = -\hat{q} \), i.e. minimizing the unconstrained functional (2.16) via, e.g., a gradient scheme. What is of interest for the present work is that this does not require the direct computation of matrix \( \hat{G} \), but only involves the resolution of fracture-local problems. Indeed, the application of a gradient scheme to solve \( \hat{G} u = -\hat{q} \) corresponds to the following algorithm: starting from a tentative \( u^0 \), solve in cascade, for \( n = 0, 1, \ldots \):

\[
\begin{align*}
\mathcal{A} h &= B u^n \\
\mathcal{A}^T p &= G^h h - \alpha B^h u^n \\
g^n &= B^T p + G^u u - \alpha B^u h \\
u^{n+1} &= u^n + \lambda^n g^n
\end{align*}
\]

where \( g^n \) is the gradient direction \( \nabla \hat{J}(u^n) \) at iteration \( n \) and \( \lambda^n \) is computed through an exact line search, [15]. Given the block-diagonal structure of matrix \( \mathcal{A} \), problems (2.17)-(2.18) can be solved independently on each fracture of the DFN. A conjugate gradient scheme will actually be used to solve the optimization problem, and thus a
PARALLEL FLOW SIMULATIONS IN DISCRETE FRACTURE NETWORKS

PARALLEL FLOW SIMULATIONS IN DISCRETE FRACTURE NETWORKS

(a) Master-Slave Topology
(b) Tree Topology

Figure 1: MPI Topologies

A key aspect of the program is the parallel environment topology. A MasterSlave topology has been chosen in place of a PointToPoint communication approach as also proposed in [15]. In a MasterSlave topology (Figure 1a), one of the parallel processes (the Master) is uniquely devoted to receive and send the data that the other processes (called computing processes or Slaves) need to share. Point-to-point communications would require every process to directly share data with a number of other processes, which depends on the connectivity of the network and on its partitioning. For example, when large faults are present in a DFN, entirely crossing the network, if point-to-point communications are implemented, almost all the processes might need to share data among each other, thus yielding a growth of the number of communications proportional to the square of the number of processes. On the other side, with a MasterSlave topology the number of communications always scales like the number of processes, independently of the geometry of the network.

The MasterSlave approach has the drawback of a possible overloading of the Master process when a large number of parallel processes is used, thus creating unwanted bottlenecks. For this reason a new tree-type topology is implemented, allowing to handle a hierarchy of Master processes, with multiple bottom-level Master processes up to a single top-level Master process, as sketched in Figure 1b, where one top-level and three bottom level Master processes are used. This configuration is called Multi-Master communication topology. Now, each bottom-level Master process manages the communication of a fraction of the computing processes and the highest level Master process handles the communications of the lower-lever Masters.
The whole code can be subdivided in seven different tasks, as follows: 1) DFN geometry import; 2) main connected component computation; 3) main connected component distribution among parallel processes; 4) mesh generation; 5) discrete problem matrices assembly; 6) discrete problem resolution; 7) solution post-processing and export.

3.1. DFN geometry import. Typically a fracture network is described as a set of fractures, identified by the coordinates of their vertexes in the three-dimensional space. Since geometry is provided in a single file, no parallel implementation is used for this phase. Both textual and binary input files are supported. If a bounding box is specified, fractures are cut accordingly, storing information on the fracture edges lying on each face of the box. Boundary conditions and other properties of the fractures are imported at this phase. All the data on the fractures are then communicated to all the parallel processes.

3.2. Main connected component computation. Usually, fracture networks are stochastically generated, thus the imported “raw” geometrical data might contain fractures or networks of fractures disconnected from the main connected component. The main connected component is defined as the largest set of connected fractures or networks of fractures disconnected from the main connected component.

Each fracture of the raw imported network is labeled with an integer \( i \) ranging between one to the total number of imported fractures \( I_T \). Then, fracture indexes are split among the \( n_P \) + 1 parallel processes \( P_k, k = 0, \ldots, n_P \), and sets \( I_k \) of fracture indexes are assigned to processes \( P_k \), balancing the quantity \( X_k = \sum_{i \in I_k} I_T - (i + 1) \), such that \( X_k \sim \frac{I_T(I_T+1)}{2(n_P+1)} \). Each process \( P_k \) builds a local adjacency matrix \( A_k^D \) by checking if each of the fractures \( F_i, i \in I_k \) has intersections with fractures \( F_j, j = i + 1, \ldots, I_T \). Thus, balancing the quantity \( X_k, k = 0, \ldots, n_P \) corresponds to balance the number of checks that each process has to do.

The computation of the main connected component is then performed in parallel. On each process \( P_k \), a local connected component is computed and stored in an array \( c_k \in \mathbb{R}^{I_T} \), initialized as \( c_k(i) = i \). At each position \( j, j = 1, \ldots, I_T \) of \( c_k \), the fracture index \( \ell \) is stored, defined as \( \ell = \arg \min_{i \in I_k} F_i \cap F_j \neq \emptyset \). The array \( c_k \) can be easily built starting from the local adjacency matrix \( A_k^D \). Then, for each \( i \in I_k \), if \( c_k(i) \neq i \), we set \( j = c_k(i) \) and, recursively, \( c_k(i) = c_k(j) \) and \( j = c_k(i) \), while \( c_k(i) \neq c_k(j) \). Subsequently an MPI_Allreduce operation is used to compute the array \( c \), defined at each position \( i = 1, \ldots, I_T \), as \( c(i) = \min_{k=0, \ldots, n_P} c_k(i) \). Finally, again, for each \( i \in I_k \), if \( c(i) \neq i \), we set \( j = c_k(i) \), and, recursively, \( c_k(i) = c_k(j) \) and \( j = c_k(i) \), while \( c_k(i) \neq c_k(j) \). At the end of this operation the connected component is given taking the largest set of fractures \( \Omega_\zeta \), where for \( \zeta = 1, \ldots, I_T \), \( \Omega_\zeta = \{ F_i, i = 1, \ldots, I_T : c(i) = \zeta \} \). The DFN \( \Omega \) is then set equal to the main connected component identified in such way.

3.3. DFN partitioning. Partitioning is aimed at minimizing the communications, balancing, at the same time, the computational load assigned to each computing process, and is performed using the METIS library [34]. In a Multi-Master topology, the tree of the processes is first built (see Figure 1b), thus defining a hierarchy of processes, with \( n_P^M > 0 \) Master processes, organized in different levels, and \( n_P^S > 0 \) Slave processes at the bottom level. If the number of Slave processes is zero the code is executed in serial. We assume that the Master-Slave tree is a completely balanced tree.
To each Master, the set $\mathcal{P}^k_0$ of processes is then assigned, which might contain either lower level Masters, either a set of Slave processes, such that $\mathcal{P}_k, k = 0, \ldots, n^M_P - 1$ manages the communications among all the processes in $\mathcal{P}^k_0$. The set $\mathcal{P}^k_1$, contains, instead, the unique process at higher level for process $\mathcal{P}_k, k = 1, \ldots, n_P$ (i.e. the higher level Master), and it is empty for $\mathcal{P}_0$.

On the Master process $\mathcal{P}_0$ the DFN is first split through the METIS balanced graph partitioning tool into a number of parts equal to the number of processes in $\mathcal{P}^k_0$, and then each part is assigned to a different process in $\mathcal{P}^k_1$. In the case of Figure 1b, $\mathcal{P}^k_0$ contains three Master processes, and then $\Omega$ is first split into three parts. The operation is repeated on each Master $\mathcal{P}_k \in \mathcal{P}^k_0$, splitting the assigned portion of the network into sub-networks among the processes in $\mathcal{P}^k_1$, i.e. at the lower level, until the level of the Slave processes is reached. In the example of Figure 1b, on each of the three lower level Master processes the DFN is split between two computing processes, although, in practical applications the number of Slaves is much larger than the number of Masters. This procedures ensures that the communications at each level of the topology graph are minimized, at the cost of multiple calls to the METIS library. However, this approach is not expensive compared to the whole resolution process and it is capable of optimizing the partitioning of the graph. Information on the connectivity among the various parts of the original DFN are stored during the partitioning process, to be used for the communication phases of the algorithm.

An estimate of the number of degrees of freedom per fracture and per trace is computed to evaluate the computational cost related to the resolution of linear systems on the fractures and the cost of communications related to each trace. These information can then be used by METIS in order to perform the balanced partitioning of the DFN.

At the end of the partitioning process a set of fracture indexes is assigned to each computing process $\mathcal{P}_k, k = n^M_P, \ldots, n_P$, denoted by $\mathcal{I}^{\text{loc}}_k$. Fractures $F_i, i \in \mathcal{I}^{\text{loc}}_k$ are called process-local fractures for $\mathcal{P}_k$, whereas indexes of fractures that are not process-local, but that form at least one trace with a local fracture are stored in the set $\mathcal{I}^{\text{ext}}_k$, i.e. $j \in \mathcal{I}^{\text{ext}}_k$ if $j \notin \mathcal{I}^{\text{loc}}_k$ and $F_j \cap F_r \neq \emptyset$, for some $r \in \mathcal{I}^{\text{loc}}_k$. We then define, for each computing process $\mathcal{P}_k$ the set $\mathcal{I}^{\text{tot}}_k = \mathcal{I}^{\text{loc}}_k \cup \mathcal{I}^{\text{ext}}_k$. Further, process-local fractures are split between communicating fractures, whose indexes are collected in the set $\mathcal{I}^{\text{com}}_k$, that are fractures $F_i, i \in \mathcal{I}^{\text{loc}}_k$ that have at least one trace in common with a fracture $F_j, j \in \mathcal{I}^{\text{ext}}_k$, and non communicating fractures, whose indexes are collected in $\mathcal{I}^{\text{noc}}_k$, that are fractures $F_i, i \in \mathcal{I}^{\text{loc}}_k$ forming traces only with other fractures $F_j, j \in \mathcal{I}^{\text{loc}}_k$.

The indexes of the fractures associated to each process are collected in the index set $\mathcal{M}^{\text{loc}}_k$, such that $m \in \mathcal{M}^{\text{loc}}_k$ if $S_m \in \mathcal{S}_i$ for some $i \in \mathcal{I}^{\text{loc}}_k$. Trace indexes in $\mathcal{M}^{\text{loc}}_k$ are split into two sets: non-communicating trace set, $\mathcal{M}^{\text{noc}}_k$, collecting indexes of traces formed by two process-local fractures, i.e. $m \in \mathcal{M}^{\text{loc}}_k$ if $I_S(m) \subset \mathcal{I}^{\text{loc}}_k$, and communicating trace set, $\mathcal{M}^{\text{com}}_k := \mathcal{M}^{\text{loc}}_k \setminus \mathcal{M}^{\text{noc}}_k$.

### 3.4. Mesh generation

The generation of the computational mesh is performed by each computing process $\mathcal{P}_k, k = n^M_P, \ldots, n_P$ on the set of local fractures $F_i, i \in \mathcal{I}^{\text{loc}}_k$ and local traces $S_m, m \in \mathcal{M}^{\text{loc}}_k$. As already mentioned, the mesh for the hydraulic head $h_i$ on each fracture $F_i$ is independent from the mesh on the other fractures and from the position of fracture intersections. The triangulation on the fractures is obtained using the Triangle library [45].

On each trace $S_m$ of each fracture $F_i$ a mesh is built for the discretization of function $u^{\text{loc}}_m$, which is, in general, independent from the mesh for the hydraulic head and from the mesh of the same trace on fracture $F_j, (i, j) = I_S(m)$. Thus, thanks to
the proposed optimization approach, the meshing process can be performed in parallel and becomes a trivial task. Only a simple communication phase is required at the end of the meshing process to build a support mesh on each trace, for the evaluation of the integrals on the traces of basis functions (or restriction of basis functions) defined on the different meshes.

Concerning the mesh on the traces, we will distinguish among: Discretization mesh, Induced mesh, and Union mesh, see Figure 2. Each of these meshes is defined differently depending if the trace $S_m$ is seen as an object of fracture $F_i$ or $F_j$, $(i,j) = I_S(m)$, $m = 1, \ldots, M$: in the following we will denote as $S_m^-$ trace $S_m$ as an object of $F_i$ and as $S_m^+$ as an object of $F_j$, recalling that $i < j$. The Discretization mesh can be arbitrarily chosen, as for example a mesh of equally spaced nodes, as shown in Figure 2a. In this case the nodes on $S_m^-$ and $S_m^+$ are staggered, i.e. shifted of a given quantity (typically half of the interval length), with the exception of the first and last node, which always coincide with the two extremes of the trace. As another example, the Discretization mesh might, instead, coincide with the Induced mesh. The Induced mesh is simply obtained taking as nodes the intersection points between the trace and the elements of the fracture mesh, Figure 2b. Finally the Union mesh is given by the union of the Discretization and Induced mesh, see Figure 2c. Another mesh is defined, called Interface mesh, which is unique for each trace, and is given on $S_m$ by the union of the Union mesh of $S_m^-$ and $S_m^+$, as depicted in Figure 2c. This mesh is only used as a support for integration purposes, and its construction requires the communication of the two Union meshes for each trace $S_m$, $m \in M^{\text{com}}$. Values of basis functions $\{\varphi_{i,k}\}_{k=1,\ldots,N_i}$ and $\{\psi_{i,k}\}_{k=1,\ldots,N_i}$ $\forall i \in I_S(m)$, for $m = 1, \ldots, M$ in each quadrature node of the Interface mesh are computed on each process and then shared through an MPI communication by both processes sharing the trace.

3.5. Discrete problem matrix assembly. As mentioned in Section 2.2, equations (2.17)-(2.19) can be split, at each iteration, into sub-problems which can be solved in parallel. Problem (2.17)-(2.19) can be written fracture-wise in the following way: for $i = 1, \ldots, I$:

\begin{align}
(3.1) \quad A_i [h_i]^n &= B_i [u_i]^n + q_i, \\
(3.2) \quad A_i^T [p_i]^n &= G_i^h [h_i]^n - \alpha B_i^h [u_i]^n, \\
(3.3) \quad [g_i]^n &= B_i^T [p_i]^n + G_i^u [u_i]^n - \alpha B_i^u [h_i]^n
\end{align}
where the array $h^+_i$ is obtained appending column-wise to $h_i$ vectors $h_j$, for $j = 1, \ldots, I$ such that $j \neq i$ and $F_i \cap F_j \neq \emptyset$; $u^+_i$ is obtained in a similar way, grouping column-wise vectors $u^+_m$ and $u^+_n$ for $S_m \in S_i$, $(i, j) = I_S(m)$. Matrices $G^h_i$ and $B^u_i$ are the sub-matrices obtained from $G^h$ and $B^u$, respectively, extracting the rows relative to the position of the degrees of freedom of $h_i$ in $h$, and the columns relative to the position of the DOFs of $h^+_i$ in $h$, and similarly matrices $G^u_i$ and $B^h_i$, are obtained from $G^u$ and $B^h$ extracting the rows corresponding to the DOFs of $u_i$ in $u$ and columns corresponding to the DOFs of $u^+_i$ in $u$; finally $g_i$ is the local gradient direction. The parallel implementation proposed in [15] is based on this fracture-wise splitting of the network.

For implementation purposes a different organization of computations appears more effective, with vectors and matrices assembled in blocks as explained in the following, capable of optimizing the performances of the library used for linear algebra computations and the efficiency of MPI communications. On each process $\mathcal{P}_k$, $k = u^+_k, \ldots, n_p$, the following vectors are defined: vector $h^\text{com}_i \in \mathbb{R}^{N^\text{com}_k}$, grouping column-wise vectors $h_i$, $i \in \mathcal{I}_{k}^{\text{com}}$; vector $h^\text{loc}_i \in \mathbb{R}^{N^\text{loc}_k}$, grouping vectors $h_i$, $i \in \mathcal{I}_{k}^{\text{loc}}$; and vector $h^\text{tot}_i \in \mathbb{R}^{N^\text{tot}_k}$, obtained collecting all the vectors $h_i$ for $i \in \mathcal{I}_{k}^{\text{tot}}$, such that $h^\text{tot}_i$ contains $h^\text{com}_i$, $h^\text{loc}_i$, and finally vector $h^\text{ext}_i$, which is given grouping vectors $h_i$, $i \in \mathcal{I}_{k}^{\text{ext}}$. Other vectors are also assembled, named $u^\text{com}_m \in \mathbb{R}^{N^\text{com}_k}$, obtained grouping $u^m$ and $u^n$ for $m \in \mathcal{M}^\text{com}_k$ with $(i, j) = I_S(m)$, and vectors $u^\text{loc}_n \in \mathbb{R}^{N^\text{loc}_k}$, grouping vectors $u^m$ and $u^n$ for $m \in \mathcal{M}^\text{loc}_k$, $(i, j) = I_S(m)$. Matrix $A^\text{com}_k \in \mathbb{R}^{N^\text{com}_k \times N^\text{com}_k}$ is created as a block diagonal matrix with blocks formed by matrices $A_i$ with $i \in \mathcal{I}_{k}^{\text{com}}$ and similarly matrix $A^\text{loc}_k \in \mathbb{R}^{N^\text{loc}_k \times N^\text{loc}_k}$ is a block diagonal matrix with blocks $A_i$, for $i \in \mathcal{I}_{k}^{\text{loc}}$. Matrix $B^\text{com}_k \in \mathbb{R}^{N^\text{com}_k \times N^\text{com}_k}$ is instead formed extracting from matrix $B$ the rows corresponding to the position of the DOFs of $h^\text{com}_i$ in $h$ and the columns corresponding to the position of the DOFs of $u^\text{loc}_i$ in $u$; analogously is obtained $B^\text{loc}_k \in \mathbb{R}^{N^\text{loc}_k \times N^\text{loc}_k}$ from $B$. Matrix $G^\text{com}_k \in \mathbb{R}^{N^\text{com}_k \times N^\text{com}_k}$ is extracted from $G^h$ taking the rows corresponding to the position of $h^\text{com}_i$ in $h$ and the columns corresponding to the position of $h^\text{tot}_i$ in $h$, similarly for $G^\text{loc}_k \in \mathbb{R}^{N^\text{loc}_k \times N^\text{loc}_k}$. Finally matrix $B^h_k \in \mathbb{R}^{N^\text{com}_k \times N^\text{com}_k}$ is created extracting rows from matrix $B^h$ corresponding again to the position of $h^\text{com}_i$ in $h$ and to the columns corresponding to the position of $u^\text{loc}_i$ in $u$, and similarly for $B^\text{loc}_k \in \mathbb{R}^{N^\text{loc}_k \times N^\text{loc}_k}$. Then, at iteration $n$, problem (2.17)-(2.18) is re-written, on each process $\mathcal{P}_k$, as:

$$A^\text{com}_k [h^\text{com}_k]^n = q^\text{com}_k + B^\text{com}_k [u^\text{loc}_k]^n$$
$$A^\text{loc}_k [h^\text{loc}_k]^n = q^\text{loc}_k + B^\text{loc}_k [u^\text{loc}_k]^n$$
$$A^\text{com}_k [p^\text{com}_k]^n = G^\text{com}_k [h^\text{tot}_k]^n - \alpha B^\text{com}_k [u^\text{loc}_k]^n$$
$$A^\text{loc}_k [p^\text{loc}_k]^n = G^\text{loc}_k [h^\text{tot}_k]^n - \alpha B^\text{loc}_k [u^\text{loc}_k]^n.$$

This implementation allows for an improved use of the memory, as it avoids multiple copies of the same data, with respect to a fracture-wise organization, in which arrays $h^+_i$ and $u^+_i$ would contain data already present in $h_i$ and $u_i$, and, in general several copies of the same data would be present on each process.

Equations (3.4) and (3.5) are split since after the resolution of problems (3.4), the data in the array $h^\text{out}_k$ are sent to the Master process by each Slave process. A non blocking MPI send operation is used, and then each Slave process can proceed with the resolution of problem (3.5), before receiving array $h^\text{in}_k$ from the Master, which is required for the resolution of (3.6), where it appears into array $h^\text{tot}_k$. This allows to
maximize the parallel performances of the algorithm, as communication time can be shadowed, on each Slave process, by the resolution of the linear system related to the non communicating fractures. Despite no communication occurs between equations (3.6) and (3.7) they are split to save memory, reusing matrices $A_k^{\text{com}}$ and $A_k^{\text{noc}}$, or their factorizations. Finally equation (2.19) is written on each process $p_k$, $k = 1, \ldots, n_p$ as:

$$g_k^{\text{loc}} = (B_k^{\text{tot}})^T p_k^{\text{tot}} + G_k^{u,\text{loc}} u_k^{\text{loc}} - \alpha B_k^{u,\text{tot}} h_k^{\text{tot}}$$

where $g_k^{\text{loc}}$ is the gradient direction computed for the local traces, and matrices $G_k^{u,\text{loc}}$...
Algorithm 3.2 Communication phase

1: for \( k = 0, \ldots, n^M_P - 1 \) do
2: \[ \text{if } n = 0 \text{ then} \]
3: \[ \text{Initialize MPI indexed arrays } h^\text{com}_k \text{ and } u^\text{com}_k \]
4: \[ \text{Initialize the MPI_SEND_INIT and MPI_RCV_INIT buffers} \]
5: \[ \text{end if} \]
6: \[ \text{Receive } [h^\text{out}]^n_k \text{ and } [u^\text{out}]^n_k \text{ from processes } \mathcal{P}_\ell \in \mathcal{P}_k \]
7: \[ \text{Assemble } [h^\text{in}]^n_k \text{ and } [u^\text{in}]^n_k \]
8: \[ \text{Send } [h^\text{out}]^n_k \text{ and } [u^\text{out}]^n_k \text{ to process } \mathcal{P}_k \]
9: \[ \text{Receive } [h^\text{in}]^n_k \text{ and } [u^\text{in}]^n_k \text{ from process } \mathcal{P}_k \]
10: \[ \text{send } [h^\text{in}]^n_k \text{ and } [u^\text{in}]^n_k \text{ to processes } \mathcal{P}_\ell \in \mathcal{P}_k \]
11: \[ \text{end for} \]

\( h^\text{out, tot}_k \) and \( B^u \) are extracted from \( G^u \) and \( B^u \), respectively, analogously to what described above.

Let us define, for each trace \( S_m \) on fracture \( F_i \) a set of active DOFs, i.e. the subset of DOFs of \( h^i \) which determines the value of \( h_{ij}^i S_m \), i.e. the solution on the trace. Since coefficients of matrices \( G^h \) and \( B^h \) only depend from those basis functions whose support has a non empty intersection with a trace, for efficiency reasons, at each iteration of the scheme (3.4)-(3.7) only the set of active DOFs in arrays \( h^\text{tot}_k \) on each process \( \mathcal{P}_k \) needs to be updated. More in details, \( \mathcal{P}_k \) has to send to the \textit{Master} a subset of the array \( h^\text{com}_k \), denoted as \( h^\text{out}_k \), containing the set of all active DOFs of its communicating fractures \( F_i, i \in \mathcal{T}_k \), and, in turn, receives from the \textit{Master} a subset of \( h^\text{ext}_k \), denoted as \( h^\text{in}_k \), containing the active DOFs of fractures \( F_i, i \in \mathcal{T}_k \). Analogously, concerning trace-variables, the part of \( u^\text{com}_k \) sent to the \textit{Master} is denoted as \( u^\text{out}_k \) and the part received is denoted \( u^\text{in}_k \), now \( u^\text{out}_k \) and \( u^\text{in}_k \) defining the whole \( u^\text{com}_k \) (actually, as shown in the next paragraph, the increment of \( u^\text{com}_k \) is communicated at each iteration and not \( u^\text{com}_k \) itself). The organization of arrays in blocks on each process, as here proposed, also yields efficient MPI communications. Using persistent MPI communication protocols \( \text{MPI_SEND_INIT} \) and \( \text{MPI_RCV_INIT} \) instructions, see for example [48]) and the \( \text{MPI_Type_indexed} \) [49] for \( h^\text{out}_k \), and \( u^\text{com}_k \) it is possible to define, at the first iteration, the indexes to the active DOFs in \( h^\text{com}_k \), \( h^\text{ext}_k \) and \( u^\text{com}_k \), allowing for an optimized indexing and a minimization of communication overhead. In contrast, a fracture-wise organization would require repeated indexing into fracture-local arrays, with a detrimental impact on the efficiency of MPI communications.

3.6. Problem resolution. The optimization problem (2.16) is solved via a conjugate gradient scheme whose parallel implementation is detailed Algorithm 3.1. The notation \( \delta(\cdot) \) is introduced to denote the increment of a variable between two subsequent iterations of the method, as e.g. \( [\delta h^\text{com}_k]^n \) is the variation of \( h^\text{com}_k \) on process \( \mathcal{P}_k \) between iterations \( n \) and \( n+1 \), i.e. \( [\delta h^\text{com}_k]^n = [h^\text{com}_k]^{n+1} - [h^\text{com}_k]^n \). Algorithm 3.2 reports the steps of a communication phase for a \textit{Master} process, at a generic iteration number \( n \geq 0 \). The definition of arrays \( h^\text{out}_k, h^\text{in}_k \) and \( u^\text{out}_k, u^\text{in}_k \) is extended to \textit{Master} processes \( \mathcal{P}_k, k = 1, n^M_P - 1 \) as the set of DOFs that need to be sent and received, respectively, from a different process. Steps 8-9 are not performed by the top level \textit{Master} \( (\mathcal{P}_0^M = \emptyset) \).

A version of the code based on different C++ libraries for linear algebra is de-
4. Results. This Section reports the performances of the proposed algorithm, in terms of achieved speedups in iteration time when the number of parallel processes increases. More in details, by iteration time we mean the average time per iteration over 1000 iterations of Algorithm 3.1. The pre-processing time, needed to perform tasks from 1 to 5 described at the beginning of Section 3, is negligible with respect to resolution time (task 6), and further, pre-processing is readily parallel as it involves almost no communications.

The strong scalability and the weak scalability performances of the algorithm will be shown. In strong scalability tests the dimension of the problem is fixed (number of fractures and mesh parameter) and the time to obtain the solution is measured when the number of parallel processes increases. Performances are measured in terms of the speedup \( S_p := t_1/t_p \), corresponding to the ratio between the serial execution time \( t_1 \) and the time with \( p \) parallel computing processes \( t_p \); and in terms of efficiency \( E_p := S_p/p \) representing the achieved speedup divided by the ideal speedup \( p/1 \). On the other hand, in weak scalability tests both problem size (number of fractures, with a fixed mesh parameter) and the number of parallel processes grow with a fixed ratio. A measure of the performances is obtained, in this case, comparing the ratio between computing time and number of processes (or, equivalently, problem size) with the expected fixed one. Scalability performances are computed with respect to the number of Slave processes, not taking into account the number of Masters. This is done in order to provide an insight of the optimality of the performances with respect to the actual number of computing processes, and to better highlight how performances vary as the number of processes changes, being, instead, the number of Master processes fixed for each simulation.

The networks used for the simulations are generated from random probability distribution functions concerning the size, the position, the orientation, the number and the hydraulic transmissivity of fractures, adapted from the data available in [47]. The resulting networks display a large variability in terms of fracture sizes, which span about four orders of magnitude; also, highly connected fractures, having more than \( 10^2 \) traces, coexist with scarcely connected fractures, with only few traces.

An extensive set of simulations is reported, performed on the partition A1 of cluster Marconi, located at the Italian HPC center CINECA [19]. The machine is composed by 720 nodes, with 2x18-cores Intel Xeon E5-2697 v4 (Broadwell) processors at frequency 2.30 GHz, and 128 GB of RAM per node. The OpenMPI implementation of the MPI communication protocol version 3 is used, [50].
Table 1: Strong scalability tests - Network data

<table>
<thead>
<tr>
<th>Id</th>
<th>Tot. Fracs</th>
<th>Conn. Fracs</th>
<th>Traces Min</th>
<th>Trcs Max</th>
<th>Total Trcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>64186</td>
<td>64178</td>
<td>122543</td>
<td>1</td>
<td>390</td>
</tr>
<tr>
<td>128K</td>
<td>167321</td>
<td>129552</td>
<td>248422</td>
<td>1</td>
<td>392</td>
</tr>
<tr>
<td>256K</td>
<td>328206</td>
<td>253150</td>
<td>482755</td>
<td>1</td>
<td>442</td>
</tr>
</tbody>
</table>

Table 2: Strong scalability tests - DOFs data

<table>
<thead>
<tr>
<th>Id</th>
<th>Mesh</th>
<th>Fracture DOF’s</th>
<th>Trace DOF’s</th>
<th>Total DOF’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>100</td>
<td>6.07 · 10^6</td>
<td>6.13 · 10^5</td>
<td>6.68 · 10^6</td>
</tr>
<tr>
<td>128K</td>
<td>100</td>
<td>1.23 · 10^7</td>
<td>1.24 · 10^6</td>
<td>1.35 · 10^7</td>
</tr>
<tr>
<td>256K</td>
<td>100</td>
<td>2.39 · 10^7</td>
<td>2.41 · 10^6</td>
<td>2.64 · 10^7</td>
</tr>
</tbody>
</table>

Figure 4: Strong scalability on DFN 64K and 128K

Figure 5: Sparse matrix-vector product performance test

Figure 6: Strong scalability with different task×node configurations
4.1. Strong Scalability. Strong scalability results are shown for three different networks with an increasing number of fractures. The three networks, labelled 64K, 128K and 256K, are displayed in Figure 3. Table 1 reports, for each DFN the initial number of fractures, (column Tot. Fracs), i.e. before the definition of the main connected component, the number of connected fractures (column Conn. Fracs) and their traces (column Traces) and the minimum and maximum number of traces per fracture (columns Min Trcs and Max Trcs, respectively). The chosen mesh parameter and the resulting degrees of freedom on the fractures, on the traces, and their sum, are shown in Table 2. Linear finite elements are used for the description of the discrete hydraulic head $h$ on the fractures, and trace-meshes with staggered nodes and piecewise-constant basis functions are used to describe function $u$ on the traces. The diameter of the mesh on each trace $S_m$, $m = 1, \ldots, M$, is chosen equal to the average between the maximum diameter of the triangular elements intersected by the trace on $F_i$ and $F_j$, $(i, j) = I_S(m)$.

Figure 4 plots the speedup $S_p$ for $p$ ranging between 1 and 15, for the 64K and 128K networks, also reporting, in the embedded table, the corresponding efficiency values. Simulations are performed ensuring that no more than a single process is allocated on each computing node of the cluster, thus avoiding any possible memory-related access conflict between processes. An optimal speedup is achieved for the smaller network, whereas a speedup exceeding the optimal one is observed for the larger DFN. This circumstance is not an isolated case, as shown in the following, and is given by the combined effect of a very low communication cost, achieved by the present implementation, with the behaviour of the cost of linear algebra operations when the size of the involved matrices varies. This behaviour is described by the example in Figure 5, reporting how the cost of sparse matrix-vector product performed with Eigen changes when the same matrix is split in several sub-matrices taking chunks of rows, keeping instead fixed the size of the vector. The test is performed on a single core, and does not involve communications: starting from an initial reference matrix, seven subsequent simulations are performed, each time defining a sub-matrix taking one half
of the number of rows of the matrix at the previous simulation, and measuring the
time required to perform the sparse matrix vector product. This mimics what happens
in Algorithm 3.1 when the network is split among an increasing number of parallel
processes. Speedup measurements for this test are obtained dividing the time for the
computation of the matrix-vector product for the initial reference matrix by the time
required to perform the same operation on each fraction of the reference matrix. In
Figure 5 the achieved speedups are shown with respect to the **NNZ Ratio**, defined as
the ratio between the non zero elements of the initial reference matrix and the number
of non zero elements of the considered fractions of this matrix; also efficiency values
are shown in the embedded table, defined, as usual, as the ratio between the achieved
speedup and the ideal one. It can be seen that a speedup much larger than optimal
is obtained: as an example, when the number of non zero elements of the matrix
is reduced of a factor \(\frac{71}{174}\) the computing time is reduced of a factor close to 150.
Consequently, efficiency values exceeding 200% are reached. This effect becomes less
relevant when the number of non-zero elements further decreases. A similar behaviour
is documented for the Eigen library in [23] in the case of full matrices, and also applies
to other linear algebra libraries. As the DFN is partitioned among a larger number
of parallel processes, the size of the matrices in equations (3.4)-(3.8) is reduced, as
less fractures are present in sets \(T_k^{loc} \), \(k = n_M^P, \ldots, n_P^P\), and thus the number of non
zero entries of the matrices on each process reduces. The cost of communications is
negligible with respect to the cost of linear algebra computations, and well shadowed
by the operations on the non communicating fractures, especially when the number
of parallel processes is relatively small. Thus a speedup higher than the ideal one can
be achieved, and this is more evident, as expected, on larger networks, where the cost
of linear algebra operations is more relevant. When more than a single process per
node is allocated, cache memory access conflicts of different processes on the same
computing node can deteriorate the speedup, with an impact that increases on larger
networks, for which more memory is used, see Figure 6.

Figure 7 shows the achieved speedup for a number of computing processes ranging
between 15 and 127, using eight cores on each node of the cluster, for the 64K, 128K
and 256K networks. Solid lines represent the speedups obtained with a single **Master
process. A scalability higher than the ideal optimal value is again observed, more
evident for the larger 256K network. However, when the number of parallel processes
increases, the cost of the communications becomes more and more relevant, up to
overweight the cost related to linear algebra manipulations. This happens earlier for
the smaller networks, for the reduced cost of linear algebra operations. The reason of
the decay of the parallel performances of the method, however, lies in the overload of
the **Master process, as demonstrated observing the results in dashed lines, where the
same tests are repeated using a two level **Multi-Master topology, with one top-level
**Master process and three bottom level Master processes (as in Figure 1b) and an
increasing number of **Slave processes. It can be seen that now the decay of speedup
performances is completely cured on all the considered networks. The performances in
the pre-decay range, i.e. 15 – 63 **Slaves for the 64K and 128K networks and 15 – 104
**Slaves for the 256K network are unaffected by the change of the communication
topology, again showing the negligible weight of communications in the proposed
algorithm when the **Master process is under-saturated.

Results in Table 3 show the effect of an increment in the connectivity of the net-
work on the scalability performances. A new DFN, labelled 64K* is introduced, hav-
ing about the same number of fracture as the 64K network, but with about \(1.7 \times 10^5\)
traces, i.e. more than 10 times the number of traces of the 64K DFN. Using the
Table 3: Strong scalability test on a high density network and on a different architecture

<table>
<thead>
<tr>
<th>Slaves</th>
<th>64K</th>
<th>64K*</th>
<th>64K*(MM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S_p$</td>
<td>$E_p$ (%)</td>
<td>$S_p$</td>
</tr>
<tr>
<td>8</td>
<td>7.94</td>
<td>99.28</td>
<td>7.96</td>
</tr>
<tr>
<td>16</td>
<td>15.70</td>
<td>98.10</td>
<td>14.45</td>
</tr>
</tbody>
</table>

Table 4: Weak scalability tests: data on DFN families

<table>
<thead>
<tr>
<th>ℓ</th>
<th>$I_\ell$</th>
<th>Samples</th>
<th>Tot. Fracs</th>
<th>Conn. Fracs</th>
<th>Traces</th>
<th>DOFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>13</td>
<td>1037</td>
<td>1035</td>
<td>1901</td>
<td>$1.33 \cdot 10^5$</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>25</td>
<td>2073</td>
<td>2070</td>
<td>3896</td>
<td>$2.67 \cdot 10^5$</td>
</tr>
<tr>
<td>3</td>
<td>4000</td>
<td>19</td>
<td>4260</td>
<td>4254</td>
<td>8197</td>
<td>$5.51 \cdot 10^5$</td>
</tr>
<tr>
<td>4</td>
<td>8000</td>
<td>20</td>
<td>8366</td>
<td>8357</td>
<td>15991</td>
<td>$1.08 \cdot 10^6$</td>
</tr>
<tr>
<td>5</td>
<td>16000</td>
<td>11</td>
<td>16125</td>
<td>16116</td>
<td>30747</td>
<td>$2.08 \cdot 10^6$</td>
</tr>
<tr>
<td>6</td>
<td>32000</td>
<td>16</td>
<td>32000</td>
<td>31987</td>
<td>60923</td>
<td>$4.12 \cdot 10^6$</td>
</tr>
<tr>
<td>7</td>
<td>64000</td>
<td>12</td>
<td>77302</td>
<td>64535</td>
<td>123848</td>
<td>$8.34 \cdot 10^6$</td>
</tr>
<tr>
<td>8</td>
<td>128000</td>
<td>10</td>
<td>167321</td>
<td>129479</td>
<td>248275</td>
<td>$1.67 \cdot 10^7$</td>
</tr>
</tbody>
</table>

same mesh parameter of the previous simulations, $5.86 \times 10^6$ DOFs for the hydraulic head (i.e. approximately the same number as DFN 64K) and $1.4 \times 10^7$ DOFs for the variables on the traces are obtained, which is over one order of magnitude more than the 64K network. Despite the sensible increase in the number of traces and trace DOFs, and the consequent increase of the cost of communications, the scalability performances of the code are only marginally affected, as it can be noticed comparing columns 64K and 64K* of Table 3, where speedup and efficiency values are reported for the 64K and 64K* networks, respectively. Scalability performances on the 64K* network can be further improved introducing a multi-Master topology, as shown in column 64K*(MM). For memory capacity reasons, scalability data are computed using the time with four computing processes as reference, and using a single core per node. The results of Table 3 are computed on the partition A2 of the cluster Marconi, equipped with Intel Xeon Phi7250 1.4 GHz (KnightLandings) processors, a different architecture from that used for the previous tests, thus also showing the good scalability of the code on different architectures.

4.2. Weak Scalability. Weak scalability tests are performed on families $\mathcal{D}_\ell$, $\ell = 1, \ldots, 8$ of randomly generated DFNs: each family is characterized by networks having approximately the same number of fractures, whereas all other properties, such as the position, orientation, density and the hydraulic conductivity of the fractures, are randomly generated starting from the same probability distributions, based on the data in [47]. For each family $\mathcal{D}_\ell$, an initial total number of fractures is fixed and then various samples are generated, according to the distributions of the various parameters: information on each DFN family can be found in Table 4, where the number of networks considered (column Samples), the initial total number of fractures (column Tot. Fracs), the average number of connected fractures (column Conn. Fra...
Fracs), the average number of traces (column Traces) and the average total number of degrees of freedom (column DOFs) are reported. It can be seen that networks with a number of fractures ranging between about $10^3 = I_1$ and $128 \times 10^3 = I_8$ are used, and the total number of DOFs spans two orders of magnitude. Given $p$ the number of parallel computing processes and $I_\ell$ the number of fractures characterizing family $D_\ell$, $\ell = 1, \ldots, 8$, three different tests are performed, each with an approximately fixed ratio $N_p := I_\ell / p$, obtained varying $p$ between 1 and 63, and choosing $\ell$ accordingly.

Weak scalability performances are reported in Figure 8 for values of $N_p \sim 1000$ and $N_p \sim 2000$ and a number of computing processes ranging between 2 and 16, with a single process allocated on each computing node of the cluster. Each point in the picture represents the value of iteration time for one of the networks, and the dashed lines correspond to the best fitting line of the data for each of the two $N_p$-values considered. Similar results are obtained for both values of $N_p$. It can be seen that best fitting lines are nearly parallel and horizontal, thus showing that iteration time is, on average, constant when the number of processes increases and $N_p$ is kept almost constant. We also observe that the variability in computing time among DFNs belonging to the same family tends to decrease as the number of fractures in the network increases.

For values of $N_p \lesssim 1000$ the scalability performances start to gradually deteriorate, suggesting that, for optimal performances, each parallel process should handle not less than $10^5$ DOFs.

Figure 9 plots the results obtained with $N_p = 2000$, by using eight cores per node and a number of parallel computing processes ranging between 7 and 63. Again, each point in this figure corresponds to the computing time of a different network and the dashed line is the best fitting of the data. The fitting line is almost horizontal, slightly decreasing, and this can be probably related again to the super-scalability behaviour observed in terms of strong scalability, for the larger networks.

5. Conclusions. A new implementation of an optimization-based approach for the simulation of subsurface flows has been described and tested. Mesh generation, problem discretization and resolution as well as other pre-processing and post-processing operations, are all performed in parallel by the proposed code. Full im-
plementation details are given, as, in particular, on the organization of algebraic operations, on the use of MPI communications and on the chosen graph topology for the parallel environment. Results are shown in terms of both strong and weak scalability on massive networks of over than $10^5$ fractures, using up to 128 parallel processes, obtaining optimal scalability performances. In contrast to other numerical schemes for DFN flow simulations, where meshing time can be orders of magnitude larger than resolution time, the proposed approach achieves negligible meshing time, without compromising resolution time, thus being a new efficient tool for underground flow simulations at the reservoir scale.

Similar scalability performances are expected in the resolution of non stationary advection-dispersion problems, where, following the method described in [14], a problem analogous to (2.11) needs to be solved at each time-step. In the framework of time-dependent simulations, excellent scalability properties are of paramount importance for the practical applicability of the method. The same setting, as proposed in the present work, can be used for the parallel implementation of the optimization method for fracture-matrix flow simulations on non conforming meshes with the DFM approach, proposed in [13]. In this case, the balancing of the load among the computing processes results to be more difficult, as now bi-dimensional and three-dimensional problems need to solved simultaneously.

REFERENCES


PARALLEL FLOW SIMULATIONS IN DISCRETE FRACTURE NETWORKS


J. D. Hyman, S. Karra, N. Makedonska, C. W. Gable, S. L. Painter, and H. S.


