Arch Comput Methods Eng (2013) 20:239–262
DOI 10.1007/s11831-013-9086-4

Implementation and Scalability Analysis of Balancing Domain Decomposition Methods

Santiago Badia · Alberto F. Martín · Javier Principe

Received: 10 March 2013 / Accepted: 10 March 2013 / Published online: 2 July 2013

Abstract In this paper we present a detailed description of a high-performance distributed-memory implementation of balancing domain decomposition preconditioning techniques. This coverage provides a pool of implementation hints and considerations that can be very useful for scientists that are willing to tackle large-scale distributed-memory machines using these methods. On the other hand, the paper includes a comprehensive performance and scalability study of the resulting codes when they are applied for the solution of the Poisson problem on a large-scale multicore-based distributed-memory machine with up to 4096 cores. Well-known theoretical results guarantee the optimality (algorithmic scalability) of these preconditioning techniques for weak scaling scenarios, as they are able to keep the condition number of the preconditioned operator bounded by a constant with fixed load per core and increasing number of cores. The experimental study presented in the paper complements this mathematical analysis and answers how far can these methods go in the number of cores and the scale of the problem to still be within reasonable ranges of efficiency on current distributed-memory machines. Besides, for those scenarios where poor scalability is expected, the study precisely identifies, quantifies and justifies which are the main sources of inefficiency.

S. Badia (✉) · A.F. Martín · J. Principe
Centre Internacional de Mètodes Numèrics a l'Enginyeria (CIMNE), Parc Mediterrani de la Tecnologia, UPC, Esteve Terradas 5, 08860 Castelldefels, Spain
e-mail: sbadia@cimne.upc.edu

A.F. Martín
e-mail: amartin@cimne.upc.edu

J. Principe
e-mail: principe@cimne.upc.edu

S. Badia · A.F. Martín · J. Principe
Universitat Politècnica de Catalunya, Jordi Girona 1-3, Edifici C1, 08034 Barcelona, Spain

1 Introduction

Scientific phenomena governed by partial differential equations (PDEs) can range from solid mechanics to fluid mechanics and electrodynamics, including any of the possible couplings. The solution of these equations can be approximated with the aid of computers by a discretization (and possibly linearization) and the subsequent numerical solution of the resulting sparse set of linear equations. This work is concerned with the fast solution of the Poisson problem discretized by the finite element (FE) method. Although the Poisson problem is the simplest model problem for, e.g., fluid flow simulation, it is still very useful as a building block for the "physics-based" preconditioning of very complex scientific applications governed by coupled systems of PDEs [1].

The ever increasing demand of reality in the simulation of the complex scientific and engineering three-dimensional (3D) problems faced nowadays ends up with the solution of very large and sparse linear systems with several hundreds and even thousands of millions of equations/unknowns. The solution of these systems in a moderate time requires the vast amount of computational resources provided by current multicore-based distributed-memory machines. It is therefore essential to design parallel algorithms able to take profit of their underlying architecture.

Non-overlapping domain decomposition (DD) methods (also referred as iterative sub-structuring or Schur complement methods) provide a natural framework for the development of fast parallel solvers tailored for distributed-memory machines, as they have by construction the desirable design principle of maximizing local computations while minimizing interprocessor communication. One-level DD preconditioners, such as the Neumann-Neumann preconditioner, are highly parallel as they only require the solution of local problems and communication among neighboring subdomains, but unfortunately they do not posses optimality properties. Consequently, e.g., in a weak scaling scenario, where the number of processors is increased while keeping the load per processor constant, more a more computational resources are wasted because extra iterations are required to converge. Two-level DD preconditioners combine local and global terms acting in an additive or in a multiplicative fashion in order to achieve quasi-optimal condition number bounds (in the sense discussed in the next paragraph). The global term couples all the subdomains and involves the solution of a "small" (relative to the original linear system) coarse-grid problem. Besides, its size has typically only a linear dependence with the number of processors. However, only a small amount of parallelism can be exploited for the solution of this coarse-grid problem, which results in increasing parallel overheads (i.e., loss of efficiency) with the number of processors.

In this work we focus on two-level DD methods of balancing type, namely the Balancing Neumann-Neumann [28] (BNN) and Balancing DD by Constraints [10] (BDDC) methods; other two-level DD preconditioners are found in the FETI family [11, 12], although they are not explored here. These methods are quasi-optimal (algorithmically scalable) with a poly-logarithmic expression of the condition number of the preconditioned system $\kappa = 1 + \log^2(\frac{H}{h})$, where h and H are, respectively, the mesh and subdomain characteristic sizes, $(\frac{H}{h})^d$ is the size of the local problems and d is the space dimension. Consequently, in weak scaling scenarios (i.e., $\frac{H}{h}$ fixed), the number of iterations of the preconditioned conjugate gradient (PCG) solver is (asymptotically) independent of the number of processors.

Even though the mathematical theory of these methods is well established, there is a surprising lack of scientific works on the design/implementation issues to be considered for the efficient exploitation of distributed-memory machines. And even more surprising is the lack of comprehensive performance and scalability analysis on large-scale distributed-memory machines. This situation is in contrast to multigrid methods, see e.g. [3, 27]. We believe that the ability of balancing DD (BDD) methods to exploit large-scale distributed-memory machines is the most cited feature but their least examined one. To the best of our knowledge only few studies focus on these aspects for this particular family of algorithms [7, 19, 20, 38], none of which with the degree of detail and up to the scale that are considered here. Given this lack, the contribution of this paper to the state-of-the-art is twofold. On the one hand, we present a comprehensive coverage of design/implementation issues provided by the experience we have acquired by implementing them from scratch in our FE/numerical linear algebra library which results in part from preliminary scalability studies. This coverage is intended to provide scientists with some hints and issues that have to take into account if they want to tackle large-scale problems efficiently on distributed-memory machines. On the other hand, we present a comprehensive weak scalability study of this implementation on a distributed-memory machine with up to 4096 cores. The main objective of this study is to identify and quantify sources of overhead in our current implementation (mainly the impact of the coarse-grid solver) and determine to what degree they are weakly scalable, i.e., how far can these methods go in the number of cores and the scale of the problem to still be within reasonable ranges of efficiency.

The article is organized as follows. In Sect. 2 we present the basic ideas underlying BDD preconditioners. For theoretical aspects of the algorithms we refer the reader to the vast literature devoted to DD methods; see, e.g, [42] and references therein. Our high-performance distributed-memory implementation of these methods is described in Sect. 3, and Sect. 4 presents the aforementioned scalability study. Finally, some concluding remarks are enumerated in Sect. 5.

2 Overview of BDD Methods

This section describes non-overlapping DD methods of balancing type. Section 2.1 covers the general framework of these methods. In Sect. 2.2, the Neumann-Neumann (NN) preconditioner [9, 18] is presented. Although this preconditioner is not algorithmically scalable (as it does not include a coarse-grid correction), it is the basis for the more sophisticated BNN preconditioner [28], which is covered in Sect. 2.3. Finally, the widely used BDDC [10] preconditioning technique is described in Sect. 2.4.

2.1 General Framework

As model problem, let us consider the Poisson problem on a domain $\Omega \subset \mathbb{R}^d$, with homogeneous Dirichlet boundary conditions on $\partial\Omega$, where $d = 2, 3$ is the number of space dimensions. We also consider a uniform FE partition (mesh) $\mathcal{T} = \{K_i : i = 1, \dots, n_{\text{elm}}\}$ of Ω with characteristic size h. We are interested in solving the set of linear equations

$$Ax = b, \tag{1}$$

which arises from the Galerkin FE discretization of the continuous problem corresponding to $\mathcal{T}$.

Further, we consider a uniform non-overlapping partition of Ω into subdomains $\{\Omega_i : i = 1, \dots, n_{\text{sbd}}\}$ with characteristic size H and a partition of the global mesh $\mathcal{T}$ into local meshes $\{\mathcal{T}_i : i = 1, \dots, n_{\text{sbd}}\}$ such that $\mathcal{T}_i$ is a conforming mesh of Ω_i. The interface of Ω_i is defined as $\Gamma_i = \partial\Omega_i \setminus \partial\Omega$ and the whole interface (skeleton) of the domain decomposition is $\Gamma = \bigcup_{i=1}^{n_{\mathrm{sbd}}} \Gamma_i$. The set of nodes of $\mathcal{T}$ that belong to Γ (resp. Γ_i) is denoted by Γ_h (resp. Γ_h^i). This partition of the domain into non-overlapping subdomains induces the following block reordered structure of(1):

$$\begin{bmatrix} A_{II} & A_{I\Gamma} \\ A_{\Gamma I} & A_{\Gamma\Gamma} \end{bmatrix} \begin{bmatrix} x_I \\ x_\Gamma \end{bmatrix} = \begin{bmatrix} b_I \\ b_\Gamma \end{bmatrix}, \tag{2}$$

where x_Γ contains the unknowns corresponding to the nodes in Γ_h and x_I the remaining ones, associated with subdomain interiors. Besides, A_{II} presents a block diagonal structure (and therefore very amenable to parallelization), i.e.,

$$A_{II} = \mathrm{diag}\big(A_{II}^1, A_{II}^2, \dots, A_{II}^{n_{\mathrm{sbd}}}\big),$$

where A_{II}^i is the local matrix which represents the coupling of internal unknowns at subdomain i. Eliminating x_I from (2) (exactly as in the static condensation of internal nodes of high order FEs), this linear system is reduced to the Schur complement problem

$$Sx_\Gamma = g,$$

where $S = A_{\Gamma\Gamma} - A_{\Gamma I} A_{II}^{-1} A_{I\Gamma}$, and

$$g = b_\Gamma - A_{\Gamma I} A_{II}^{-1} b_I. \tag{3}$$

Let us denote the cardinality of Γ_h and Γ_h^i by $\widehat{n}$ and n_i respectively. The vector space of interface nodal values in Γ_h is denoted by $\widehat{V}$; clearly, $\widehat{V}$ is equivalent to $\mathbb{R}^{\widehat{n}}$. We also define the local space V_i of interface nodal values on Γ_h^i, which is equivalent to $\mathbb{R}^{n_i}$.[1] Clearly, the Schur complement matrix $S : \widehat{V} \times \widehat{V} \to \mathbb{R}$. System (3) can be written as the assembly (sum) of local Schur complement matrices and right hand side vectors as

$$S = \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t S_i R_i, \qquad g = \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t g_i, \tag{4}$$

where $R_i : \widehat{V} \to V_i$ is the restriction operator and R_i^t its transpose. The former applied to a vector $y \in \widehat{V}$ gives the vector of local values $y_i = R_i y \in V_i$, while the latter applied to a local vector gives a global vector (filled with zeros for nodes not belonging to subdomain i). The local Schur complement S_i and local right hand side vector g_i are defined as:

$$\begin{aligned} S_i &= A_{\Gamma\Gamma}^i - A_{\Gamma I}^i \big(A_{II}^i\big)^{-1} A_{I\Gamma}^i, \\ g_i &= b_\Gamma^i - A_{\Gamma I}^i \big(A_{II}^i\big)^{-1} b_I^i. \end{aligned} \tag{5}$$

The number of subdomains sharing the node with identifier p is denoted by $n(p)$. We will also make use of the global set of *replicated* local nodes $\Pi_{i=1}^{n_{\mathrm{sbd}}} \Gamma_h^i$, i.e., p is replicated $n(p)$ times, and the corresponding product space $V = \Pi_{i=1}^{n_{\mathrm{sbd}}} V_i$. By definition, the cardinality of this space is $n_\gamma = \sum_{i=1}^{n_{\mathrm{sbd}}} n_i$, which is equivalent to $\mathbb{R}^{n_\gamma}$.[2] Any vector $s \in V$ is univocally defined by local values $\{s_i : i = 1, \dots, n_{\mathrm{sbd}}\}$ due to the product space definition. It is possible to obtain an *averaged* global vector $z \in \widehat{V}$ from s as

$$z = \sum_{i=1}^{n_{\mathrm{sbd}}} I_i s_i, \tag{6}$$

where $I_i = R_i^t W_i : V_i \to \widehat{V}$ is the injection operator, and W_i is a diagonal weighting matrix such that

$$y = \sum_{i=1}^{n_{\mathrm{sbd}}} I_i R_i y, \quad \text{for any } y \in \widehat{V}. \tag{7}$$

W_i can be defined as $(W_i)_{pp} = 1/n(p)$, $p = 1, \dots, n_i$, although more elaborated expressions must be considered for discontinuous physical properties [42].

We can readily check that $n_{\mathrm{sbd}} = H^{-d}$; assuming a one-to-one mapping between subdomains and processors, we will denote the number of processors $P = n_{\mathrm{sbd}}$. The size of the global problem (1) is denoted by $N = h^{-d}$. The condition number of the global matrix A is $\mathcal{O}(N^{\frac{2}{d}})$ whereas that of the Schur complement S is $\mathcal{O}(N^{\frac{1}{d}} P^{\frac{1}{d}})$ [42]. It is well known that the number of iterations required by the PCG Krylov solver is $\mathcal{O}(\sqrt{\kappa})$, where κ is the condition number of the preconditioned operator [32]. Therefore, the estimated number of PCG iterations is $\mathcal{O}(N^{\frac{1}{d}})$ and $\mathcal{O}(N^{\frac{1}{2d}} P^{\frac{1}{2d}})$ when it is applied to (1) and (3), respectively. Although the number of PCG iterations is certainly cut down by the restatement of the problem on the interface by the DD approach (since $N \gg P$ for practical ranges of application), there is a lot of margin for improvement via preconditioning. In the rest of this section, we present some non-overlapping DD preconditioners for the Schur complement matrix S such that the resulting condition numbers become (almost) independent of N and P.

2.2 Neumann-Neumann Preconditioner

We can readily observe that a local contribution to the Schur complement S_i is a singular matrix for every *floating* subdomain i, i.e., $\partial\Omega \cap \partial\Omega_i = \emptyset$ with $\ker(S_i) = \{1_i\}$ (the space of constant functions). We denote the pseudo-inverse of the

[1] The spaces $\widehat{V}$ and V_i can also be understood in a functional setting as the global and local spaces of discrete harmonic functions (see [6]).

[2] In a functional setting, functions in $\widehat{V}$ are uni-valued on Γ. On the contrary, since every node p in Γ_h is replicated $n(p)$ times in $\Pi_{i=1}^{n_{\mathrm{sbd}}} \Gamma_i$, functions in V can take different values at different subdomains. As in [42], $\widehat{\cdot}$ is used to denote uni-valued functions on Γ.

local Schur complement S_i as $S_i^{\dagger}$. The Neumann-Neumann (NN) preconditioner is an additive Schwarz preconditioner built from local pseudo-inverses as

$$B_{\mathrm{NN}}^{-1} = \sum_{i=1}^{n_{\mathrm{sbd}}} I_i S_i^{\dagger} I_i^t.$$

For non-floating subdomains $S_i^{\dagger} = S_i^{-1}$ whereas for floating ones $S_i^{\dagger}$ applied to a vector $r \in \ker(S_i)^{\perp}$gives the unique solution of the singular problem $S_i x = r$ such that $x \in \ker(S_i)^{\perp}$ (see [39]). When $r \notin \ker(S_i)^{\perp}$, the problem $S_i x = r$ does not have a solution and $S_i^{\dagger} r$ only minimizes $\|r - S_i x\|$. The following condition number estimate (cf. [42])

$$\kappa\left(B_{\mathrm{NN}}^{-1} S\right) \leq C P^{\frac{2}{d}} \left[1 + \frac{1}{d^2} \log^2\left(\frac{N}{P}\right)\right], \tag{8}$$

gives $\mathcal{O}(P^{\frac{1}{d}})$ number of PCG iterations for weak scaling analysis.

2.3 Balancing Neumann-Neumann Preconditioning

The NN preconditioner was enhanced in [28] by introducing a coarse-grid solver, the so-called balancing, designed to satisfy $I_i^t r \in \ker(S_i)^{\perp}$ on each subdomain and to provide a global communication mechanism among subdomains. This condition is equivalent to

$$0 = \varphi_i^t I_i^t r = (I_i \varphi_i)^t r, \qquad \text{for any } \varphi_i \in \ker(S_i),$$

i.e., making the residual orthogonal to the injection I_i of functions $\varphi_i \in \ker(S_i)$ to $\widehat{V}$. We introduce the coarse space

$$H_0 = \operatorname{span}\left\{\phi^i : i = 1, \ldots, n_{\mathrm{sbd}}\right\} \subseteq \widehat{V},$$

where $\phi^i = I_i \varphi_i$; H_0 is readily represented by $\mathbb{R}^{n_{\mathrm{sbd}}}$ vectors. The coarse-grid preconditioner can be written as

$$B_{\mathrm{C}}^{-1} = I_0 S_0^{-1} I_0^t,$$

where $I_0 : H_0 \rightarrow \widehat{V}$ is the injection defined as $I_0 \gamma = \sum_{i=1}^{n_{\mathrm{sbd}}} \phi^i \gamma_i$ (i.e., the columns of I_0 are the basis functions ϕ^i), and $S_0 = I_0^t S I_0$ is the coarse-grid space operator.

The coarse-grid balancing preconditioner is combined with the Neumann-Neumann one in a multiplicative fashion, leading to the BNN preconditioner. In order to preserve symmetry, this combination results in the following (naive) form of the BNN preconditioner:

$$B_{\mathrm{BNN}}^{-1} = B_{\mathrm{C}}^{-1} + \left(I - B_{\mathrm{C}}^{-1} S\right) B_{\mathrm{NN}}^{-1} \left(I - S B_{\mathrm{C}}^{-1}\right).$$

It can be further shown [41] that if the initial residual in the PCG algorithm is balanced, i.e.,

$$r^* = \left(I - S B_{\mathrm{C}}^{-1}\right) r,$$

then the preconditioner can be rewritten as

$$B_{\mathrm{BNN}}^{-1} = B_{\mathrm{C}}^{-1} + \left(I - B_{\mathrm{C}}^{-1} S\right) B_{\mathrm{NN}}^{-1}.$$

An important observation is that the BNN preconditioner is more efficiently implemented as:

$$B_{\mathrm{BNN}}^{-1} = B_{\mathrm{C}}^{-1} \left(I - S B_{\mathrm{NN}}^{-1}\right) + B_{\mathrm{NN}}^{-1},$$

as it was originally proposed in [28]. This equivalent expression results in a three-step application, (1) $z = B_{\mathrm{NN}}^{-1} r$; (2) $t = B_{\mathrm{C}}^{-1}(r - Sz)$; (3) update $z := z + t$. Let us remark that in this case the application of the BNN preconditioner only requires to solve one coarse-grid problem. A modified implementation of the algorithm that leads to a spare of one Dirichlet solve per PCG iteration has recently been proposed in [2].[3] The condition number estimate (cf. [28])

$$\kappa\left(B_{\mathrm{BNN}}^{-1} S\right) \leq C \left[1 + \frac{1}{d^2} \log^2\left(\frac{N}{P}\right)\right], \tag{9}$$

results in a constant number of PCG iterations for weak scaling analysis.

2.4 Balancing DD by Constraints Preconditioner

The BDDC preconditioner also presents a two-level structure where local fine-grid and global coarse-grid corrections are combined. However, in contrast to the BNN preconditioner, the combination is additive and the coarse problem is not a Galerkin projection. The construction of the BDDC preconditioner is based on a topological classification of the nodes on the interface as corners, or members of edges or faces.

We denote by $N(p)$ the *index set* of subdomains that share node p, i.e., $N(p) = \{i : p \in \Gamma_h^i\}$, with cardinality already defined in Sect. 2.1 as $n(p)$. We can construct the set $\mathcal{G} = \{G_a : a = 1, \ldots, n^{\mathrm{cts}}\}$, where every *object* G_a is a maximal subset of nodes in Γ_h with identical index set, i.e., $N(p) = N(q)$ for any $p,\ q \in G_a$, denoted as $N(G_a)$.[4] Now, we can consider a topological classification of the objects as follows: G_a is a *face* if $|N(G_a)| = 2$ and $|G_a| > 1$, an *edge* if $|N(G_a)| > 2$ and $|G_a| > 1$ and a *corner* if $|G_a| = 1$; this definition corresponds to a 3D space but can readily be restricted to 2D. Grouping together the objects of the same type, we obtain the set of faces $\mathcal{F} = \{F_a,\ a = 1, \ldots, n^F\}$,

[3] It is based on the observation that $S I_0 v_0$ for $v_0 \in H_0$ can readily be obtained as a linear combination of $S\phi^i$ quantities, that have already been computed when assembling S_0 at the preconditioner set-up. This observation, combined with a slight modification of the PCG recurrence described in [2], spares one Schur complement-vector product per PCG iteration.

[4] This definition of $\{G_a : a = 1, \ldots, n_{cts}\}$ generates a unique partition of Γ_h.

the sets of edges $\mathcal{E} = \{E_a,\ a = 1, \dots, n^E\}$ and the set of corners $\mathcal{C} = \{C_a,\ a = 1, \dots, n^C\}$; clearly $n_{\text{cts}} = n_F + n_E + n_C$. We can consider the restriction of all these sets to a given subdomain i as follows: we define $\mathcal{G}_i = \{G_a \in \mathcal{G} : i \in N(G_a)\}$ and $n^i_{\text{cts}} = |\mathcal{G}_i|$ denotes the number of constraints on subdomain i; analogously for $(\mathcal{F}_i,\ n^i_F)$, $(\mathcal{E}_i,\ n^i_E)$ and $(\mathcal{C}_i, n^i_C)$.

The local, fine-grid preconditioner in the BDDC method is defined as

$$B_{\text{F}}^{-1} = \sum_{i=1}^{n_{\text{sbd}}} I_i \big(S_i^c\big)^{-1} I_i^t,$$

where $(S_i^c)^{-1}$ is a “constrained” inverse of the local Schur complement S_i. The application of $(S_i^c)^{-1}$ to a vector r involves the solution of the following (constrained) linear system

$$\begin{bmatrix} S_i & C_i^t \\ C_i & 0 \end{bmatrix} \begin{bmatrix} (S_i^c)^{-1} r \\ \lambda \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix},$$

where $C_i \in R^{n^i_{\text{cts}} \times n_i}$ is the matrix of constraints. Denoting by $l_i(a)$ the local numbering of $G_a \in \mathcal{G}_i$ at subdomain i, the $l_i(a)$ row of C_i (a constraint) is defined as

$$(C_i)_{l_i(a)p} = \begin{cases} 1 & \text{if } p \in G_a, \\ 0 & \text{otherwise,} \end{cases} \tag{10}$$

followed by a further scaling to have all rows with unit 1-norm. With such a definition, the product $z = C_i y$, with $z \in R^{n^i_{\text{cts}}}$ and $y \in R^{n_i}$, gives a vector y whose component $y_{l_i(a)}$ is related to object G_a. If G_a is a corner this component take the value of z in this corner, i.e., $y_{l_i(a)} = z_{l_i(a)}$; if G_a is a face (edge), it takes the mean value of z on the face (edge), i.e., $z_{l_i(a)} = |G_a|^{-1} \sum_{j \in G_a} y_j$. The three most common variants of the BDDC method, here referred as BDDC(c), BDDC(ce) and BDDC(cef), are based on only corner constraints, corner and edge constraints, and corner, edge and face constraints, respectively.

The coarse space $H_0 \subseteq V$ is defined as

$$H_0 = \operatorname{span}\big\{\phi^a : a = 1, \dots, n_{\text{cts}}\big\},$$

where every coarse function ϕ^a is associated with coarse object G_a. Coarse functions are constructed as the tensor product of local values, i.e., $\phi^a = \{\phi_i^a,\ i = 1, \dots, n_{\text{sbd}}\} \in V$. If $i \in N(G_a)$, ϕ_i^a is obtained as the solution of

$$\begin{bmatrix} S_i & C_i^t \\ C_i & 0 \end{bmatrix} \begin{bmatrix} \phi_i^a \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ e_{l_i(a)} \end{bmatrix},$$

where e_b denotes the b column of the identity matrix; $\phi_i^a = 0$ otherwise. Note that by its definition the coarse space is *non-conforming*, i.e., $H_0 \not\subset \widehat{V}$. The coarse-grid space basis functions define the mapping $I_0 : H_0 \to \widehat{V}$ as

$$I_0 \gamma = \sum_{a=1}^{n_{\text{cts}}} \sum_{i=1}^{n_{\text{sbd}}} I_i \phi_i^a \gamma_a, \tag{11}$$

and the coarse space operator as

$$(S_0)_{ab} = \sum_{i=1}^{n_{\text{sbd}}} R_i^t \phi_i^a S_i \phi_i^b R_i. \tag{12}$$

The final preconditioner can be written as an additive combination of a coarse and a fine-grid contribution

$$B_{\text{BDDC}}^{-1} = B_{\text{C}}^{-1} + B_{\text{F}}^{-1},$$

where $B_{\text{C}}^{-1} = I_0 S_0^{-1} I_0^t$ as in the BNN method. The condition number estimate (cf. [29])

$$\kappa\big(B_{\text{BDDC}}^{-1} S\big) \le C\left[1 + \frac{1}{d^2} \log^2\left(\frac{N}{P}\right)\right],$$

results in a constant number of PCG iterations (neglecting the logarithmic factor) for fixed load per core and increasing number of cores. For the 2D Poisson problem, BDDC(c) already achieves this bound, but for the 3D Poisson problem, at least BDDC(ce) is required [6].

3 Parallel Implementation

In this section we describe in detail a parallel distributed-memory implementation of DD methods of balancing type, namely the BNN and BDDC methods. This implementation inherits the two-level structure of the preconditioners subject of study. On the first level, the subdomains resulting from the non-overlapping partition of the global mesh are mapped to the MPI tasks, with a one-to-one mapping among subdomains, MPI tasks and computational cores of the underlying distributed-memory computer. On this level, all data structures required for the (preconditioned) iterative solution of the interface problem (3) are distributed among MPI tasks conformally with the underlying non-overlapping partition, and both computation and message-passing among MPI tasks are inherently of local nature, therefore, highly parallel. On the second level, the one corresponding to the global coupling among subdomains, the coarse-grid problem is assembled and solved serially on one processor (or all processors in the communicator) and therefore no parallelism is exploited at all.

For the sake of efficiency and portability, our implementation relies on several standard computational kernels provided by the dense/sparse BLAS and LAPACK, and highly efficient cache-aware vendor implementations of these kernels in order to achieve high flop rates on the computational core level (such as Intel MKL or IBM ESSL for Intel and IBM PPC multicore CPUs, respectively). Besides, through proper interfaces to third party libraries (e.g., PARDISO [36, 37]), the local Dirichlet (Schur complement-vector product) and Neumann (fine-grid correction) problems, as well as the global coarse-grid problem, are solved via cutting-edge sparse direct solvers. These solvers typically follow a super-nodal/multi-frontal approach for the efficient exploitation of the core cache subsystem thorough the level 3 BLAS [8]. The use of sparse direct methods is mandatory in the current setting as the iterative solution of (3) requires the Schur complement-vector to be computed *exactly*.[5] Besides, sparse direct solvers are highly robust numerically and very successful in the efficient exploitation of the cache hierarchy. The former factor is very helpful for the validation of computer implementations, while the latter one is becoming more important given the poor (main memory) bandwidth scalability in current/future multi-core/many-core CPUs.

The implementation is (essentially) split into two main phases. The first phase sets up the Schur complement and preconditioner before starting the iterations. This phase is in turn divided into a symbolic phase, where the geometrical information is computed beforehand (e.g., the sparsity pattern of a sparse Cholesky triangular factor), and a numerical phase, where the actual numerical computations take place. The second phase is the actual iterative solution of (3). Special attention will be paid to the application of the Schur complement and preconditioner to a vector. The rest of the section is structured as follows. Section 3.1 presents the data distribution and the general setting for a distributed-memory implementation of the PCG method. Sections 3.2 and 3.4 discuss the implementation of the computation and communication kernels required by the fine-grid preconditioning level, respectively, while Sects. 3.3 and 3.5 focus on those required by the coarse-grid preconditioning level.

3.1 Data Distribution and Basic Building Blocks

Algorithm 1 depicts the BDD-PCG iterative solver applied to the interface problem (3).[6] In a distributed-memory implementation of this algorithm, all vectors $y \in \widehat{V}$ and the Schur complement $S \in \widehat{V} \otimes \widehat{V}$ are partitioned and distributed among MPI tasks conformally with the underlying non-overlapping partition of the global interface. For those vectors $y \in \widehat{V}$ in Algorithm 1 which are naturally expressed as the assembly (sum) of subdomain contributions (in particular r, Ap, g), it is convenient (for reasons made clear below) that each MPI task keeps on its local address space *partially summed* contributions, e.g., g_i to g in (4). The same idea is applied to the interface block-matrix $A_{\Gamma\Gamma}$, where every processor stores local contributions $A^i_{\Gamma\Gamma}$ which, together with the corresponding *local* arrays $(A^i_{II}, A^i_{I\Gamma}, A^i_{\Gamma I})$, form partially summed contributions S_i to S in (4), as defined in (5). Our implementation does not compute/store explicitly S_i on each MPI task. Instead, the application of S to a vector is computed implicitly following the approach described below.[7]

Algorithm 1: Preconditioned Conjugate Gradient algorithm

```
BDD_PCG (Input: $(S, B_{\mathrm{BDD}}, g, x_0)$, Output: $x$)
$r_0 := g - Sx_0$
$z_0 := B_{\mathrm{BDD}}^{-1} r_0$ (see Algorithms 2 and 3)
$p_0 := z_0$
for $j = 0, 1, \ldots,$ till convergence do
    $\alpha_j := (r_j, z_j)/(Sp_j, p_j)$
    $x_{j+1} := x_j + \alpha_j p_j$
    $r_{j+1} := r_j - \alpha_j Sp_j$
    $z_{j+1} := B_{\mathrm{BDD}}^{-1} r_{j+1}$ (see Algorithms 2 and 3)
    $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
    $p_{j+1} := z_{j+1} + \beta_j p_j$
end for
```

On the other hand, for the rest of vectors $y \in \widehat{V}$ in Algorithm 3 (in particular, x, z, p), it is convenient that each MPI task keeps local *fully summed* (i.e., assembled) entries y_i, such that $y_i = R_i y$. Finally, any vector in the product space $v \in V$ is naturally distributed in such a way that each MPI task stores one local component $v_i \in V_i$ of v. Vectors $v \in V$ are not explicitly present in Algorithm 1, but as the result of intermediate steps during the application of the preconditioner. Following this approach, Algorithm 1 is therefore implemented in a subdomain-by-subdomain form, in the same way as in element-by-element techniques [13].

We next enumerate the basic building blocks of Algorithm 1, highlighting implementation details in our distributed-memory codes:

[5] BDD methods can certainly be reformulated as preconditioners for the global linear system (1). This enables approximate solvers (e.g., AMG [35, 40, 43]) to be used in conjunction with BDD methods. Although this approach relaxes the arithmetic/memory demands of sparse direct solvers (particularly in 3D), it would result in a new source of difficulties (e.g., robustness/complexity trade-off evaluation, parameter tuning, load unbalancing issues, poor flop rates) that deserve further research.

[6] For the BNN method, we assume that the initial value x_0 is such that r_0 is balanced (see [28]).

[7] An explicit assembly of S_i is required for some DD preconditioners (see e.g. [17]). Besides, this approach allows one to exploit the dense level 2 BLAS for the Schur complement-vector product, which can only compensate the expensive set-up of S_i for non-scalable preconditioners with high iteration counts.

Algorithm 2: $z := B_{\mathrm{BNN}}^{-1} r$

```
1: $z_{\mathrm{F}} := B_{\mathrm{NN}}^{-1} r$
2: $r := r - Sz$
3: $z_{\mathrm{C}} := B_{\mathrm{C}}^{-1} r$
4: $z := z_{\mathrm{F}} + z_{\mathrm{C}}$
```

Algorithm 3: $z := B_{\mathrm{BDDC}}^{-1} r$

```
1: $z_{\mathrm{F}} := B_{\mathrm{F}}^{-1} r$
2: $z_{\mathrm{C}} := B_{\mathrm{C}}^{-1} r$
3: $z := z_{\mathrm{F}} + z_{\mathrm{C}}$
```

Task 1 Vector updates in lines 6, 7 and 10. No communications are required. Each MPI task performs a local update using local data structures.

Task 2 Computation of inner products in lines 5 and 9. The computation of the inner product $z^t r$ is particularly simple as

$$z^t r = z^t \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t r_i = \sum_{i=1}^{n_{\mathrm{sbd}}} (R_i z)^t r_i = \sum_{i=1}^{n_{\mathrm{sbd}}} z_i^t r_i .$$

After the computation of local inner products $z_i^t r_i$, a global sum reduction operation (MPI_Reduce) is required to assemble the result on all MPI tasks.

Task 3 Application of the Schur complement operator in line 5. No communications are required. Each MPI task performs a local product $g_i = S_i y_i$. This local operation results in a vector g such that each MPI task keeps local *partially summed* contributions, i.e.:

$$g = Sy = \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t S_i R_i y = \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t S_i y_i = \sum_{i=1}^{n_{\mathrm{sbd}}} R_i^t g_i . \tag{13}$$

During Schur complement set-up, A_{II}^i is extracted from A^i and the sparse Cholesky factorization of A_{II}^i is computed. The product $S_i y_i$ is computed following a three-step algorithm: (1) compute $t = -A_{I\Gamma}^i y_i$;(2) solve $A_{II}^i u = t$; (3) $g_i = A_{\Gamma\Gamma}^i y_i + A_{\Gamma I}^i u$. A sparse backward/forward substitution is required for step (2), and one and two sparse matrix-vector products for steps (1) and (3), respectively. These sparse matrix-vector products are performed using a standard (sparse) level 2 BLAS kernel.

Task 4 The application of the preconditioner in lines 2 and 8 as shown in Algorithms 2 and 3 for the BNN and BDDC preconditioners, respectively. Here is where the two-level structure of both preconditioners is exposed. In the case of the BNN method, there is an additional residual update; see line 2 in Algorithm 2. The Schur complement-vector in this update is performed as in (13).

The first step in the application of the preconditioner is the computation of the fine-grid correction z_{F} (see line 1 in Algorithms 2 and 3). For both methods, the fine-grid preconditioner is applied to a global vector r (a residual) which is distributed in such a way that each MPI task keeps local *partially summed* contributions. In both the application of B_{NN}^{-1} and B_{F}^{-1}, the following computation has to be performed first:

$$I_i^t r = I_i^t \sum_{j=1}^{n_{\mathrm{sbd}}} R_j^t r_j , \tag{14}$$

which requires to obtain *fully summed* entries of r at each MPI task, followed by the application of the weighting matrix. The former fully summed assembly of r involves communication among nearest neighbors. The efficient implementation of this communication kernel is described in Sect. 3.4. The latter application of the weighting matrix is highly parallel as it can be applied locally on each MPI task. After the solution of local fine-grid problems (see Sect. 3.2), we obtain $s_i = S_i^\dagger I_i^t r$ and $s_i = (S_i^c)^{-1} I_i^t r$ for the BNN and BDDC methods, respectively, which define a global element of the product space $s_{\mathrm{F}} \in V$. Finally, it is transformed to a continuous $z_{\mathrm{F}} \in \widehat{V}$ as in (6). This operation also involves communication among nearest neighbors, as described in Sect. 3.4.

The second step in the application of the preconditioner is the computation of the coarse-grid correction z_{C} (see lines 3 and 2 in Algorithms 2 and 3, respectively). The coarse-grid preconditioner is also applied to a residual r which is distributed in such a way that each MPI task stores partially summed contributions. For both preconditioners the first step is a projection onto the coarse space. For the BNN method it reads as:

$$\left(I_0^t r\right)_i = \left(\phi^i\right)^t r = \left(\phi^i\right)^t \sum_{j \in N(i)} R_j^t r_j = \sum_{j \in N(i)} \left(R_j \phi^i\right)^t r_j , \tag{15}$$

where abusing the notation of Sect. 2.3, $N(i)$ is the set of subdomains neighboring subdomain i (included itself). For the BDDC method, since the coarse-grid space is non-conforming (see (11)), we compute $(I_0^t r)_a = \sum_{i \in N(a)} (\phi_i^a)^t I_i^t r$, where the computation of $I_i^t r$ is reused from the fine-grid component (see (14)). The solution of the global problem $\gamma = S_0^{-1} I_0^t r \in H_0$ is then injected into the fine-grid space. For the BNN method, since $H_0 \subset \widehat{V}$ and $z_{\mathrm{C}} = I_0 \gamma = \sum_{i=1}^{n_{\mathrm{sbd}}} \phi^i \gamma_i$, we can compute locally $(z_{\mathrm{C}})_i = R_i z_{\mathrm{C}} = \sum_{j \in N(i)} R_i \phi^j \gamma_j$. In the case of the BDDC method, we first compute the local components $\sum_{a \in \mathcal{G}_i} \phi_i^a \gamma_a$ of $s_{\mathrm{C}} \in V$ which has to be transformed to a continuous $z_{\mathrm{C}} \in \widehat{V}$ as described above with z_{F}. Note that as both the fine-grid and coarse-grid correction lead to $s_{\mathrm{F}} \in V$ and $s_{\mathrm{C}} \in V$, respectively, we can actually compute $s = s_{\mathrm{F}} + s_{\mathrm{C}}$ in line 3 of Algorithm 3, to finally obtain z from s as in (6). In other words, the above communication required to transform $s_{\mathrm{F}} \in V$ to a continuous $z_{\mathrm{F}} \in \widehat{V}$ can be postponed until the coarse-grid and fine-grid corrections are combined. The distributed-memory implementation of $I_0^t r$ and $I_0 \gamma$ requires global communications as described in Sect. 3.5. Besides, they require the coarse-grid basis functions which are obtained as the solution of local problems as described in Sect. 3.3. The assembly and solution of the coarse-grid problem is also described in Sect. 3.3.

As it is apparent from the description of Algorithm 1, the distributed-memory implementation of BDDC and BNN methods has only subtle differences, so that a common implementation framework for both has been used.

3.2 Fine-Grid Preconditioning Level

This section describes the computations to be performed in the fine-grid preconditioning level of the BNN (Sect. 3.2.1) and BDDC (Sect. 3.2.2) preconditioners. Special emphasis is put on the identification of the (dense/sparse) standard computational kernels and techniques that lead to an efficient implementation of the algorithms subject of study.

3.2.1 BNN

The implementation of the BNN fine-grid preconditioning level must take care of the efficient computation of $s_i = S_i^{\dagger} I_i^t r$. Recall from Sect. 2.3 that, when the subdomain is non-floating, $S_i^{\dagger} I_i^t r = S_i^{-1} I_i^t r$. This is (most) efficiently computed as the solution of the following linear system:

$$\begin{bmatrix} A^i_{II} & A^i_{I\Gamma} \\ A^i_{\Gamma I} & A^i_{\Gamma\Gamma} \end{bmatrix} \begin{bmatrix} t \\ s_i \end{bmatrix} = \begin{bmatrix} 0 \\ I_i^t r \end{bmatrix}. \tag{16}$$

On the other hand, when the subdomain is floating (i.e., S_i is singular), then $S_i^{\dagger} I_i^t r$ is computed as the solution of the following (constrained) local system:

$$\begin{bmatrix} A^i_{II} & A^i_{I\Gamma} & 0 \\ A^i_{\Gamma I} & A^i_{\Gamma\Gamma} & 1_i \\ 0 & 1_i^t & 0 \end{bmatrix} \begin{bmatrix} t \\ s_i \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ I_i^t r \\ 0 \end{bmatrix}, \tag{17}$$

where $\lambda \in \mathbb{R}$ is the Lagrange multiplier, with $\lambda = 0$ as $I_i^t r \in \ker(S_i)^{\perp} = \operatorname{span}(1_i)^{\perp}$ by construction of the BNN coarse-grid space (see Sect. 2.3). We stress that the constrained linear system (17) is symmetric indefinite but *non-singular*. Although λ is known in advance, the elimination of the third equation in (17) leads to a singular problem, therefore unsolvable by sparse direct solvers (this is indeed one of the main drawbacks of some BNN implementations).

The solution of (16) and (17) requires, during preconditioner set-up, the computation of a sparse direct factorization of their corresponding coefficient matrix, while the application of the preconditioner requires a sparse backward/forward substitution to finally obtain s_i. The algorithms for the direct solution of symmetric indefinite linear systems (e.g. (17)) are typically more expensive than those required for symmetric definite ones (e.g. (16)).[8] An alternative approach (also implemented in our codes) that we recommend to deal with floating subdomains is the one presented in [2]. Essentially, it is based on the observation that (17) can be transformed into an equivalent positive definite (PD) system by simply fixing judiciously (in particular by analyzing the kernel of S_i) picked degrees of freedom; for the Laplacian problem it simply reduces to fix one arbitrary degree of freedom. We refer the reader to [2] for a detailed explanation and comparison of both approaches for elasticity and Laplacian problems.

3.2.2 BDDC

The BDDC fine-grid preconditioning level is responsible for the computation of the product $s_i = (S_i^c)^{-1} I_i^t r$. This is obtained as the solution of the following constrained linear system:

$$\begin{bmatrix} A^i_{II} & A^i_{I\Gamma} & 0 \\ A^i_{\Gamma I} & A^i_{\Gamma\Gamma} & C_i^t \\ 0 & C_i & 0 \end{bmatrix} \begin{bmatrix} t \\ s_i \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ I_i^t r \\ 0 \end{bmatrix}. \tag{18}$$

There are two solution approaches for this symmetric indefinite (but non-singular) linear system. The first one is to tackle (18) directly using a sparse direct solver for symmetric indefinite linear systems. In such a case, a sparse direct factorization of the coefficient matrix is computed for the set-up of the preconditioner, while a sparse backward/forward substitution is required to compute s_i during preconditioner application. The second approach, originally presented in [10], exploits the particular structure of C_i to enable the exploitation of symmetric definite problems for the solution of (18). In the rest of this section we discuss this second approach and its efficient implementation.

Let us consider a reordering of (18) in such a way that FE equations/unknowns related to corners (C) are numbered first, followed by the rest (R) of nodes, ordered as internal nodes first, followed by nodes members of faces and nodes

[8] For example, PARDISO is based on the sparse Cholesky factorization without pivoting for symmetric positive definite problems, while for symmetric indefinite problems, it uses a more expensive sparse LDL^T factorization which, for numerical stability purposes, combines static (prior-to-factorization) pivoting via symmetric weighted matchings and classical Bunch-Kaufman dynamic (during factorization) pivoting only applied inside the supernodes [36, 37].

members of edges. Further, rows and columns of C_i related to nodes in $\mathcal{C}_i$ are labeled first, followed by those of $\mathcal{F}_i$ and $\mathcal{E}_i$. Let us also assume that the local ordering of corners is conformal with that of the rows of C_i corresponding to corner constraints. Then we obtain the following block reordered (constrained) linear system:

$$\begin{bmatrix} A^i_{CC} & (A^i_{RC})^t & I & 0 \\ A^i_{RC} & A^i_{RR} & 0 & (C^i_R)^t \\ I & 0 & 0 & 0 \\ 0 & C^i_R & 0 & 0 \end{bmatrix} \begin{bmatrix} x^i_C \\ x^i_R \\ \lambda^i_C \\ \lambda^i_R \end{bmatrix} = \begin{bmatrix} b^i_C \\ b^i_R \\ 0 \\ 0 \end{bmatrix}, \qquad (19)$$

where

$$A^i_{RR} = \begin{bmatrix} A^i_{II} & A^i_{IF} & A^i_{IE} \\ A^i_{FI} & A^i_{FF} & A^i_{FE} \\ A^i_{EI} & A^i_{EF} & A^i_{EE} \end{bmatrix},$$

$$C^i_R = \begin{bmatrix} 0 & C^i_F & 0 \\ 0 & 0 & C^i_E \end{bmatrix},$$

and λ^i_C and λ^i_R are the Lagrange multipliers associated to corner constraints and the rest of constraints, respectively. The particular block structure of C_i is easily derived from its definition in (10). Although the permutation matrix which leads to (19) from (18), and its inverse, have been omitted for simplicity, note that $(b^i_C\ b^i_R)^t$ is obtained from $(0\ I^t_i r)^t$ and $(t\ s_i)^t$ from $(x^i_C\ x^i_R)^t$ after the application of the former and the latter permutations, respectively. In our implementation, both permutations are explicitly computed (and stored as usual in permutation arrays) as they are used for the extraction of those blocks of (19) which are required for its solution (see below).

The solution of (19) is computed as follows. From the third and second block equations of (19) we have that $x^i_C = 0$ and

$$x^i_R = \left(A^i_{RR}\right)^{-1} b^i_R - \left(A^i_{RR}\right)^{-1} \left(C^i_R\right)^t \lambda^i_R, \qquad (20)$$

respectively, where A^i_{RR} is a *large, sparse, symmetric positive definite* matrix. If the first two block equations are eliminated from (19), Lagrange multipliers λ^i_R are obtained as the solution of the following system:

$$C^i_R \left(A^i_{RR}\right)^{-1} \left(C^i_R\right)^t \lambda^i_R = -\left(C^i_R\right)^t \left(A^i_{RR}\right)^{-1} b^i_R, \qquad (21)$$

where $C^i_R(A^i_{RR})^{-1}(C^i_R)^t$, the Schur complement matrix associated to λ^i_R, is a *small, dense, symmetric positive definite* matrix (of size $n^F_i + n^C_i$). During preconditioner set-up, the following four tasks are performed: (1) compute the sparse Cholesky factorization of A^i_{RR}; (2) compute $(A^i_{RR})^{-1}(C^i_R)^t$ by means of a kernel which allows the exploitation of the level 3 BLAS during the (blocked) sparse backward/forward substitution, and store the result in a dense work array for later use; (3) compute $C^i_R(A^i_{RR})^{-1}(C^i_R)^t$ (reusing $(A^i_{RR})^{-1}(C^i_R)^t$ from step (2)); (4) compute a dense Cholesky factorization of $C^i_R(A^i_{RR})^{-1}(C^i_R)^t$ using the corresponding LAPACK kernel. We stress that $(C^i_R)^t$ is stored in dense storage mode as required by the kernel exploited in step (2). However, C^i_R is not stored. Instead, it is more efficient to implement the matrix-vector (and matrix-matrix) multiplication using a subroutine which generates its entries "on the fly" to save storage and that only operates with the non-zero entries of C^i_R to save floating-point calculations. On the other hand, during the preconditioner application, the following tasks are performed: (1) solve $A^i_{RR}t^i_R = b^R_i$ by sparse backward/forward substitution; (2) compute $w^i_R = C^i_R t^i_R$; (3) solve $C^i_R(A^i_{RR})^{-1}(C^i_R)^t\lambda^i_R = -w^i_R$ using the corresponding level 2 BLAS kernel for the triangular solution of dense linear systems; (4) compute $x^i_R = t^i_R - (A^i_{RR})^{-1}(C^i_R)^t\lambda^i_R$. The computation of step (4) requires a former level 2 BLAS dense matrix-vector product. The dense work array setup during preconditioner construction for the storage of $(A^i_{RR})^{-1}(C^i_R)^t$ is reused for this product.

3.3 Coarse-Grid Preconditioning Level

In this section we focus on the implementation details of the second level in the two-level structure of the BNN and BDDC preconditioners, namely the coarse-grid preconditioning level. Duties on this level include: (1) the computation of coarse-grid space basis functions during preconditioner set-up; (2) the computation of the contribution of each subdomain to the coarse-grid coefficient matrix S_0 and coarse-grid residual, during preconditioner set-up and application, respectively; (3) the assembly, Cholesky factorization of S_0 and solution of the coarse-grid system during preconditioner set-up and application, respectively. Although there are several implementation approaches for (3) in a distributed-memory code (e.g., in parallel on all or a subset of processors in the global communicator), we turn our attention into a MPI implementation that solves the coarse-grid problem serially. Apart from the evaluation of the scalability of this solution, another purpose of this paper is to determine whether it is more efficient to assemble/factorize/solve the coarse-grid problem on *one processor*, and then distribute the solution over the rest of processors, or to assemble/factorize/solve an identical problem *on all processors*, where no global communication is required afterwards. The solution on one or all processors decision depends on the extra synchronization and communication overhead incurred by the global collectives required to implement each option. Section 3.5 presents these collectives and evaluates their performance and scalability on a large-scale distributed-memory machine, and Sect. 4 studies the weak scalability of the coarse-grid preconditioning level, paying special attention to the one or all processors decision. The rest of the current section describes in detail implementation considerations of this preconditioning level.

We denote by Φ_i the matrix whose columns are the local coarse basis functions. In the case of the BNN method they are $R_i\phi^j$ for $j \in N(i)$. In the case of the BDDC method they are ϕ_i^a for $a \in \mathcal{G}_i$ (see Sect. 2.4). In the case of the BDDC method, the computation of Φ_i involves the solution of the following (constrained) sparse linear system with several right hand sides:

$$\begin{bmatrix} A^i_{II} & A^i_{I\Gamma} & 0 \\ A^i_{\Gamma I} & A^i_{\Gamma\Gamma} & C^t_i \\ 0 & C_i & 0 \end{bmatrix} \begin{bmatrix} \beta_i \\ \Phi_i \\ \Lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ I \end{bmatrix}, \qquad (22)$$

where ϕ_i^a, i.e., the a-column of Φ_i, is the restriction of a (corner, edge, or face) constraint basis function to subdomain i, β_i^j is the discrete harmonic extension of ϕ_i^j, λ_i^j is the vector of Lagrange multipliers, and I is the identity matrix of size n^i_{cts}. The solution of (22) reuses the data structures which are computed during the set-up of the fine-grid preconditioner; see Sect. 3.2.2. Any of the two approaches described for the solution of (18) can be used for (22). However, the efficient computation of (22) is carried out with dense level 3 BLAS (e.g., the solution of a dense triangular linear system with several right hand sides) and sparse kernels which allow the exploitation of the level 3 BLAS during the (blocked) sparse backward/forward substitution. Besides, given the identity matrix on the right hand side of (22) (instead of a zero matrix as in (18)), A_{RC} has also to be extracted from (19) during preconditioner set-up and stored as a dense matrix (as it becomes the input of a kernel for the solution of triangular systems with several right hand sides).

Coarse-grid preconditioning level duties also include the computation of the contribution of each subdomain to the coarse-grid coefficient matrix S_0 and coarse-grid residual. The sparse matrix S_0 can be obtained as the assembly (sum) of subdomain contributions $\sum_{i=1}^{n_{\text{sbd}}}(\Phi_i)^t S_i \Phi_i$ as in (12). In the case of the BNN method, the computation of $S_i\Phi_i$ is split (as in (13)) into two steps: (1) solve $A^i_{II}\beta_i = -A^i_{I\Gamma}\Phi_i$; (2) $S_i\Phi_i = A^i_{\Gamma I}\beta_i + A^i_{\Gamma\Gamma}\Phi_i$. Step (1) is efficiently computed with a kernel for the sparse direct solution of linear systems with several right hand sides, i.e., a kernel which allows the exploitation of the level 3 BLAS during the (blocked) sparse backward/forward substitution. Step (2) requires a pair of sparse-dense matrix-matrix multiplications, which are efficiently computed using the corresponding kernel in the (sparse) level 3 BLAS. In case of the BDDC method, the computation of $S_i\Phi_i$ comes almost for free as $S_i\Phi_i = -C_i^T\Lambda_i$ (see (22)). Once $S_i\Phi_i$ is computed, the computation of $\Phi_i^t S_i \Phi_i$ just requires a further dense matrix-matrix multiplication, which for portability and efficiency, is performed using the corresponding kernel in the level 3 BLAS. On the other hand, the computation of the contribution of subdomain i to the coarse-grid residual during preconditioner application, i.e., $\Phi_i^t r_i$, is most conveniently carried out with a level 2 BLAS matrix-vector multiplication.

Finally, the coarse-grid preconditioning level is responsible for the assembly, the Cholesky factorization of S_0 and the solution of the coarse-grid system during preconditioner set-up and application, respectively. For convenience, we split the presentation into these two phases.

Coarse Preconditioner Set-up In a first symbolic phase, the adjacency graph of the sparse coarse-grid coefficient matrix S_0 is built on one or all processors. Two steps are required: (a) the computation of a global ordering of coarse-grid nodes (objects for BDDC and subdomains for BNN); (b) the computation of the global (although sparse) coupling among coarse-grid nodes. Step (a) is naive in the case of the BNN method, as this ordering already coincides with the global ordering of the subdomains. However, in the case of the BDDC method, its construction requires that one or all processors gather all the coarse-grid nodes each MPI task has identified on its local interface. The labeling of coarse-grid nodes is greatly simplified in our implementation by the fact that coarse-grid nodes are extracted from communication objects on the local interface (see Sect. 3.4 and Fig. 1), which are already labeled with a global identifier. Step (b) depends on the support of coarse basis functions on Γ. In the case of the BDDC method, the support of these functions is such that a given coarse-grid node is coupled with all the coarse-grid nodes identified on the subdomains that surround the node. Therefore, it is sufficient that the processor(s) in charge of S_0 gather, per each subdomain, the list of subdomains that surround each coarse-grid node. In case of the BNN method, the support of basis functions is such that two coarse-nodes are coupled if they are neighbors or neighbors of neighbors. Therefore, to determine the sparsity pattern of S_0, it is sufficient to gather, per each subdomain, its list of neighboring subdomains. This global data structure is referred on the literature as subdomain graph or partition graph (see, e.g. [31, 33, 34]). In a second numerical phase, the matrix S_0 is both assembled and then factorized on one or all processors. In order to do so, processor(s) in charge of S_0 gather subdomain local contributions $\Phi_i^t S_i \Phi_i$, and then assemble them into S_0. The correspondence among the entries of each subdomain elemental matrix and S_0 for this assembly process is given by the global ordering (pre)computed on step (a).

Coarse Preconditioner Application The processor(s) responsible for the coarse-grid problem solution first gather subdomain local contributions (see (15) and then assemble them to build the coarse-grid residual. The backward/forward substitution with the Cholesky factor of S_0

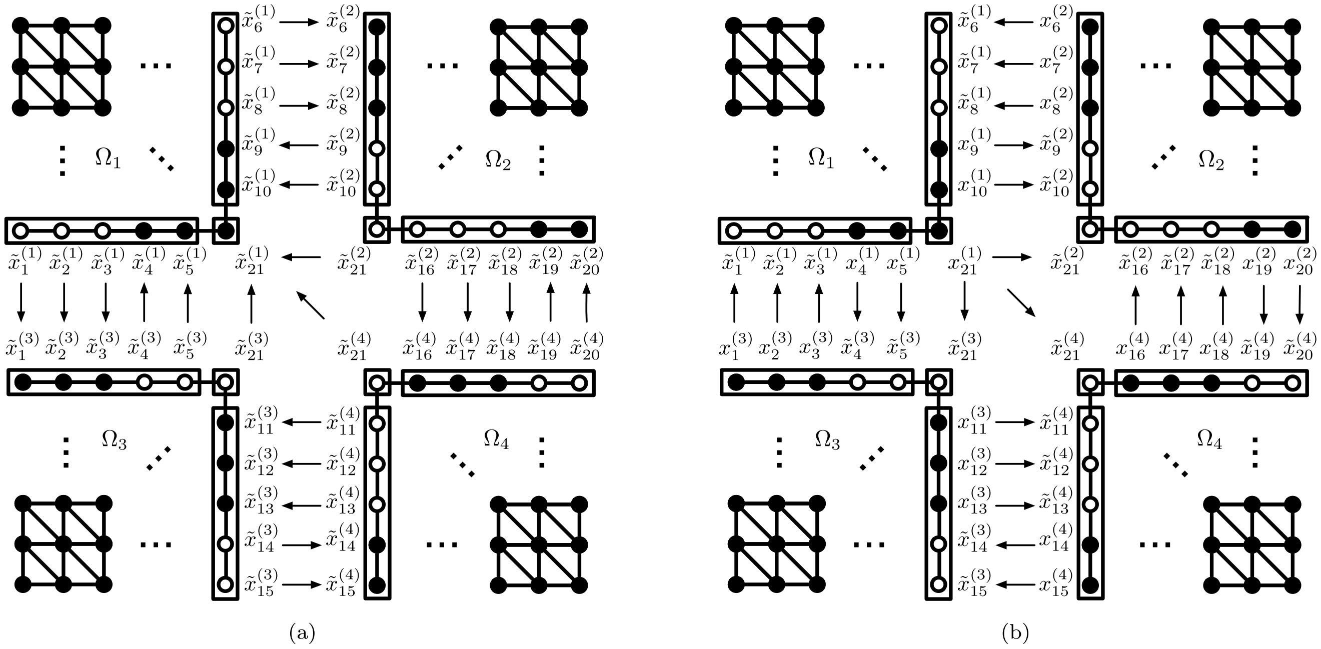

Ω_1
Ω_2
Ω_3
Ω_4
$\tilde{x}_6^{(1)}$ $\tilde{x}_7^{(1)}$ $\tilde{x}_8^{(1)}$ $\tilde{x}_9^{(1)}$ $\tilde{x}_{10}^{(1)}$
$\tilde{x}_6^{(2)}$ $\tilde{x}_7^{(2)}$ $\tilde{x}_8^{(2)}$ $\tilde{x}_9^{(2)}$ $\tilde{x}_{10}^{(2)}$
$\tilde{x}_1^{(1)}$ $\tilde{x}_2^{(1)}$ $\tilde{x}_3^{(1)}$ $\tilde{x}_4^{(1)}$ $\tilde{x}_5^{(1)}$ $\tilde{x}_{21}^{(1)}$ $\tilde{x}_{21}^{(2)}$ $\tilde{x}_{16}^{(2)}$ $\tilde{x}_{17}^{(2)}$ $\tilde{x}_{18}^{(2)}$ $\tilde{x}_{19}^{(2)}$ $\tilde{x}_{20}^{(2)}$
$\tilde{x}_1^{(3)}$ $\tilde{x}_2^{(3)}$ $\tilde{x}_3^{(3)}$ $\tilde{x}_4^{(3)}$ $\tilde{x}_5^{(3)}$ $\tilde{x}_{21}^{(3)}$ $\tilde{x}_{21}^{(4)}$ $\tilde{x}_{16}^{(4)}$ $\tilde{x}_{17}^{(4)}$ $\tilde{x}_{18}^{(4)}$ $\tilde{x}_{19}^{(4)}$ $\tilde{x}_{20}^{(4)}$
$\tilde{x}_{11}^{(3)}$ $\tilde{x}_{12}^{(3)}$ $\tilde{x}_{13}^{(3)}$ $\tilde{x}_{14}^{(3)}$ $\tilde{x}_{15}^{(3)}$
$\tilde{x}_{11}^{(4)}$ $\tilde{x}_{12}^{(4)}$ $\tilde{x}_{13}^{(4)}$ $\tilde{x}_{14}^{(4)}$ $\tilde{x}_{15}^{(4)}$
(a)
$\tilde{x}_6^{(1)}$ $\tilde{x}_7^{(1)}$ $\tilde{x}_8^{(1)}$ $x_9^{(1)}$ $x_{10}^{(1)}$
$x_6^{(2)}$ $x_7^{(2)}$ $x_8^{(2)}$ $\tilde{x}_9^{(2)}$ $\tilde{x}_{10}^{(2)}$
$\tilde{x}_1^{(1)}$ $\tilde{x}_2^{(1)}$ $\tilde{x}_3^{(1)}$ $x_4^{(1)}$ $x_5^{(1)}$ $x_{21}^{(1)}$ $\tilde{x}_{21}^{(2)}$ $\tilde{x}_{16}^{(2)}$ $\tilde{x}_{17}^{(2)}$ $\tilde{x}_{18}^{(2)}$ $x_{19}^{(2)}$ $x_{20}^{(2)}$
$x_1^{(3)}$ $x_2^{(3)}$ $x_3^{(3)}$ $\tilde{x}_4^{(3)}$ $\tilde{x}_5^{(3)}$ $\tilde{x}_{21}^{(3)}$ $\tilde{x}_{21}^{(4)}$ $x_{16}^{(4)}$ $x_{17}^{(4)}$ $x_{18}^{(4)}$ $\tilde{x}_{19}^{(4)}$ $\tilde{x}_{20}^{(4)}$
$x_{11}^{(3)}$ $x_{12}^{(3)}$ $x_{13}^{(3)}$ $\tilde{x}_{14}^{(3)}$ $\tilde{x}_{15}^{(3)}$
$\tilde{x}_{11}^{(4)}$ $\tilde{x}_{12}^{(4)}$ $\tilde{x}_{13}^{(4)}$ $x_{14}^{(4)}$ $x_{15}^{(4)}$
(b)

Fig. 1 Illustration of the two nearest neighbor exchange phases required to obtain a distributed vector stored in fully summed form from the same distributed vector stored in partially summed form. The computational mesh has been partitioned into 4 subdomains, with 5 communication objects on the interface. Four communication objects are shared by subdomains 1-3, 1-2, 3-4, and 2-4, respectively, while the remaining object is shared among all subdomains. The owner subdomain of a given interface point is represented in *black*, while a non-owner one in *white*

provides the preconditioned coarse-grid residual, which is then distributed among all processors only in case one processor is responsible for the solution of the coarse-grid residual.

3.4 Local Nearest Neighbor Exchange Communications

A basic communication kernel to be implemented in non-overlapping DD preconditioners is the one that given a distributed vector stored in partially summed form returns after communication the same distributed vector stored in fully summed form, see (6) and (14). In contrast to global dense collectives such as all-to-all, gather or scatter, this collective communication is highly sparse as it only requires communication among (a moderate number of) nearest neighbors. For example, for structured meshes, each part has only 8 and 26 neighbors in 2D and 3D, respectively. An implementation of this kernel requires a representation of the local interface of each subdomain in terms of *communication objects*. Communication objects embrace all the nodes of the mesh interface that are shared by the same subdomains. Each mesh point residing on the interface is assigned an *owner* subdomain. The rest of subdomains sharing this mesh point automatically become non-owners. See Fig. 1 for a graphical representation of these concepts. In a first data exchange among nearest neighbors, non-owner subdomains send their local contributions to the owner subdomain, while the owner subdomain is in charge for data accumulation and update to obtain fully summed entries. This is illustrated in Fig. 1(a). For example, subdomains 2, 3 and 4 send $\tilde{x}_{21}^{(2)}$, $\tilde{x}_{21}^{(3)}$ and $\tilde{x}_{21}^{(4)}$, respectively, while subdomain 1 receives and accumulates them to obtain $x_{21}^{(1)}$. Then, in a second data exchange, owners send copies of its fully summed entries to the non-owner sides, which just copy them into their local data structures. This exchange phase is illustrated in Fig. 1(b). The strategy based on owner and non-owner subdomains of mesh interface points is also covered in [34].

There are two main implementation issues that significantly influence the performance and scalability of the data exchanges among nearest neighbors. The first one targets the strategy to determine the owner subdomain of each mesh interface point, as this strategy influences the trade-off among message size and number of messages to be exchanged. Our implementation relies on a very simple strategy (that we demonstrate afterwards to be quite efficient): (1) objects shared by two subdomains are divided equally among them (i.e., one subdomain is the owner of the first half, while the other of the second half), so that message sizes to be exchanged on each side are balanced; (2) objects shared by more than two parts are (arbitrarily) owned by the part with minimum identifier. This results in a trade-off which chooses a smaller number of larger messages over a larger number of smaller messages (e.g., assuming a given object is shared among n parts, a n -to-1 followed by a 1-to-n com-

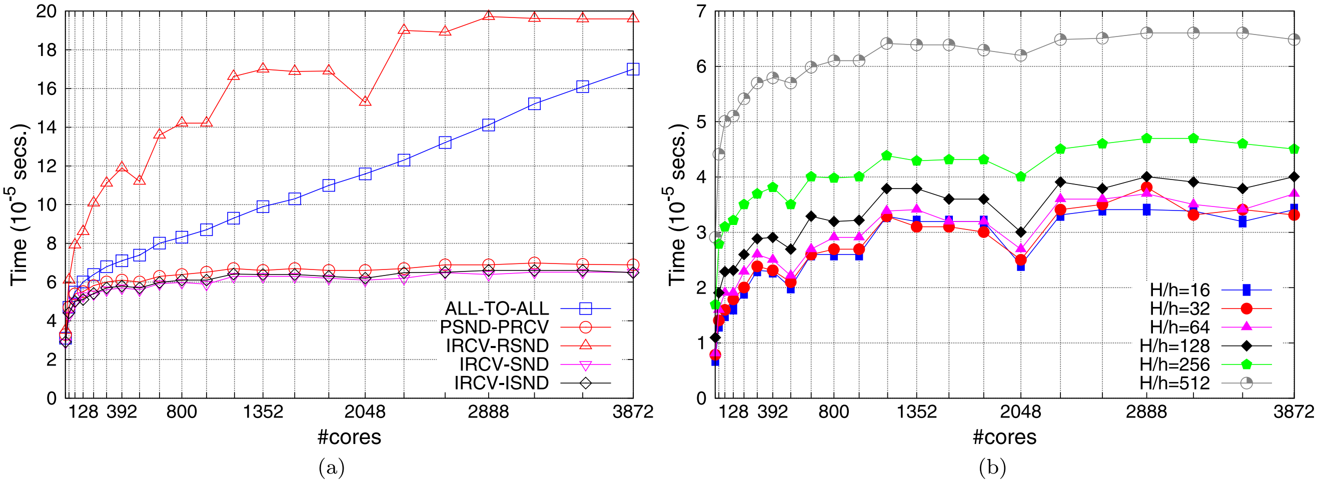


Fig. 2 Weak scaling for the two exchange phases illustrated in Fig. 1. (**a**) Several implementations and fixed local problem size $\frac{H}{h} = 512$. (**b**) IRCV_ISND implementation for several local problem sizes $\frac{H}{h}$

munication pattern is preferred over two consecutive n-to-n communication patterns with smaller messages). The second issue targets the MPI implementation of each data exchange phase, as the MPI standard does not currently support sparse collectives. There are two possible solutions to bypass this limitation: (a) to use existing MPI *dense* collectives, in particular, irregular (vector), personalized all-to-all exchange, in which no data are exchanged between processes that are not neighbors (i.e., `MPI_Alltoallv`). This solution has several disadvantages as discussed in detail in [25]; (b) to implement the sparse collective by means of point-to-point communication operations. This is the solution followed by the vast majority of existing distributed-memory linear algebra codes. After a comprehensive literature (see, e.g. [21, 22]) and parallel software (e.g., TRILINOS [23, 24], PETSC [4, 5], PSBLAS [14, 15]) review, existing general-purpose solutions can be categorized as follows depending on the order on which the send/receive operations are issued and the particular blocking semantics of the point-to-point communication operations:

- PSND-PRCV. Each MPI rank traverses its local neighborhood and issues a send operation per neighbor. Then, in a second traversal, it issues a receive operation per neighbor. Send operations are *locally blocking* while receive operations are blocking (i.e., MPI_Recv). Locally blocking semantics ensure that the send operation immediately returns the control to the application after the message to be sent has been copied in an intermediate buffer and therefore avoids the potential deadlock of blocking sends (i.e., MPI_Send). This is the strategy followed by the PSBLAS library, which in turn inherited the locally blocking semantics from the BLACS [15].
- IRCV-RSND. All receive operations are issued first, then all send operations. Receive operations are non-blocking (i.e., MPI_IRecv), while send operations are ready blocking (i.e., MPI_Rsend). A global barrier operation (i.e., MPI_Barrier) is issued between receive and send operations as required by the ready blocking semantics. At the end, processes wait for the non-blocking receive operations to complete (i.e., MPI_Waitall). This is the strategy implemented in Epetra-Trilinos [23].
- IRCV-SND. Same as IRCV-RSND but ready blocking sends are replaced by blocking sends (i.e., MPI_Recv). No barrier is required in between sends and receives.
- IRCV-ISND. Same as IRCV-SND but blocking sends are replaced by non-blocking sends (i.e., MPI_ISend). At the end, processes wait for both the non-blocking receives and send communication operations.

In order to provide some evidence with respect to the performance and scalability of the communication kernel covered in this section, Fig. 2 illustrates typical weak scaling curves for the parallel execution time required to perform the two nearest neighbor exchange phases on the HPC-FF (see Sect. 4.1). We focus on 2D structured meshes of quadrilateral elements, although similar conclusions can be raised in the 3D structured case. We refer to Sect. 4.1 for a detailed description of the experiment set-up. Figure 2(a) compares the parallel execution time of the aforementioned MPI implementations when increasing the number of cores while keeping fixed the local problem size to $(\frac{H}{h})^2 = 512^2$ quadrilaterals. This figure shows that PSND-PRCV, IRCV-SND and IRCV-ISND are weakly scalable MPI implementations, as they reach asymptotic parallel execution time (of approximately 65 μ-seconds). However, the performance of IRCV-RSND significantly degrades with the number of cores. On the HPC-FF, it seems that any performance gain obtained as by-product of the ready blocking semantics (hand-shaking and intermediate buffer copying removal) does not pay off the overhead associated to the barrier required in between receives and sends, which introduces a global synchronization point that limits the amount of parallelism by exacerbating the effects of load imbalance. The performance and scalability of the implementation based on MPI_Alltoallv is not surprising as it does not properly capture the parallelism inherent to the sparse communication pattern (as pointed out in [25]). Figure 2(b) illustrates the results of the same weak scaling study of Fig. 2(a), but it focus on one of the three most efficient implementations (i.e., IRCV-ISND) with problem size fixed to several values. Absolute timings and the order of complexity with $\frac{H}{h}$ shown in Fig. 2(b) reveal that, *as long as a weak scalable implementation such as IRCV-ISND is employed*, the contribution of this communication kernel to the overall performance of BDD methods can be considered negligible compared to that of other building blocks in this family of algorithms, such as, e.g., the solution of local Dirichlet problems or the computation of a coarse-grid correction.

3.5 Global Collectives

The global MPI communication operations required for the implementation of the coarse-grid preconditioning level in the one processor case are MPI_Gatherv (twice for the preconditioner set-up and once per preconditioner application) and MPI_Scatterv (once per preconditioner application), while MPI_Allgatherv (twice for the preconditioner set-up and once per preconditioner application) is required in case the coarse-grid problem is solved on all processors. These varying message size collectives are the ones that most accurately capture the global communication pattern involved in the (serial) solution of the coarse-grid problem. For structured meshes and regular partitions, boundary subdomains do not send/receive the same amount of data than internal ones, and for unstructured ones, the shape of each subdomain local interface is strongly dependent on the underlying (irregular) non-overlapping partition. However, we have experimentally observed on several distributed-memory platforms that the performance of the solution of the coarse-grid problem may benefit from exploiting fixed message size collectives, namely MPI_Gather/MPI_ Scatter/MPI_Allgather. Figure 3 depicts what we have observed on the HPC-FF supercomputer. In particular, it reports the parallel execution time with increasing number of cores for fixed and varying message size collectives for powers-of-two message sizes in the range 32-8192 bytes; all message sizes exchanged by BDD methods in the case of structured 2D/3D meshes are enclosed within this range. Figure 3 reveals that for message sizes below or equal to 512 bytes the performance of MPI_Gather is superior to that of MPI_Gatherv, and the smaller the message size the more superior MPI_Gather over MPI_Gatherv. We can observe just the opposite for message sizes beyond 512 bytes, with the largest gains of MPI_Gatherv over MPI_Gather with the largest message sizes. In the case of scatter communication, a much larger message size of approximately 4096 bytes is required by MPI_Scatterv to become superior to MPI_Scatter. Besides, for "small" message sizes, larger gains of MPI_Scatter over MPI_ Scatterv are attained compared to those observed for gather communication. Finally, the performance of MPI_AllGatherv and MPI_AllGather collectives is almost coincident.

4 Scalability Study

4.1 Experimental Framework

The algorithms subject of study were implemented in FEMPAR, an in-house, developed from scratch, OO framework which, among other features, provides the basic tools for the efficient message-passing (MPI) implementation of substructuring DD solvers, using METIS [26] for unstructured meshes. All experiments reported in the sequel were obtained on a large-scale multicore-based distributed-memory machine, the HPC-FF (HPC for Fusion), located at the Juelich (Germany) Supercomputing Centre. The HPC-FF is a QDR Infiniband interconnected commodity cluster composed of 1080 Bull NovaScale R422-E2 blades. Each blade is equipped with two Intel Xeon X5570 QuadCore processors running at 2.93 GHz (8 computational cores in total) and 24 GBytes of DDR3 memory, and runs a full-featured SUSE SLES 11 Linux OS. The codes were compiled using Intel Fortran compiler (12.1.4) with recommended optimization flags and we used Parastation 5.0 MPI tools and libraries for native message-passing. The codes were linked against the BLAS/LAPACK and PARDISO available on the Intel MKL library (version 10.3, build 10). Peak flop performance per core is 11.72 GFLOPs/sec (i.e., 93.76 GFLOPs/sec per blade) and measured MPI intrasocket, intersocket and internode latency and bandwidth for this machine are 0.26 μ-seconds and 4.6 GBytes/sec, 0.57 and 3.7, and 1.49 and 3.1, respectively. We stress that we have also evaluated the codes on several radically different platforms (e.g., MareNostrum, a Myrinet-interconnected cluster composed of 2560 IBM JS21 compute nodes at the Barcelona Supercomputing Center). We skip the corresponding results because similar balances to those reported next for the HPC-FF were achieved.

The experimental study in this paper focuses on the evaluation of the *weak scalability* of several sub-structuring DD solvers when applied to the Poisson problem. Recall that weak scaling studies determine at which rate a given magnitude evolves with the number of cores P while keeping the local problem size $\frac{H}{h}$ constant. In particular, mag-

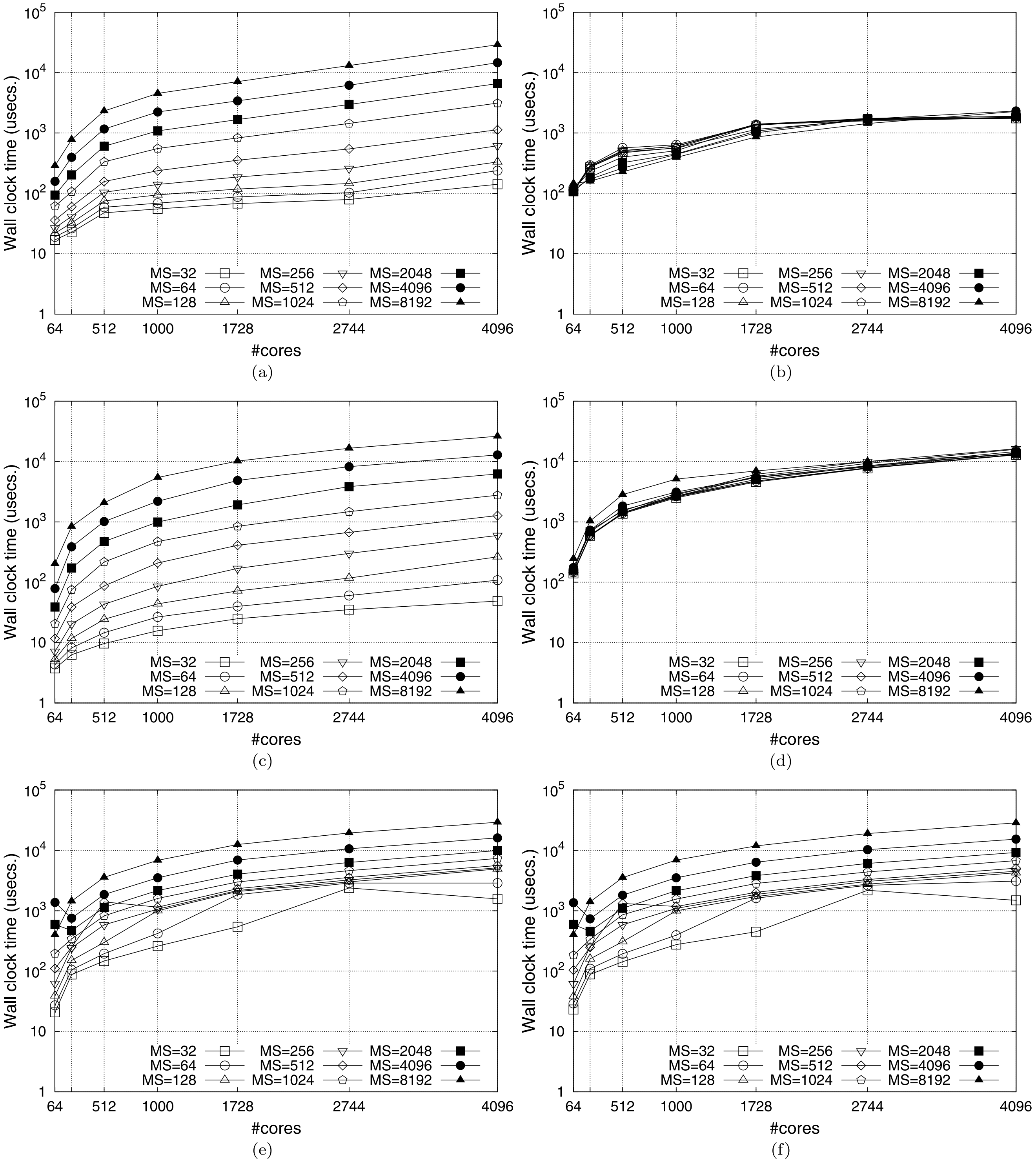


Fig. 3 Performance and scalability of fixed and varying message size collectives on the HPC-FF supercomputer. Fixed: (**a**) MPI_Gather; (**c**) MPI_Scatter; (**e**) MPI_Allgather. Varying: (**b**) MPI_Gatherv; (**d**) MPI_Scatterv; (**f**) MPI_Allgatherv

nitudes of interest for our study are the total computation time, and the number of PCG iterations required to solve the preconditioned interface problem (3). The former magnitude is in turn concentrated on three phases: the Schur-complement system and preconditioner set-up, and the iterative solution of (3) by the PCG Krylov subspace solver. In Sects. 4.1.1, and 4.1.2 we introduce the particular ranges of P and $\frac{H}{h}$ that our study explores for 2D and 3D, respec-

Table 1 Arithmetic complexities of the different stages in the serial direct solution of sparse linear systems

Phase	2D complexity ($d = 2$)	3D complexity ($d = 3$)
Reordering	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Symbolic factorization	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{\frac{4}{3}})$
Numerical factorization	$\mathcal{O}(n^{\frac{3}{2}})$	$\mathcal{O}(n^2)$
Triangular solution	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{\frac{4}{3}})$

Table 2 Parallel complexity for the NN-PCG and BDD-PCG methods. Sparse direct solvers are assumed as well as the serial solution of the coarse-grid problem. It is obtained as the sum of the set-up phase complexity, with unknown constant c_s, and the iterative phase complexity, with unknown constant c_i. In the case of BDD-PCG methods, the unknown constants c_s and c_i are separated into coarse-grid (c) and fine-grid (f) preconditioning contributions

Method	2D complexity ($d = 2$)	3D complexity ($d = 3$)
NN-PCG	$c_s n^{\frac{3}{2}} + c_i \sqrt{P} n \log(n)$	$c_s n^2 + c_i \sqrt[3]{P} n^{\frac{4}{3}}$
BDD-PCG	$c_{sf} n^{\frac{3}{2}} + c_{sc} P^{\frac{3}{2}} + c_{if} n \log(n) + c_{ic} P \log P$	$c_{sf} n^2 + c_{sc} P^2 + c_{if} n^{\frac{4}{3}} + c_{ic} P^{\frac{4}{3}}$

tively, and how the problem is mapped to the underlying computer.

4.1.1 *Set-up for 2D Experiments*

We consider the solution of the Poisson problem on a rectangle $\overline{\Omega} = [0, 2] \times [0, 1]$, a global conforming uniform mesh (partition) of $\overline{\Omega}$ into quadrilaterals, and a bilinear finite element discretization (i.e., Q1-elements). The 2D mesh was partitioned into rectangular grids of $P = 4m \times 2m$ square subdomains, and distributed over $m = 1, 2, \ldots, 22$ nodes, with 4×2 subdomains/MPI Ranks per node and one MPI Rank per core of the HPC-FF. In order to evaluate the weak scaling of the solvers under several computation/communication balances, we consider increasing values for $\frac{H}{h} = 16, 32, 64, 128, 256$ and 512, with the two extremes being the most and least communication-bounded scenarios of the sample. The largest problem size $\frac{H}{h} = 512$ was selected strategically to be the largest power-of-two that fits into the machine given a memory limit per core of 1.7 GBytes. Note that in 2D the number of quadrilaterals on each local mesh is therefore $\frac{H}{h} \times \frac{H}{h}$, and that of the global mesh is given by $4m\frac{H}{h} \times 2m\frac{H}{h}$.

4.1.2 *Set-up for 3D Experiments*

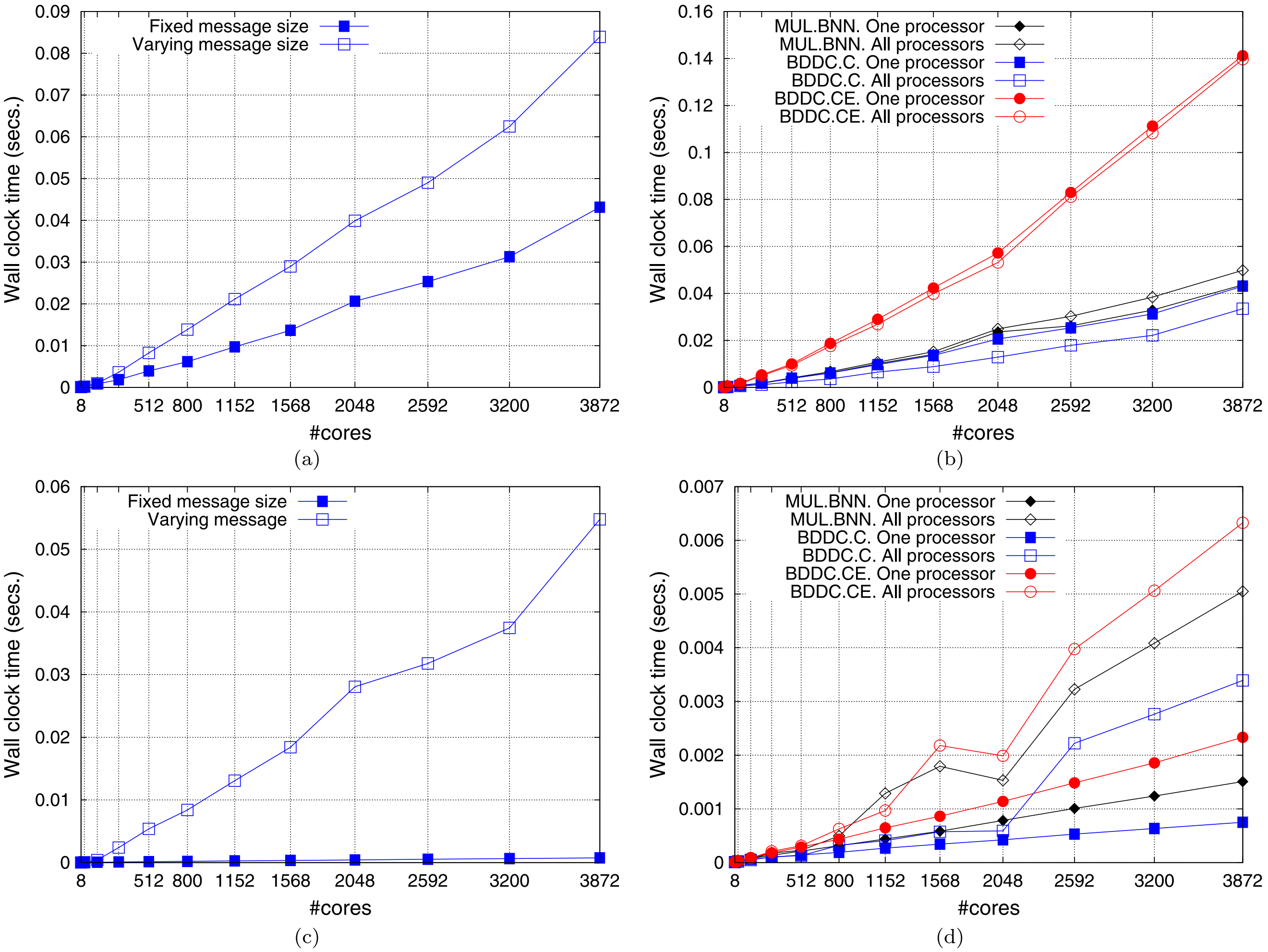
We consider the solution of the Poisson problem on a cube $\overline{\Omega} = [0, 1] \times [0, 1] \times [0, 1]$, a global conforming uniform mesh (partition) of $\overline{\Omega}$ into hexahedra and a trilinear finite element discretization (i.e., Q1-elements). The 3D mesh is partitioned into cubic grids of $P = 2m \times 2m \times 2m$ cubic subdomains and distributed over $m = 1, 2, \ldots, 8$ nodes, with $2 \times 2 \times 2$ subdomains/MPI Ranks per node and one MPI Rank per core of the HPC-FF. We consider increasing values for $\frac{H}{h} = 10, 20, 30$ and 40. The largest problem size $\frac{H}{h} = 40$ was selected strategically to be the largest multiple-of-ten that fits into the machine given a memory limit per core of 1.7 GBytes. Note that in 3D the number of hexahedra on each local mesh is $\frac{H}{h} \times \frac{H}{h} \times \frac{H}{h}$, and that of the global mesh is given by $2m\frac{H}{h} \times 2m\frac{H}{h} \times 2m\frac{H}{h}$.

4.2 A Simple Computational Model

Table 1 summarizes the well-known [16] order of arithmetic complexity of the different stages of the serial direct solution of sparse linear systems arising from the discretization of a square or cube with a uniform mesh with n nodes, with $d = 2, 3$ the dimension of the space.

If the uniform mesh is distributed over a uniform subdomain grid with $P = H^{-d}$ subdomains, with $n = (\frac{H}{h})^d$ nodes on each subdomain, the estimated complexity the NN-PCG and BDD-PCG methods is given on Table 2. Recall that, in our codes, Reordering, Symbolic Factorization and Numerical Factorization are performed during preconditioner and Schur complement set-up, while Triangular Solution is performed at each PCG iteration during Schur complement and preconditioner application. The complexity of reordering and symbolic factorization has been omitted in the table. Besides, for simplicity, the effect of communication and load unbalancing has been neglected. Constants c_s and c_i depend on the particular stencil the sparse direct method is applied to (linear FEs, quadratic FEs, etc.), and also on the ability of the software for the efficient exploitation of the underlying machine characteristics. For BDD-PCG methods, c_i actually depends on n, although this dependence is very mild (see e.g. (9)) and can be neglected as stated in Sects. 2.3 and 2.4.

The simple computational model in Table 2 reveals that the scalability of NN-PCG and BDD-PCG is composed of two components, a scalable one that does not depend on P,



(a) (b) (c) (d)

Fig. 4 Weak scalability for the parallel execution time of the coarse-grid preconditioner set-up ((**a**) and (**b**)) and application ((**c**) and (**d**)) for the 2D Poisson problem on the HPC-FF. (**a**) and (**c**): BDDC(c) preconditioner, one processor is responsible for coarse-grid problem duties, comparison of fixed and varying message size collectives. (**b**) and (**d**): comparison of BDDC(c), BDDC(ce) and BNN preconditioners with one or all processors responsible for coarse-grid problem duties, fixed message size collectives

and a non-scalable one that grows with P. In the former method, the non-scalable component is associated with the lack of a coarse-grid correction and subsequent degradation of PCG convergence rates with P (see Sect. 2.2). This is inherent to the preconditioning approach and cannot be mitigated. In the latter method, the non-scalable component is associated to the extra cost of the solution of the coarse-grid problem, and there is a lot of margin for improvement of this term via high-performance computing techniques (e.g., fine-grid/coarse-grid overlapping, distributed-memory implementation of coarse-grid preconditioning level). Anyway, *the key of BDD-PCG methods is that the non-scalable component does not depend on* $\frac{H}{h}$. This means that, as long as $\frac{H}{h}$ is "large enough", a balance among the non-scalable and scalable components can be reached such that the latter determines the overall (weak) scalability of the solution. The purpose of Sects. 4.3–4.4 is to demonstrate that, with the current software and distributed-memory machines, constants in Table 2 are such that our implementation can be very efficient for interesting ranges of applicability.

4.3 Coarse-Grid Preconditioning Weak Scalability

We first evaluate the weak scalability of the coarse-grid preconditioning level for the BNN and BDDC solvers. Besides, given the scenario depicted in Sect. 3.5, we will determine which is the fastest solution among the following three approaches: solution on one processor with fixed or varying message size collectives, and solution on all processors. Sections 4.3.1 and 4.3.2 cover the 2D and 3D Poisson problems, respectively.

4.3.1 2D Experiments

Figures 4(a) and (b) illustrate the weak scalability for the parallel execution time of the coarse-grid preconditioner set-up and Figs. 4(c) and (d) that of the coarse-grid preconditioner application for the BNN, BDDC(c), and BDDC(ce) solvers. The coarse-grid preconditioner set-up execution times includes both a symbolic phase, where the graph of S_0 is built and then symbolically factorized, and a numerical phase, where S_0 is assembled and then factorized by the sparse Cholesky factorization included in PARDISO. Highly parallel computations in this preconditioning level (such as the computation of $\Phi_k^t S_k \Phi_k$ or $\Phi_k^t r_k$) are excluded from the figure, i.e., only those terms that grow with P in Table 2, have been considered.

Figure 4 reveals that the weak scalability of the coarse-grid preconditioning level degrades as higher core counts are employed. This is caused by the combined effect of the global collectives scalability for increasing number of cores (see Fig. 3) and the *serial* preconditioner set-up and application. The latter factor results in idling or wasted computation parallel overheads, as the processors just waste their time waiting on a collective (if they do not have coarse-solver duties) or performing replicated computation, respectively. The coarse-grid preconditioning level is therefore a critical component in our current MPI implementation and any improvement can have significant impact on the performance and scalability of the overall solution (which will be later evaluated in Sect. 4.3.2). Figures 4(a) and (c) show one such improvement for the BDDC(c) preconditioner set-up and application, respectively, which comes from the use of fixed message size collectives in case one processor is responsible for coarse-grid level duties. This improvement can be justified by looking carefully at message sizes sent/received in the collectives. This in turn strongly depends on the particular algorithm and phase. For the assembling of the coarse-grid residual, each subdomain sends a message size proportional to the number of neighboring subdomains plus one in case of the BNN preconditioner, i.e., 9 elements × 8 bytes/element = 72 bytes, and to the number of local coarse-grid nodes in case of the BDDC preconditioner, i.e., $4 \times 8 = 32$ bytes and $8 \times 8 = 64$ bytes, for the BDDC(c) and BDDC(ce), respectively. These quantities are squared for the (numerical) assembling of S_0, i.e., $9^2 \times 8 = 648$, $4^2 \times 8 = 128$, and $8^2 \times 8 = 512$ bytes. As pointed out by the above discussion of Fig. 3, these message sizes are within the ranges where the use of fixed message size collectives can be beneficial over varying message size ones. Another significant improvement can be observed in Fig. 4(d) if one processor instead of all processors is responsible for coarse-grid preconditioning level duties. This can be justified by the superiority of the MPI_Gather + MPI_Scatter solution over the MPI_Allgather one for "small" message sizes (see Figs. 3(a), (b) and (c)). Indeed, "All processor" curves in Fig. 4(d) reflect the peaks that are observed for MPI_Allgather in Fig. 3(c). Focusing on the winner implementation for each phase and algorithm in Figs. 4(b) and (d), it can be observed that BDDC(ce) is the method with the most expensive coarse-grid preconditioner set-up and application and besides its computational time degrades with the number of cores at the highest rate, followed by the BNN and BDDC(c) solvers. This ranking is not surprising if one takes a closer look at the stencil of the coarse-grid coefficient matrix of each method for structured partitions. BDDC(c) presents the stencil corresponding to the Q1 FE discretization, BNN a more intricate one where neighbors of neighbors in the Q1 FE discretization are also connected, and finally that of BDDC(ce) resembles that of the Q2 FE discretization (after static condensation of interior nodes). The complexity of the sparse direct Cholesky method applied to a uniform grid with $n = P$ grid points is given in Table 1, with the particular stencil only affecting to the constant. Therefore, it is reasonable that the more intricate the stencil the higher the constant, confirming what is observed in Figs. 4(b) and (d).

4.3.2 3D Experiments

Figures 5(a) and (b) illustrate the weak scalability for the parallel execution time of the coarse-grid preconditioner set-up and Figs. 5(c) and (d) that of the coarse-grid preconditioner application for the BNN, BDDC(ce), and BDDC(cef) solvers.

The solution based on fixed message size collectives is superior to the one based on varying message size ones (see Figs. 5(a) and (c)) and the one processor dedicated to coarse-grid problem duties solution is also superior to the all processors one (see Figs. 5(b) and (d)); the justification of these results follows the one for the 2D structured case (although with larger message sizes in 3D). A much more interesting observation is the relative ranking of the BNN, BDDC(ce) and BDDC(cef) coarse-grid preconditioners and the rate at which their weak scalability degrades with the number of cores. As illustrated by Figs. 5(b) and (d), BNN turns to be the method with the cheapest coarse-grid preconditioner set-up and application and the one with the smallest rate, followed by the BDDC(ce) and BDDC(cef) in this strict order. Table 3 provides several metrics of the coarse-grid problem that helps to understand this observation, namely the size and number of non-zeros in its sparse coefficient matrix, and the size of the optimal root separator of its adjacency graph. Although the BNN coarse-grid sparse coefficient is denser, it is 4 and 7 times smaller than that of the BDDC(ce) and BDDC(cef), respectively, and its optimal root separator is 1.5 and 2 times smaller than that of the BDDC(ce) and BDDC(cef), respectively. The size of the optimal root separator, which can be used as lower bound for the complexity of the sparse direct Cholesky (actually its cube and square for the factorization and forward/backward substitution, respectively), accurately describes what is observed in Figs. 5(b) and (d).

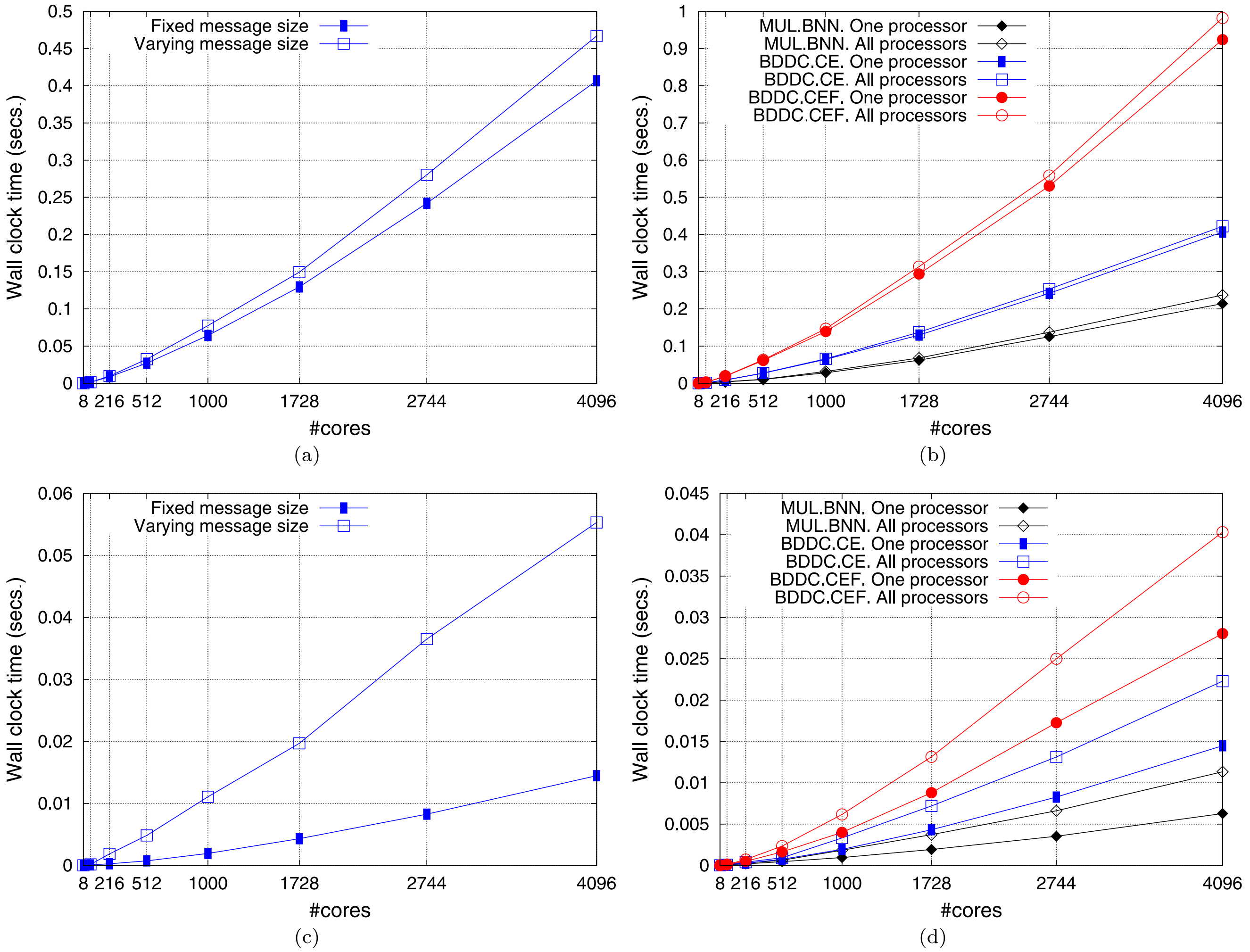


Fig. 5 Weak scalability for the parallel execution time of the coarse-grid preconditioner set-up ((**a**) and (**b**)) and application ((**c**) and (**d**)) for the 3D Poisson problem on the HPC-FF. (**a**) and (**c**): BDDC(ce) preconditioner, one processor is responsible for coarse-grid problem duties, comparison of fixed and varying message size collectives. (**b**) and (**d**): comparison of BDDC(ce), BDDC(cef) and BNN preconditioners with one or all processors responsible for coarse-grid problem duties, fixed message size collectives

Table 3 Size (n_c), non-zeros (n_z) and optimal root separator size (n_s) for the coarse-grid coefficient matrix in the BDDC and BNN algorithms. A periodic structured mesh of a cube with $P = p^3$ subdomains is assumed, with p the number of subdomains per Cartesian direction

Metric	BDDC (ce)	BDDC (cef)	BNN
n_c	$4P$	$7P$	P
n_z	$234P$	$462P$	$125P$
n_s	$3P^{2/3}$	$4P^{2/3}$	$2P^{2/3}$

4.4 Overall Scalability

In this section we take into consideration the overall scalability of the BNN and BDDC solvers. Both fine-grid and coarse-grid preconditioning contributions to the scaling curves are considered, as well as the number of PCG iterations required to converge. Sections 4.4.1 and 4.4.2 cover the 2D and 3D Poisson problems, respectively.

4.4.1 2D Experiments

Figure 6 reports the weak scalability for the total computation time of the winner implementation of the multiplicative BNN solver and two different implementations of the BDDC(c) and BDDC(ce) solvers. The best implementation of the coarse-grid preconditioning level was used (see Sect. 4.3). In the legend of the figure, `DEF` (symmetric-PD) and `IND` (symmetric INDefinite) refer to the kind of linear systems/solvers that are solved/applied for the computation of the BDDC fine-grid correction (see Sect. 3.2.2). The winner implementation of the BNN method exploits symmetric-PD solvers and saves the solution of a Dirichlet solver per

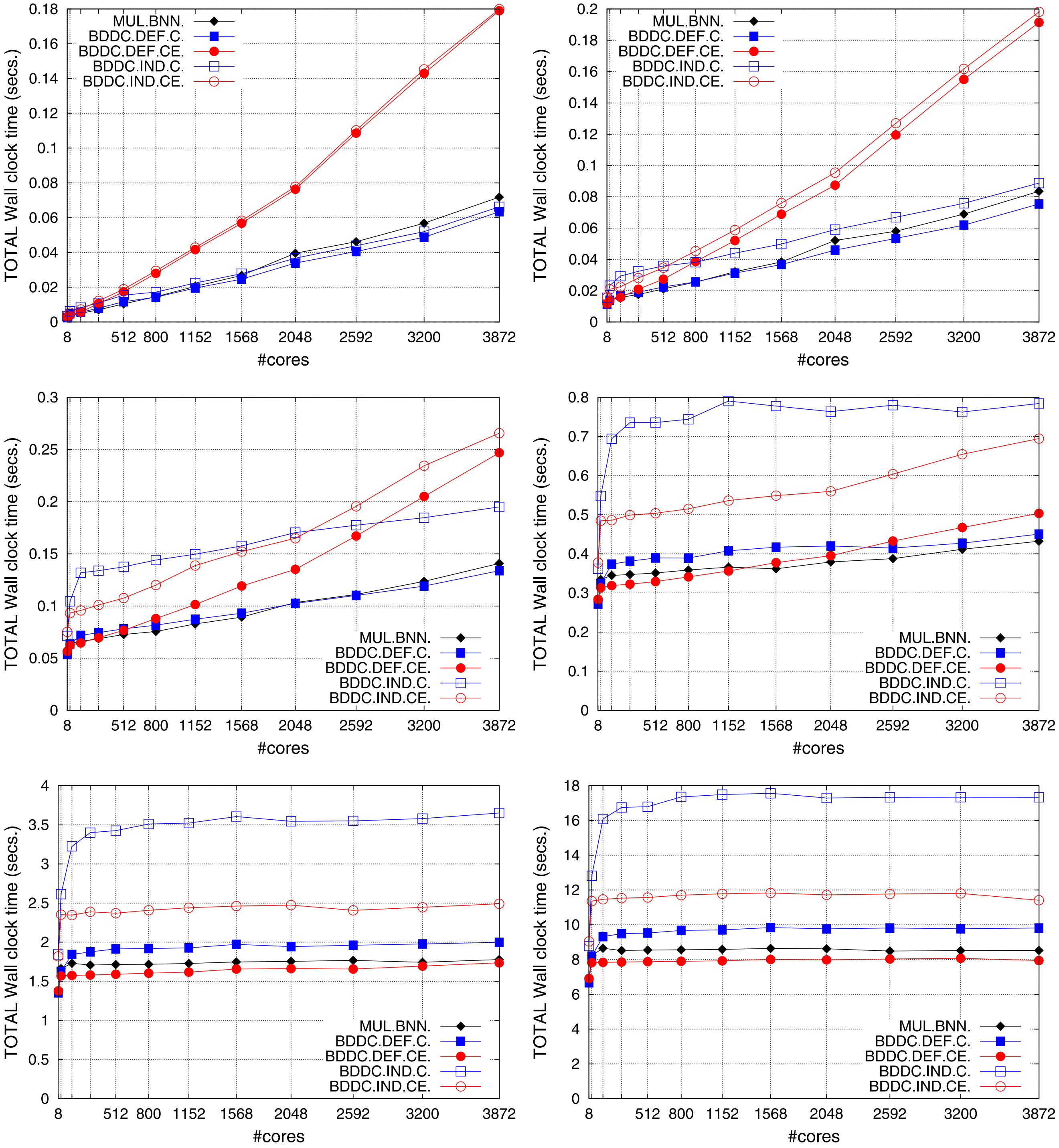


Fig. 6 Weak scalability for the total computation time of the multiplicative BNN (winner implementation) and two different implementations of the BDDC(c) and BDDC(ce) solvers for the 2D Poisson problem on HPC-FF. From *top* to *bottom* and *left* to *right* $\frac{H}{h} = 16, 32, 64, 128, 256, 512$

PCG iteration (cf. [2]). Figure 7 illustrates the weak scalability for the number of PCG iterations. In the PCG method, we set the initial solution vector guess $x_0 = 0$, and the iteration is stopped whenever the residual r_k at a given iteration k satisfies $\|r_k\|_2 \leq 10^{-6}\|r_0\|_2$; this set-up also applies to Sect. 4.4.2.

As predicted by our simple computational model in Sect. 4.2, Fig. 6 clearly evidences that weak scaling curves for the computational time of BNN/BDDC solvers result from the sum of a non-scalable and a scalable component. For "sufficiently small" $\frac{H}{h}$ (e.g., with $\frac{H}{h} = 16, 32, 64$), the non-scalable component (i.e., the one that grows with P)

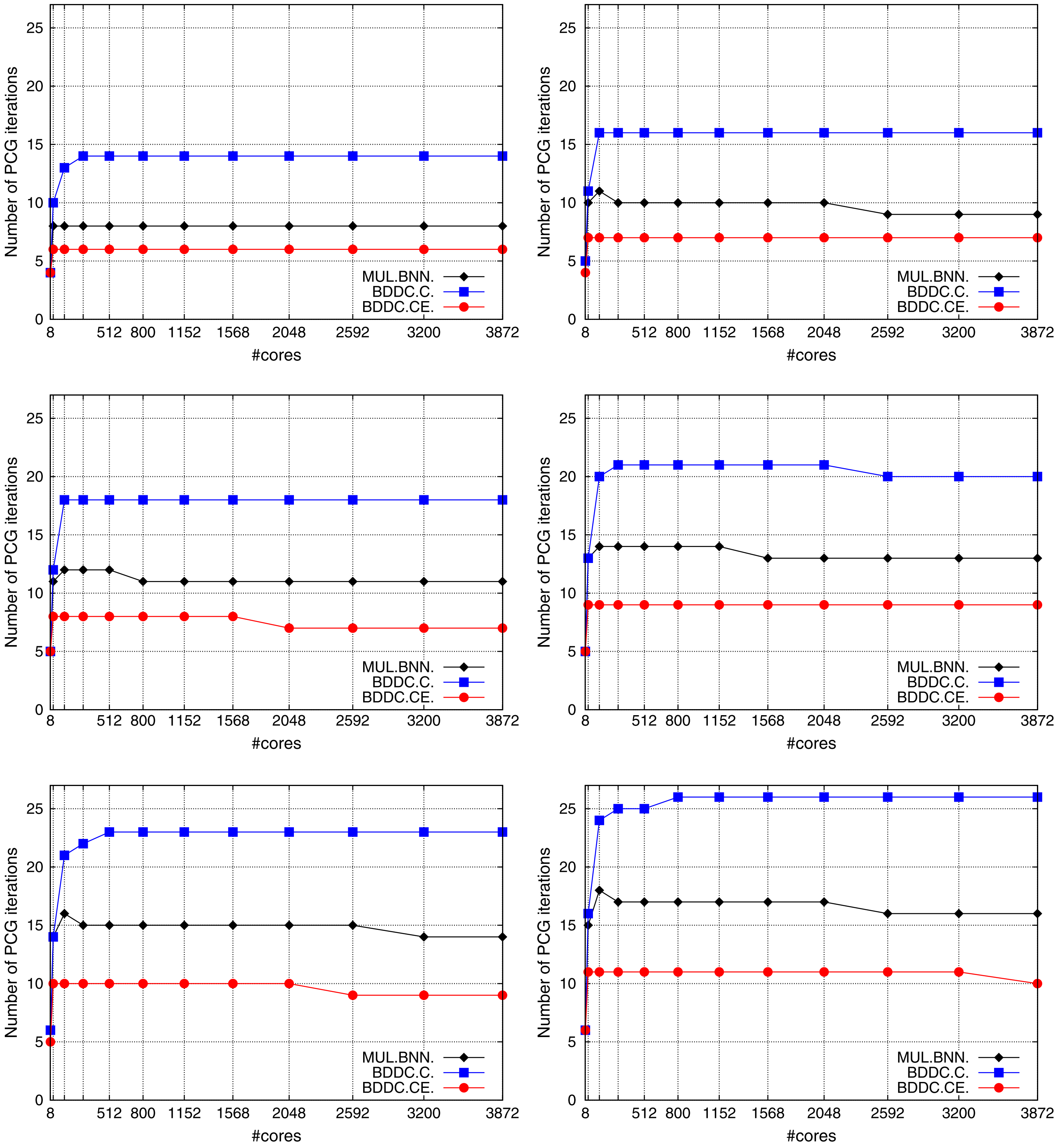
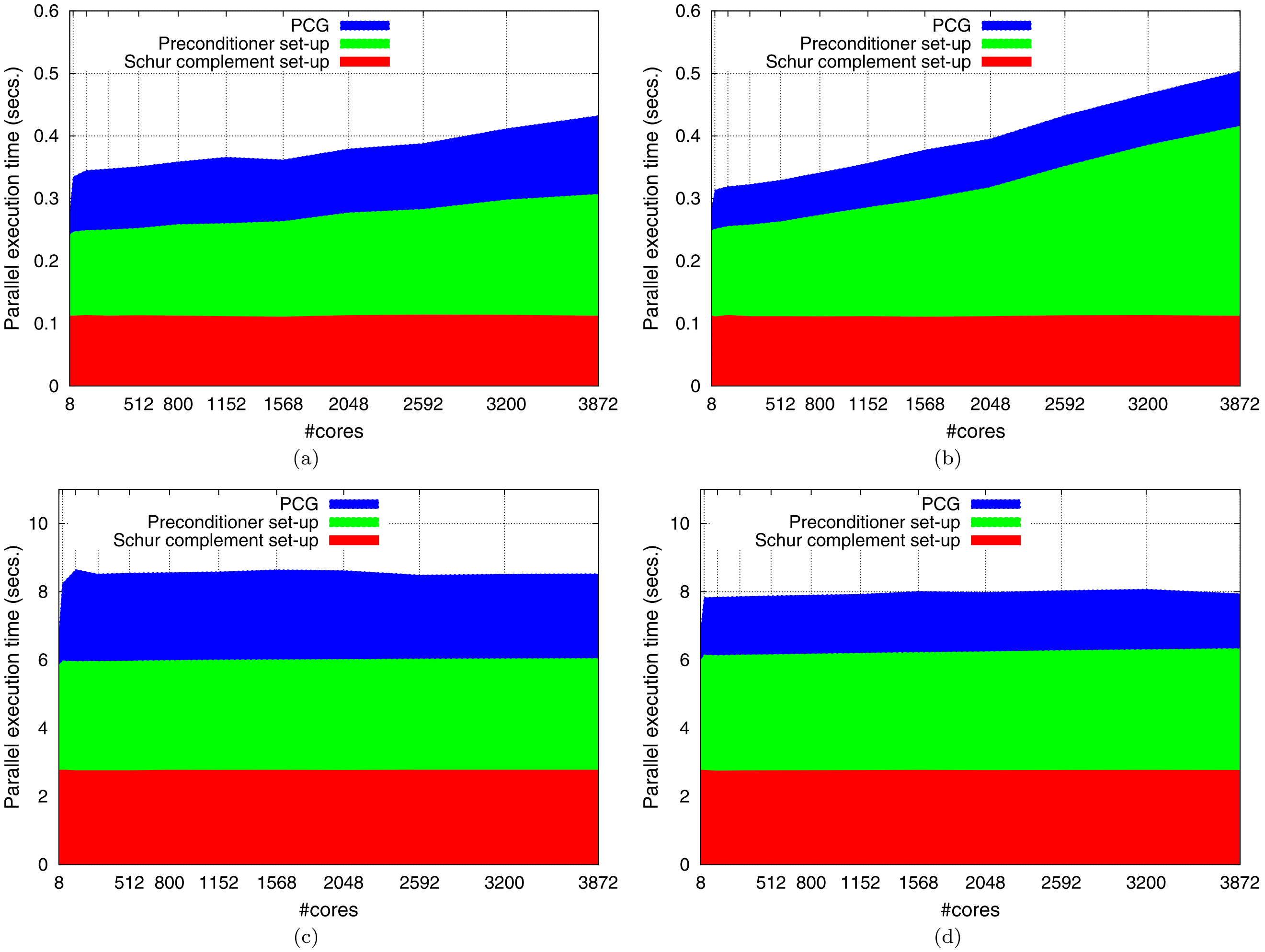


Fig. 7 Weak scalability for the number of PCG iterations of the multiplicative BNN and the BDDC(c) and BDDC(ce) solvers for the 2D Poisson problem on HPC-FF. From *top* to *bottom* and *left* to *right* $\frac{H}{h} = 16, 32, 64, 128, 256, 512$

dominates. The relative ranking of the methods is therefore determined by the extra cost required for the solution of the coarse-grid problem of each method, which has already been examined and properly justified in Sect. 4.3.1. However, a very nice observation is that for gradually larger $\frac{H}{h}$, the scalable part becomes more and more dominant, till the non-scalable component is completely masked, i.e., with $\frac{H}{h} = 256, 512$. Within this range, the winner method is the one with the least asymptotic number of PCG iterations, i.e., the BDDC(ce) solver, as illustrated in Fig. 7; a significant maximum improvement of 43 % and 30 % which comes from the use of (symmetric-PD) solvers can also be



(a)

(b)

(c)

(d)

Fig. 8 Distribution of the total computation time among Schur complement and preconditioner set-up, and the iterative solution of the interface problem for 2D Poisson on HPC-FF. Multiplicative BNN and BDDC(ce) with $\frac{H}{h} = 128$ ((**a**) and (**b**), respectively) and $\frac{H}{h} = 512$ ((**c**) and (**d**), respectively)

observed within this range for the BDDC(c) and BDDC(ce) solvers, respectively. The number of cores can certainly be increased arbitrarily so that the coarse-grid component becomes dominant. However, a very remarkable conclusion from this study is that given the memory available per core on current distributed-memory machines, and the experimental evidence we have gathered so far, *this is only expected to happen for simulations with several tens of thousands of cores*. Besides, the margin of improvement for the coarse-grid preconditioning level is huge. In Fig. 7 it can also be observed no degradation of PCG convergence rates with P and fixed $\frac{H}{h}$, and only a mild (i.e., logarithmic) grow with $\frac{H}{h}$ and P fixed. This is a well-known property of BNN/BDDC solvers easily derived from the condition number bounds of the preconditioned operator (see Sects. 2.3 and 2.4).

Although not explicitly provided in Figs. 6 and 7, the NN-PCG solver required, for $\frac{H}{h} = 16$, and $P = 228$ and $P = 3872$, 0.18 and 1.73 seconds, with 156 and 2207 PCG iterations, respectively. These numbers are significantly worse than those of the BNN/BDDC solvers. Therefore, even in the most favourable scenario for the NN-PCG solver in the experiment (i.e., the smallest local problem size considered), the extra cost associated with the coarse-grid correction more than pays off in terms of total computational time (due to the significant cut down in the number of iterations). For larger values of $\frac{H}{h} = 16$, the gap among NN-PCG and the BDDC/BNN solvers becomes progressively larger, as predicted by Table 2 (as the term that depends on $\sqrt{P}$ is multiplied by a function that grows with $\frac{H}{h}$).

Figure 8 offers a complementary view of the weak scaling curves in Fig. 6. For $\frac{H}{h} = 128$ (see Figs. 6(a) and (b)), the computation time of the preconditioner set-up and iterative phases grow with P (the latter at a very moderate pace compared to the former as predicted by Sect. 4.2 and experimentally examined in Sect. 4.3.1). However, for $\frac{H}{h} = 512$ (see

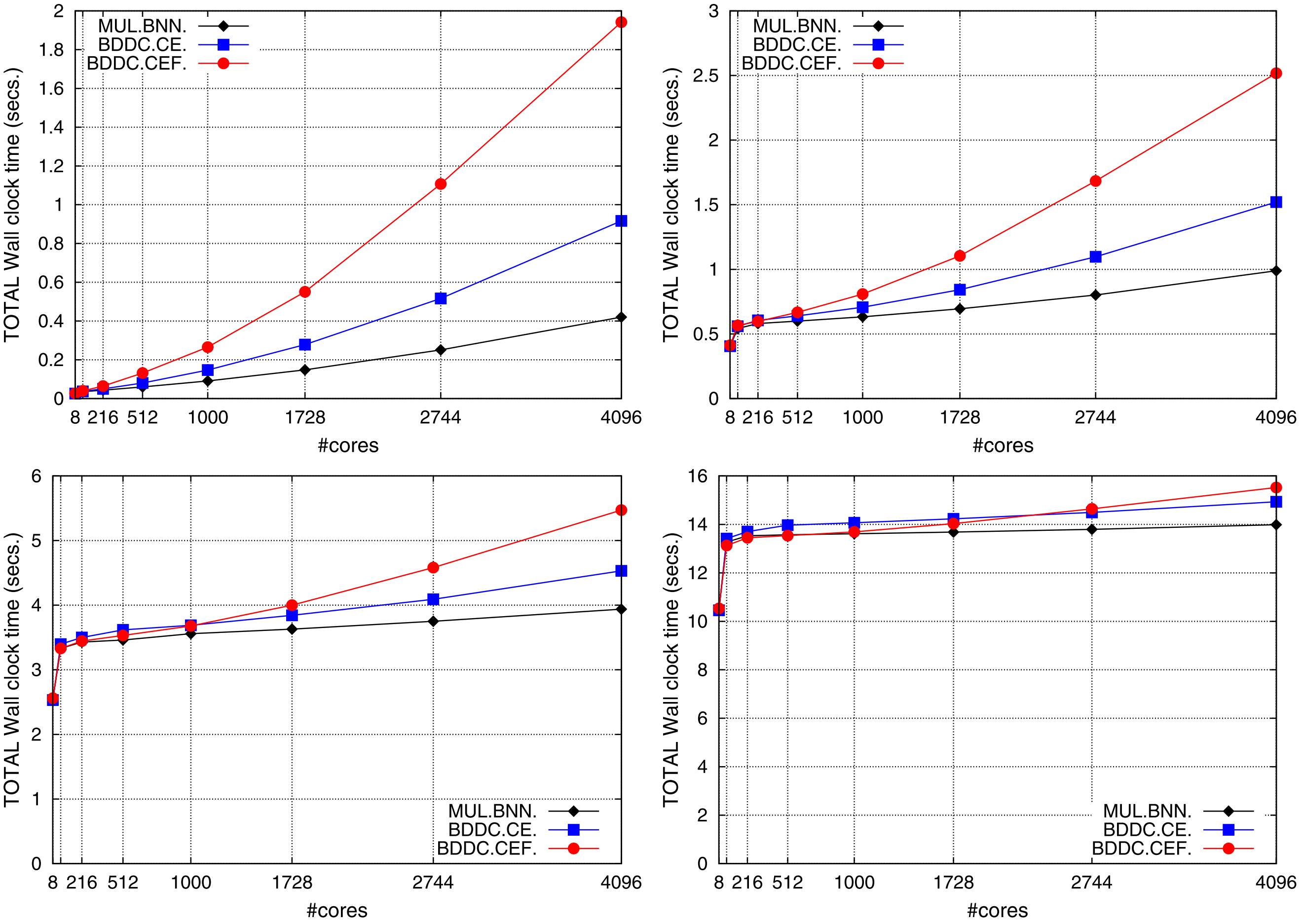


Fig. 9 Weak scalability for the total computation time of the multiplicative BNN (winner implementation) and the BDDC(ce) and BDDC(cef) solvers (winner implementation) for the 3D Poisson problem on HPC-FF. From *top* to *bottom* and *left* to *right* $\frac{H}{h} = 10, 20, 30, 40$

Figs. 6(c) and (d)), the computation time of the three phases is constant. An interesting observation for this "large" local problem size is that the computation time for preconditioner set-up is equivalent for the BNN and BDDC(ce) solvers. In the latter method, an extra number, proportional to the number of edge constraints, sparse forward/backward substitutions with the factor of A^i_{RR} are required to build the Schur complement associated to edge constraints (see Sect. 3.2.2). However, this is only a constant and modest multiple, completely masked by the higher order of complexity of the sparse Cholesky factorization of A^i_{RR}. We stress, however, that this is no longer true if approximate solvers (e.g., AMG) are used as local solvers (due to their linear order of complexity).

4.4.2 3D Experiments

Figures 9 and 10 compare the weak scalability for the total computation time and number of PCG iterations of the winner implementation of the multiplicative BNN solver and those of the winner implementation of the BDDC(ce) and BDDC(cef) solvers. Figure 9 again reveals the two components of the weak scaling curves. For sufficiently "small" $\frac{H}{h}$ (e.g., $\frac{H}{h} = 10, 20$), the total computational time is dominated by computation and communication overheads related to the solution of the coarse-grid system. To be more precise, it is dominated by the sparse Cholesky factorization of the coarse-grid coefficient matrix: the shape of the curves in Fig. 9 resembles (particularly with large P) that of the curves in Fig. 5(b). For a very precise explanation of the relative ranking of the BNN/BDDC solvers for "small" $\frac{H}{h}$, we refer the reader to Sect. 4.3.2. For large $\frac{H}{h}$, the fine-grid preconditioning component of the solvers dominates and the efficiency of the methods is very high due to their ability to keep the condition number bounded by a constant with P and $\frac{H}{h}$ fixed (see Fig. 10). It is remarkable the nice scalability of the BNN compared to that of the BDDC solver in terms of computational time (see Fig. 9). For this latter method, a (mild) degradation of the weak scalability can already be observed for large P even for the largest $\frac{H}{h}$.

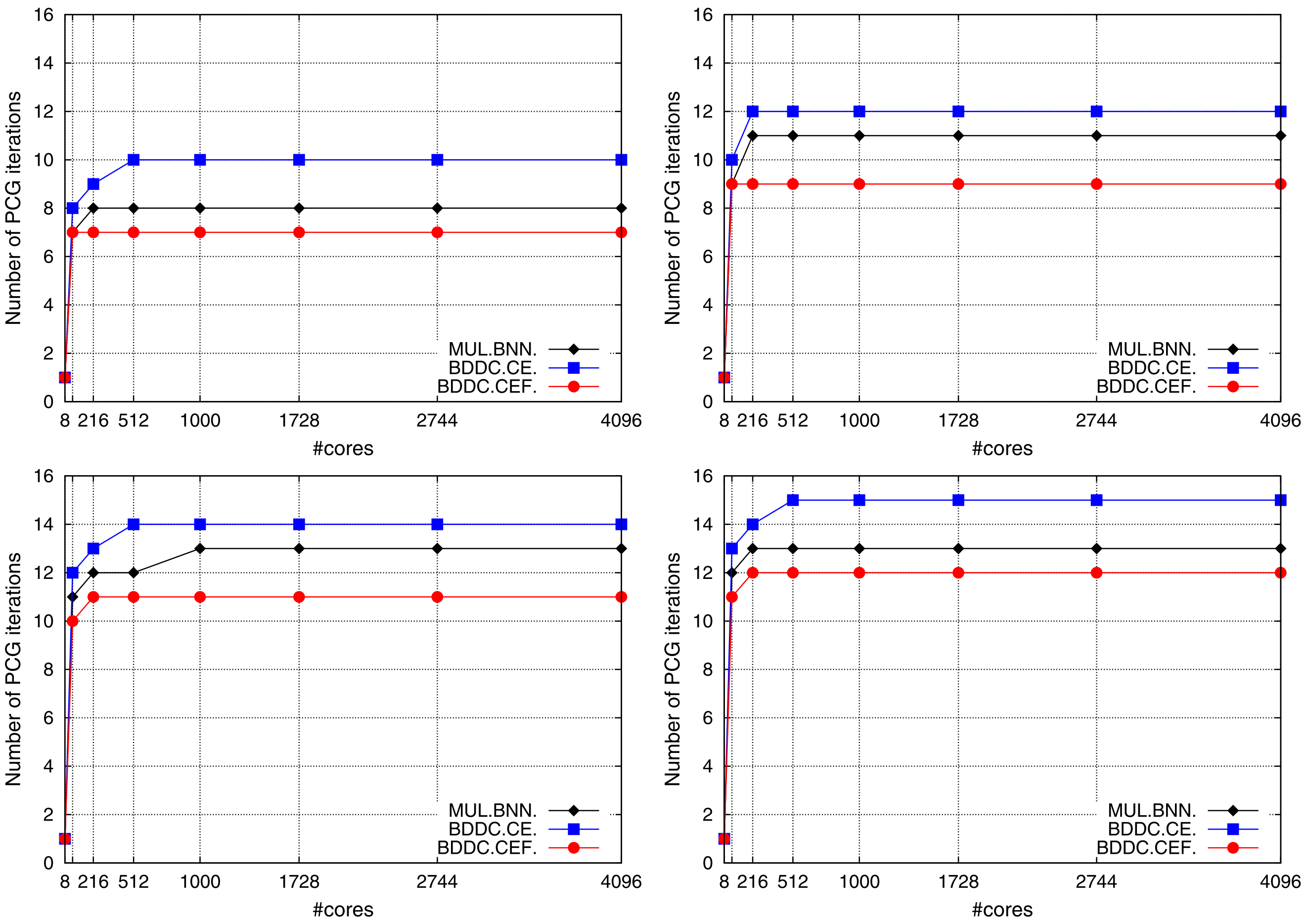


Fig. 10 Weak scalability for the number of PCG iterations of the multiplicative BNN and the BDDC(ce) and BDDC(cef) solvers for the 3D Poisson problem on HPC-FF. From *top* to *bottom* and *left* to *right* $\frac{H}{h} = 10, 20, 30, 40$

5 Conclusions and Future Work

In this work we have covered in detail the high-performance distributed-memory implementation of DD methods of balancing type. This comprehensive coverage presents a pool of hints and considerations that can be very useful for scientists that are willing to tackle large-scale distributed-memory machines using these methods. On the other hand, the paper presents a complete scalability study of BDDC/BNN preconditioners on a large-scale machine with up to 4096 cores. As far as we know, the state-of-the-art does not include any work that performs such study with these particular methods, at least with the degree of detail and up to the scale that are reached in this work. This scalability study answers the very interesting question of how far can the proposed MPI implementation go in the number of cores and the scale of the problem to still be within reasonable ranges of efficiency. The answer is up to dozens of thousands of computational cores in the solution of problems discretized with hundreds of millions FEs. Besides, the study has also precisely identified, quantified and justified which are the main sources of inefficiency and bottlenecks in our current implementation, namely communication and computation associated to the solution of the coarse-grid problem. In light of these conclusions, we have identified improvements that deserve further research in order to boost the current scalability of our MPI implementation, e.g., the use of approximate solvers (as AMG-preconditioned CG), multilevel BDD formulations (see [30]) or distributed memory coarse-grid solvers.

Acknowledgements This work has been funded by the European Research Council under the FP7 Programme Ideas through the Starting Grant No. 258443—COMFUS: Computational Methods for Fusion Technology and the FUSSIM project (Ref. ENE2011-285556), financed by the Spanish Ministry of Science and Innovation (MICINN), under the programme "Proyectos de Investigación fundamental no orientada". A.F. Martín was also partially funded by the UPC postdoctoral grants under the programme "BKC-5-Atracció i Fidelització de talent al BKC". The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Red Española de Supercomputación and the Juelich Supercomputing Centre in the exploitation of the HPC for Fusion (HPC-FF) under the EFDA HPC Implementing Agreement (EFDA (08) 39/4.1).

References

1. Badia S, Codina R (2008) Algebraic pressure segregation methods for the incompressible Navier-Stokes equations. Arch Comput Methods Eng 15:343–369
2. Badia S, Martín AF, Príncipe J (2012) Enhanced balancing Neumann-Neumann preconditioning in computational fluid and solid mechanics. Int J Numer Methods Eng (in press)
3. Baker AH, Gamblin T, Schulz M, Yang UM (2011) Challenges of scaling algebraic multigrid across modern multicore architectures In: Parallel distributed processing symposium (IPDPS). 2011 IEEE international, pp 275–286
4. Balay S, Brown J, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2012) PETSc users manual. Technical report ANL-95/11, Argonne National Laboratory
5. Balay S, Brown J, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2012). PETSc web page. http://www.mcs.anl.gov/petsc
6. Brenner SC, Scott R (2010) The mathematical theory of finite element methods, 3rd edn. Springer, Berlin
7. Conceição D, Goldfeld P, Sarkis M (2007) Robust two-level lower-order preconditioners for a higher-order stokes discretization with highly discontinuous viscosities. In: High performance computing for computational science—VECPAR 2006, pp 319–333
8. Davis TA (2006) Direct methods for sparse linear systems, vol 2. SIAM, Philadelphia
9. De Roeck YH, Le Tallec P (1991) Analysis and test of a local domain decomposition preconditioner. In: Fourth international symposium on domain decomposition methods for partial differential equations, 112 pp
10. Dohrmann CR (2003) A preconditioner for substructuring based on constrained energy minimization. SIAM J Sci Comput 25(1):246–258
11. Farhat C, Roux F-X (1991) A method of finite element tearing and interconnecting and its parallel solution algorithm. Int J Numer Methods Eng 32(6):1205–1227
12. Farhat C, Pierson K, Lesoinne M (2000) The second generation FETI methods and their application to the parallel solution of large-scale linear and geometrically non-linear structural analysis problems. Comput Methods Appl Mech Eng 184(2–4):333–374
13. Ferencz RM, Hughes TJR (1998) Iterative finite element solutions in nonlinear solid mechanics. In: Numerical methods for solids (part 3). Handbook of numerical analysis, vol VI
14. Filippone S, Buttari A (2012) Object-oriented techniques for sparse matrix computations in Fortran 2003. ACM Trans Math Softw 38(4):23:1–23:20
15. Filippone S, Colajanni M (2000) PSBLAS: a library for parallel linear algebra computation on sparse matrices. ACM Trans Math Softw 26(4):527–550
16. George A (1973) Nested dissection of a regular finite element mesh. SIAM J Numer Anal 10(2):345–363
17. Giraud L, Haidar A, Watson LT (2008) Parallel scalability study of hybrid preconditioners in three dimensions. Parallel Comput 34(6–8):363–379
18. Glowinski R, Wheeler MF (1988) Domain decomposition and mixed finite element methods for elliptic problems. In: First international symposium on domain decomposition methods for partial differential equations, pp 144–172
19. Goldfeld P (2003) Balancing Neumann-Neumann preconditioners for the mixed formulation of almost-incompressible linear elasticity. PhD thesis
20. Goldfeld P, Pavarino LF, Widlund OB (2003) Balancing Neumann-Neumann preconditioners for mixed approximations of heterogeneous problems in linear elasticity. Numer Math 95:283–324
21. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message passing interface, vol 1. MIT Press, Cambridge
22. Gropp W, Lusk E, Thakur R (1999) Using MPI-2: advanced features of the message passing interface. MIT Press, Cambridge
23. Heroux MA, Willenbring JM (2003) Trilinos users guide. Technical report SAND2003-2952, Sandia National Laboratories
24. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, Williams A, Stanley KS (2005) An overview of the Trilinos project. ACM Trans Math Softw 31(3):397–423
25. Hoefler T, Traff JL (2009) Sparse collective operations for MPI. In: Proceedings of the 2009 IEEE international symposium on parallel & distributed processing, pp 1–8
26. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
27. Lin PT, Shadid JN, Sala M, Tuminaro RS, Hennigan GL, Hoekstra RJ (2009) Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling. J Comput Phys 228(17):6250–6267
28. Mandel J (1993) Balancing domain decomposition. Commun Numer Methods Eng 9(3):233–241
29. Mandel J, Dohrmann CR (2003) Convergence of a balancing domain decomposition by constraints and energy minimization. Numer Linear Algebra Appl 10(7):639–659
30. Mandel J, Sousedík B, Dohrmann C (2008) Multispace and multilevel BDDC. Computing 83:55–85
31. Saad Y (1995) Data structures and algorithms for domain decomposition and distributed sparse matrix computations. Technical report 95-014, Department of Computer Science, University of Minnesota, Minneapolis, MN
32. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia
33. Saad Y, Sosonkina M (1999) Distributed Schur complement techniques for general sparse linear systems. SIAM J Sci Comput 21(4):1337–1356
34. Sahni O, Carothers CD, Shephard MS, Jansen KE (2009) Strong scaling analysis of a parallel, unstructured, implicit solver and the influence of the operating system interference. Sci Program 17(3):261–274
35. Sala M, Tuminaro R (2008) A new Petrov-Galerkin smoothed aggregation preconditioner for nonsymmetric linear systems. SIAM J Sci Comput 31(1):143–166
36. Schenk O, Gärtner K (2004) Solving unsymmetric sparse systems of linear equations with PARDISO. Future Gener Comput Syst 20(3):475–487
37. Schenk O, Gärtner K (2006) On fast factorization pivoting methods for sparse symmetric indefinite systems. Electron Trans Numer Anal 23:158–179
38. Šístek J, Sousedík B, Burda P, Mandel J, Novotný J (2011) Application of the parallel BDDC preconditioner to the Stokes flow. Comput Fluids 46(1):429–435
39. Strang G (2006) Linear algebra and its applications. Thomson Brooks/Cole, Belmont
40. Stüben K (2001) A review of algebraic multigrid. J Comput Appl Math 128(1–2):281–309
41. Tang J, Nabben R, Vuik C, Erlangga Y (2009) Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. J Sci Comput 39(3):340–370
42. Toselli A, Widlund O (2005) Domain decomposition methods—algorithms and theory. Springer, Berlin
43. Vaněk P, Mandel J, Brezina M (1996) Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56:179–196