Parallel Computing 36 (2010) 285–296

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Using multiple levels of parallelism to enhance the performance of domain decomposition solvers

L. Giraud [a,1], A. Haidar [a,*], S. Pralet [b,2]

[a] University of Toulouse, INPT-ENSEEIHT, France
[b] SAMTECH, Belgium

ARTICLE INFO

Article history:
Received 21 November 2008
Received in revised form 3 July 2009
Accepted 9 December 2009
Available online 11 January 2010

Keywords:
Hybrid iterative/direct linear solver
Domain decomposition
Multilevel of parallel implementation
Large scale linear systems
High performance computing

ABSTRACT

Large-scale scientific simulations are nowadays fully integrated in many scientific and industrial applications. Many of these simulations rely on modelisations based on PDEs that lead to the solution of huge linear or nonlinear systems of equations involving millions of unknowns. In that context, the use of large high performance computers in conjunction with advanced fully parallel and scalable numerical techniques is mandatory to efficiently tackle these problems.

In this paper, we consider a parallel linear solver based on a domain decomposition approach. Its implementation naturally exploits two levels of parallelism, that offers the flexibility to combine the numerical and the parallel implementation scalabilities. The combination of the two levels of parallelism enables an optimal usage of the computing resource while preserving attractive numerical performance. Consequently, such a numerical technique appears as a promising candidate for intensive simulations on massively parallel platforms.

The robustness and parallel numerical performance of the solver is investigated on large challenging linear systems arising from the finite element discretization in structural mechanics applications.

1. Introduction

Massively parallel computers promise unique power for large engineering and scientific simulations. Large parallel machines are likely to be the most widely used computers in the future. Their efficient exploitation requires a deep rethinking of some parallel methods and algorithmic designs. It is a challenge to develop and implement efficient parallel numerical techniques that fully exploit the computing power. In this paper, we focus on the solution of large sparse linear systems of equations. This "basic" numerical kernel consumes a significant part of the computing time in many intensive simulations. We focus on a hybrid solution technique that combines direct and iterative schemes. This approach borrows ideas to domain decomposition approaches and consequently can be easily described in a PDE framework while its extension to general sparse linear systems is straightforward. In that latter situation, the underlying ideas are applied to the graph of the sparse matrix and no longer to the graph of the mesh as described here.

* Corresponding author. Current address: ICL-University of Tennessee, USA.
E-mail addresses: luc.giraud@inria.fr (L. Giraud), haidar@cerfacs.fr (A. Haidar), stephane.pralet@samcef.com, stephane.pralet@bull.net (S. Pralet).
[1] Current address: INRIA-Bordeaux Sud-Ouest, France.
[2] Current address: Bull SAS, 1, rue de Provence, B.P. 208, 38432 Echirolles Cedex, France.

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.12.006

For the solution of PDEs problems on large complex 3*D* geometries, the domain decomposition techniques are natural approaches to split the problem into subproblems in order to express some parallelism. The subproblems are allocated to the different processors in a parallel algorithm [20,21]. In the classical parallel implementations of those numerical techniques, each subproblem is allocated to one processor. Numerically, for some classes of problems such as elliptic equations, some preconditioners for Krylov methods possess optimal convergence properties that are independent from the number of subdomains. This numerical scalability is often achieved thanks to the use of two-level preconditioners that are composed by local and global terms as first introduced in [5]. The global term solves a coarse problem of small dimension where the number of unknowns is generally only proportional to the number of subdomains. For problems where such a two-level scheme does not exist, a deterioration of the convergence rate is often observed when the number of subdomains is increased. Consequently, when the number of processors is increased part of the computing resource is "lost" because extra iterations are required to converge.

In order to alleviate this weakness and attempt to fully benefit from all the computer resource involved in the simulations, we consider *2-level parallel* implementations that enables us to express parallelism between the subproblems but also within the treatment of each subproblem. In this paper we illustrate the benefit of such implementation in the framework of a parallel hybrid iterative-direct numerical technique. This approach is based on an algebraic preconditioner [7,14] for the Schur complement system that classically appears in non-overlapping domain decomposition method.

The paper is organized as follows. In Section 2 we briefly describe the basic algebraic ideas that underline the non-overlapping domain decomposition method that we use to design our hybrid linear solver. In that section the basic features of the algebraic additive Schwarz preconditioner for the Schur complement and the parallel preconditioner are presented. The classical parallel domain decomposition implementation (refer to as *1-level parallel*) is first introduced in Section 3 and the implementation that exploits two levels of parallelism is detailed. For the sake of efficiency and portability our parallel implementation is developed on top of efficient available parallel numerical linear algebra libraries such as ScaLAPACK, PBLAS, BLAS [3] and MUMPS [1] that use MPI [15]. For the Krylov subspace solvers, we consider packages suited for parallel distributed computing [11–13]. We notice that the same parallel design can be developed with mixed multi-threading and message passing paradigms. It would rely on other choices of parallel libraries. We do not further investigate this possibility in this paper but mention some ongoing work in that direction in the concluding remarks section.

2. A brief overview of non-overlapping domain decomposition

In this section, methods based on non-overlapping regions are described. Such domain decomposition algorithms are often referred to as substructuring schemes. This terminology comes from the structural mechanics discipline where non-overlapping ideas were first developed.

Let us now further describe this technique and let $\mathcal{A}x = b$ be the linear problem arising from the discretization where the matrix $\mathcal{A}$ is referred to as the stiffness matrix. We assume that the domain Ω is partitioned into N non-overlapping subdomains $\Omega_1,\ldots,\Omega_N$ with boundaries $\partial\Omega_1,\ldots,\partial\Omega_N$. The governing idea behind substructuring or Schur complement methods is to split the unknowns in two subsets. This induces the following block reordered linear system:

$$\begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ \mathcal{A}_{\Gamma I} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ b_\Gamma \end{pmatrix}, \tag{1}$$

where x_Γ contains all unknowns associated with subdomain interfaces and x_I contains the remaining unknowns associated with subdomain interiors. Because the interior points are only connected to either interior points in the same subdomain or with points on the boundary of the subdomains, the matrix $\mathcal{A}_{II}$ has a block diagonal structure, where each diagonal block corresponds to one subdomain. Eliminating x_I from the second block row of Eq. (1) leads to the reduced system

$$\mathcal{S}x_\Gamma = f, \tag{2}$$

where

$$\mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}\mathcal{A}_{I\Gamma} \quad \text{and} \quad f = b_\Gamma - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}b_I. \tag{3}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix*. This reformulation leads to a general strategy for solving (1). Specifically, an iterative method can be applied to (2). Once x_Γ is known, x_I can be computed with one additional solve on the subdomain interiors.

Not surprisingly, the structural analysis finite element community has been heavily involved in the designs and developments of these techniques. Not only is their definition fairly natural in a finite element framework but their implementation can preserve data structures and concepts already present in large engineering software packages.

Let Γ denote the entire interface defined by $\Gamma = \cup \Gamma_i$ where $\Gamma_i = \partial\Omega_i \backslash \partial\Omega$; we notice that if two subdmains Ω_i and Ω_j share an interface then $\Gamma_i \bigcap \Gamma_j \neq \emptyset$. As interior unknowns are no longer considered, new restriction operators must be defined as follows. Let $\mathcal{R}_{\Gamma_i} : \Gamma \rightarrow \Gamma_i$ be the canonical point-wise restriction which maps full vectors defined on Γ into vectors defined on Γ_i. Thus, in the case of many subdomains, the fully assembled global Schur $\mathcal{S}$ is obtained by summing the contributions over the substructures/subdomains. The global Schur complement matrix (3) can be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}, \tag{4}$$

where

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{5}$$

is a local Schur complement associated with Ω_i. It can be defined in terms of sub-matrices from the local Neumann matrix $\mathcal{A}_i$ defined by

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ A_{\Gamma_i\mathcal{I}_i} & A_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{6}$$

While the Schur complement system is significantly better conditioned than the original matrix $\mathcal{A}$, it is important to consider further preconditioning when employing a Krylov method. It is well-known, for example, that $\kappa(A) = \mathcal{O}(h^{-2})$ when $\mathcal{A}$ corresponds to a standard discretization (e.g. piecewise linear finite elements) of the Laplace operator on a mesh with spacing h between the grid points. Using two non-overlapping subdomains effectively reduces the condition number of the Schur complement matrix to $\kappa(S) = \mathcal{O}(h^{-1})$. While improved, preconditioning can significantly lower this condition number further.

2.1. The algebraic additive Schwarz preconditioner

In this section we introduce the general form of the preconditioner considered in this work. The preconditioner presented below was originally proposed in [7] in two dimensions and successfully applied to large three dimensional problems and real life applications in [14,16]. To describe this preconditioner we define the local assembled Schur complement, $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} S \mathcal{R}_{\Gamma_i}^T$, that corresponds to the restriction of the Schur complement to the interface Γ_i. This local assembled preconditioner can be built from the local Schur complements $\mathcal{S}_i$ by assembling their diagonal blocks.

With these notations the preconditioner reads

$$M_d = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{7}$$

If we considered the unit square partitioned into horizontal strips (1D decomposition), the resulting Schur complement matrix has a block tridiagonal structure as depicted in (8)

$$\mathscr{S} = \begin{pmatrix} \ddots & & & \\ & \boxed{\begin{matrix} S_{k,k} & S_{k,k+1} \\ S_{k+1,k} & S_{k+1,k+1} \end{matrix}} & \boxed{S_{k+1,k+2}} & \\ & \boxed{S_{k+1,k+2}} & \boxed{S_{k+2,k+2}} & \\ & & & \ddots \end{pmatrix} \tag{8}$$

For that particular structure of $\mathcal{S}$ the submatrices in boxes correspond to the $\bar{\mathcal{S}}_i$. Such diagonal blocks, that overlap, are similar to the classical block overlap of the Schwarz method when writing in a matrix form for 1D decomposition. Similar ideas have been developed in a pure algebraic context in earlier papers [6,19] for the solution of general sparse linear systems. Because of this link, the preconditioner defined by (7) is referred to as algebraic additive Schwarz for the Schur complement.

One advantage of using the assembled local Schur complements instead of the local Schur complements (like in the Neumann-Neumann [4,8]) is that in the SPD case the assembled Schur complements cannot be singular (as $\mathcal{S}$ is SPD [7]).

2.2. Sparse algebraic additive Schwarz preconditioner

The construction of the proposed local preconditioners can be computationally expensive because the dense matrices $\mathcal{S}_i$ should be factorized. We intend to reduce the storage and the computational cost to form and apply the preconditioner by using sparse approximation of $\bar{\mathcal{S}}_i$ in M_d following the strategy described by (9). The approximation $\tilde{\mathcal{S}}_i$ can be constructed by dropping the elements of $\bar{S}_i$ that are smaller than a given threshold. More precisely, the following symmetric dropping formula can be applied:

$$\hat{s}_{\ell j} = \begin{cases} 0, & \text{if } |\bar{s}_{\ell j}| \leqslant \xi(|\bar{s}_{\ell\ell}| + |\bar{s}_{jj}|), \\ \bar{s}_{\ell j}, & \text{otherwise}, \end{cases} \tag{9}$$

where $\bar{s}_{\ell j}$ denotes the entries of $\bar{\mathcal{S}}_i$. The resulting preconditioner based on these sparse approximations reads

$$M_{sp} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \hat{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}.$$

We notice that such a dropping strategy preserves the symmetry in the symmetric case but it requires to first assemble $\bar{S}_i$ before sparsifying it. From a computing view point, the assembling and the sparsification are cheap compared to the calculation of $\mathcal{S}_i$

3. Mixing *2-levels* of parallelism and domain decomposition techniques

The original idea of non-overlapping domain decomposition method consists into subdividing the mesh into subdomains that are individually mapped to one processor. With this data distribution, each processor P_i can concurrently form the local stiffness matrix (6) and partially factorize it to compute its local Schur complement $\mathcal{S}_i$. This is the first computational phase (*Phase1*) that is performed concurrently and independently by all the processors. The second step (*Phase2*) corresponds to the construction of the preconditioner. Each processor communicates with its neighbors (in the mesh partitioning sense) to assemble its local Schur complement $\bar{\mathcal{S}}_i$, possibly sparsifies it and performs its factorization. This step only requires a few point-to-point communications. Finally, the last step (*Phase3*) is the iterative solution of the interface problem (2). For that purpose, parallel matrix–vector product involving $\mathcal{S}$, the preconditioner $M_{\star}$ and dot-product calculation must be performed. For the matrix–vector product each processor P_i performs its local matrix–vector product involving its local Schur complement and communicates with its neighbors to assemble the computed vector to comply with Eq. (4). Because the preconditioner (7) has a similar form as the Schur complement (4), its parallel application to a vector is implemented similarly. Finally, the dot products are performed as follows: each processor P_i performs its local dot-product and a global reduction is used to assemble the result. In this way, the hybrid implementation can be summarized by the above main three phases. They are further detailed in the *2-level parallel* framework in Section 3.2.

3.1. Motivations for multiple levels of parallelism

Classical parallel implementations (*1-level parallel*) of domain decomposition techniques assign one subdomain per processor. We believe that applying only this paradigm to very large applications has some drawbacks and limitations:

- For many applications, increasing the number of subdomains leads to increasing the number of iterations to converge. If no efficient numerical mechanism, such as coarse space correction for elliptic problems [2,20], is available the convergence rate might be significantly deteriorated and the solution process becomes ineffective. In order to alleviate the numerical growth of the iterations, when the number of subdomains is increased to feed each processor, we might keep the number of subdomains small while handling each subdomain by more than one processor introducing *2-levels* of parallelism. The benefit introduced by the *2-level parallel* implementation is referred to as the "*the numerical improvement*".
- Large 3*D* systems often require a huge amount of data storage so that the memory required to handle each subdomain is not available for each individual processor. On SMP (Symmetric Multi-Processors) node this constraint can be overcome as we might only use a subset of the available processors to allow the exploited processors to access more memory. Although such a solution enables large simulations to be performed, some processors are "wasted", as they are "idle" during the computation. In that context, the simulation executes at an unsatisfying percentage of per-node peak floating-point operation rates. The "idle" processors might contribute to the treatment of the data stored into the memory of the node. This takes advantage of the "idle" processors and runs closer to the peak of per-node performance. We call this "*the parallel performance improvement*" of the *2-level parallel* method.

The main goal of the development of the *2-levels* of parallelism approach is the investigation of numerical methods for the efficient use of parallel modern computers.

3.2. Two-level parallelization strategy

We focus here on the description of the *2-level parallel* algorithm. The first level of parallelism relies on the natural distribution of the subdmains. The second level of parallelism exploits sub-data distribution, where the data structures associated with each subdomain Ω_i are mapped on a subset of processors (groups G_i). In that context, the "local" Schur complement matrices $\mathcal{S}_i$ and $\bar{\mathcal{S}}_i$ are distributed using a 2D block cyclic fashion to comply with the API of the numerical kernels we used, namely SCALAPACK and MUMPS. Such a data distribution is illustrated in Fig. 1.

The *2-level parallel* implementation will be effective for our hybrid solver if its main three phases can be efficiently performed in parallel. The parallel implementation becomes more complex and requires a few MPI communicators to efficiently implement some of the numerical kernels. Let us quickly recall the main numerical kernels of our algorithm and for each of them describe the *2-level parallel* strategy.

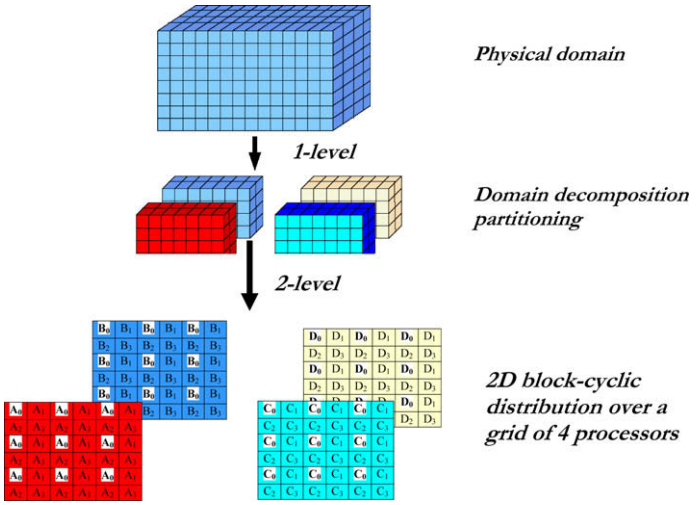


Fig. 1. Multi-level of data and task distribution.

3.2.1. Initialization Phase1

The idea of the 2-level parallel method is to take advantage of the features of a parallel sparse direct solver by using a group of processors on each subdomain. Initially the mesh is partitioned into N subdomains, so that we define as many groups/subsets as the number of subdomains. We create one MPI communicator per group. Each group G_i is logically organized as a 2D grid of processors. The "local" Neumann matrix (6) is first distributed among the processors of the group G_i and a parallel instance of the multifrontal sparse direct solver MUMPS [1] is called by the processors of each of the G_i's. Thanks to its multifrontal approach [10], this solver offers the possibility of computing the Schur complement matrices $S^{(i)}$ that is returned in a 2D block cyclic distribution over the processor grid defined on each G_i. During this first phase, N groups of processors operate concurrently the parallel sparse factorizations.

3.2.2. Preconditioner setup Phase 2

This phase consists into two steps: assembling the local Schur complement $\mathcal{S}_i$ to form the $\bar{\mathcal{S}}_i$, and factorizing the $\bar{\mathcal{S}}_i$. Notice that for the sparse variant we should sparsify the assembled local Schur $\bar{\mathcal{S}}_i$ before factorizing them.

The local Schur complements are dense and distributed in 2D block cyclic arrays spread on the grid of processor defined on G_i. This means that each processor of each G_i only stores blocks of rows and columns of the "local" Schur complement. For the sake of efficiency we do not want to centralize $\mathcal{S}_i$ to perform the assembling. For this purpose, each processor of the group that stores part of the "local" Schur complement, should know the identifier of the processors handling the corresponding part of the neighboring subdomains. In that respect a particular data structure is initialized in a preprocessing step. It enables each processor of a group G_i to know the identifier of the processors in a neighboring group G_j with which it shares common blocks of its local Schur complement $\mathcal{S}_i$. For example, if we have a subdomain Ω_i that shares an interface with a subdomain Ω_j, the group G_i is a neighbor of the group G_j. As the Schur complement is 2D block cyclically distributed over G_i a processor p_i^ℓ of G_i holds a part of the local Schur complement. The data structure enables the processor p_i^ℓ belonging to the group G_i, to know the processors identifier list of G_j that share this part of the interface with it. In this way, each processor can directly communicate (point-to-point) with its neighboring processors and assemble its own parts of the "local" Schur complement. We notice that, for the large simulations, the local Schur matrices are large, thus the amount of data to be exchanged can be very large. Using this fully distributed algorithm to assemble in parallel the "local" Schur complement is critical to get fast executions.

Once the "local" Schur complement are assembled in a 2D block cyclic array similar to the one used for $\mathcal{S}_i$, each group G_i can concurrently perform in parallel the factorization using the ScaLAPACK kernels for the dense variant of the preconditioner. For the sparse preconditioner, the sparsification is performed in parallel; each processor sparsifies its own part of the local Schur complements. The sparse preconditioner is then factorized in parallel by G_i.

3.2.3. Iterative loop Phase 3

This phase involves three numerical kernels that are: the matrix–vector product, the preconditioner application and finally the dot product. Each local Schur matrix is distributed over the group so that both PBLAS and SCALAPACK can be used easily.

For the matrix–vector product, the implementation is performed using the PBLAS routines to multiply the distributed local Schur complement with the distributed vector. The resulting distributed vector is updated directly between neighboring subdomains as each processor of a group knows with whom it should communicate.

The preconditioner application relies either on SCALAPACK kernels for the dense M_d preconditioner or on sparse direct solve MUMPS for the sparse M_{sp} preconditioner. The resulting distributed vector, is updated as for the matrix–vector product.

For the dot product calculation, each processor owning the distributed vectors performs its local dot product then the results are summed using a simple global reduction.

Because each step can be parallelized, the iterative loop calculation greatly benefits from the *2-level parallel* implementation. The complexity of the actual code is of course by no means reflected by the simplicity of the above exposure.

4. Experimental framework

In this section, we first introduce in Section 4.1 the computational framework considered for our parallel numerical experiments. In Section 4.2 we describe our model problems. We consider classes of problems related to the solution of the linear elasticity equations with constraints such as rigid bodies and cyclic conditions. These constraints are handled using Lagrange multipliers, that give rise to symmetric indefinite augmented systems. Such linear systems are preferably solved using the MINRES [18] Krylov subspace method, that can be implemented using only a few vectors thanks to the symmetry property that enables the use of short recurrences. In our study, because we intend to perform comparisons in term of computing performance and also in term of accuracy, we better use GMRES that is proved backward stable [9,17]. All the problems presented in this paper have been generated using the Samcef V12.1-02 finite element software for nonlinear analysis, Mecano developed by Samtech http://www.samcef.com/.

4.1. Parallel platforms

Our target parallel machine is an IBM JS21 supercomputer installed at CERFACS to address diverse applications in science and engineering. It works currently with a peak computing performance of 2.2 TeraFlops. This is a 4-core blade server for applications requiring 64-bit computation.

This paragraph provides more detailed information about the IBM PowerPC 970MP microprocessor, that is the processor of the BladeCenter JS21.

- The BladeCenter JS21 leverages the high-performance, low-power 64-bit IBM PowerPC 970MP microprocessor.
- The 4-core configuration comprises two dual-core PowerPC 970MP processors running at 2.5 GHz.
- Each processor core includes 32/64 KB L1 (data/instruction) and 1 MB (non-shared) L2 cache.
- Each node is equipped with 8 GBytes of main memory.
- The AltiVec is an extension to the IBM PowerPC Architecture. It defines additional registers and instructions to support single-instruction multiple-data (SIMD) operations that accelerate data-intensive tasks.

The BladeCenter JS21 is supported by the AIX 5L, Red Hat Enterprise Linux, and SUSE Linux Enterprise Server (SLES) operating systems. This latter is installed on our experimental JS21. The communication is supported through the use of Myrinet2000 network offering a bandwidth of 838 MBytes/s between nodes and a latency of 3.2 μs.

4.2. Model problems

In this paper, we focus on a specific engineering area, the structural mechanics. Those simulations often involve the discretization of linear or nonlinear 3*D* PDE on very large meshes leading to systems of equations with millions of unknowns. The use of large high performance computers is mandatory to solve these problems. They can benefit from such *2-level parallel* implementation of domain decomposition solver. We notice that such an approach can be appropriated also for some simulations in the context of seismic wave propagation. For more details and experiments on this subject, we refer the reader to [16].

We consider here a few real life problems from structural mechanics applications. Those examples are generated using Samcef-Mecano V12.1-02. Samcef-Mecano is a general purpose finite element software that solves nonlinear structural and mechanical problems. It minimizes the potential energy using the displacement (translations and/or rotations) as unknowns. For each kinematic constraint (linear constraint or kinematic joint), a Lagrange multiplier is automatically generated. In order to have a good numerical behaviour, an augmented Lagrangian method is used. The modified problem consists in finding the minimum of the potential F^*:

$$F^*(q) = F(q) + k\lambda\phi + \frac{p}{2}\Phi^T\Phi,$$

where q is the degrees of freedom vector (translations and/or rotations), k is the kinematic constraint scaling factor, p is the kinematic constraint penalty factor, ϕ is the kinematic constraint vector and λ is the Lagrange multiplier vector.

The equations of motion of the structure discretized by finite elements take the general form

$$M\ddot{q} + f^{\text{int}} = f^{\text{ext}},$$

where the notation is simplified by including in the internal forces f^{int} the contribution of the kinematic constraints and that of the elastic, plastic, damping, friction,... forces. Three types of analysis can be performed:

1. Static analysis.
2. Kinematic or quasi-static analysis.
3. Dynamic analysis.

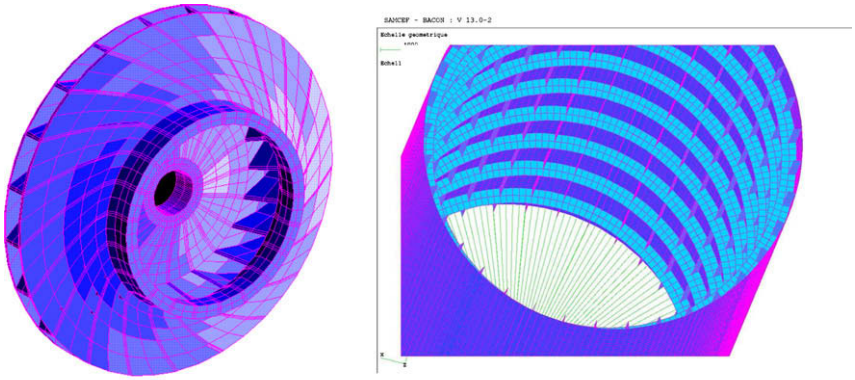
At each time step, a set of nonlinear equations has to be solved and a Newton–Raphson scheme is used in order to solve this nonlinear problem. These equations express the equilibrium of the system at a given time. In a static analysis, these equations take stiffness effects (linear or not) into account. In a kinematic analysis, the effects due to the kinematic velocities are added to the effects taken into account by the static analysis. Finally, the dynamic analysis takes all the effects of the kinematic analysis into account including also inertia effects, that do not appear in the static and the kinematic analysis.

For the numerical examples considered here, a static computation is performed. The materials are elastic: the relation between the stress σ and the strain ϵ is given by $\sigma = H\epsilon$, where H is the Hooks matrix. For each test case we run our solver on the matrix generated during the first iteration of the first time step.

The geometry of the examples are displayed in Fig. 2. The first problem is displayed in Fig. 2(a), corresponds to an impeller. This case represents a 90° sector of an impeller. It is composed of 3D volume elements. Cyclic conditions are added using elements that link displacements of the slaves nodes on one side of the sector, to master facets on the other side of the sector. These conditions are taking into account using elements with three Lagrange multipliers. Angular velocities are introduced on the complete structure and centrifugal loads are computed on the basis of the angular velocities and of the mass representation. This example is called Rouet in the sequel, the associated linear system is symmetric indefinite.

Secondly, a parameterized barrel (section of a fuselage) is depicted in Fig. 2(b). It is composed of its skin, stringers (longitudinal) and frames (circumferential, in light blue in Fig. 2 (b)). Midlinn shell elements are used: each node has 6 unknowns (3 translations and 3 rotations). On one extremity of the fuselage all the degrees of freedom are fixed. On the other extremity a rigid body element is added: all the degrees of freedom of the nodes are linked to the displacement of the master node of the element. In order to represent this dependency Lagrange multipliers are added. A force perpendicular to the axis of the fuselage is applied on the master node. This last test example is referred to as Fuselage and the associated linear system is also symmetric indefinite.

In Table 1 we display for the different mesh geometries the various sizes of the problems we have experimented. For each problem, we give the number of finite elements and the number of degrees of freedom.



(a) A wheel: Rouet example. (b) Part of a Fuselage.

Fig. 2. Various structural mechanics meshes.

Table 1
Characteristics of the various structural mechanics problems.

# Elements	# Degrees of freedoms
Rouet example	
337,000	$1.3 \cdot 10^6$
Fuselage example	
500,000	$3.3 \cdot 10^6$
750,000	$4.8 \cdot 10^6$
1,000,000	$6.5 \cdot 10^6$

5. Parallel performance studies

This section is the core of the parallel study where we illustrate through a set of experiments the advantage of the *2-level parallel* method. This subsection is devoted to the presentation and analysis of the parallel performance of both preconditioners. It is believed that parallel performance is the most important means of reducing turn around time and computational cost of real applications. In this context, we consider experiments where we increase the number of processors while the size of the initial linear system (i.e., mesh size) is kept constant. Such experiments mainly emphasize the interest of parallel computation in reducing the elapsed time to solve a problem of a prescribed size. For those indefinite systems we choose full-GMRES as Krylov solver and consider the ICGS (Iterative Classical Gram-Schmidt) orthogonalization variant that provides us with the best trade-off between numerical orthogonality and parallel performance (as the dot-product are merged). The initial guess is always the zero vector and convergence is detected when the relative residual $\frac{\|Sx_i - f\|}{\|f\|}$ less than 10^{-8} or when 300 steps have been unsuccessfully performed. We mention that only right preconditioner is considered so that the stopping criterion is independent from the preconditioner which enables us to make fair comparisons between the various approaches.

In the next sections we study the numerical benefits and parallel performance advantages of the *2-level parallel* approach. We draw the attention of the reader on the fact that, in those sections, the number of processors and the number of subdomains are most of the time different.

5.1. Numerical benefits

The numerical attractive features of the *2-level parallel* approach is that increasing the number of processors to speedup the solution of large linear systems does not imply increasing the number of iterations to converge as it is often the case with the *1-level parallel* approach.

This strategy is very attractive in the case where we have a system with many right hand side to solve. In this case we have interest to keep the number of iterations as small as possible. We report in Table 2 both the number of iterations and the parallel computing time spent in the iterative loop, for the problems depicted in Section 4.2. For each problem,

Table 2
Numerical performance and advantage of the *2-level parallel* method compared to the standard *1-level parallel* method for the Fuselage and Rouet problems and for different decomposition proposed.

# Total processors	Algo	# Subdomains	# Processors/subdomain	# Iter	Iterative loop time
Fuselage 1 million elements with $6.5 \cdot 10^6$ dof					
16 processors	*1-level parallel*	16	1	147	77.9
	2-level parallel	8	2	98	51.4
32 processors	*1-level parallel*	32	1	176	58.1
	2-level parallel	16	2	147	44.8
	2-level parallel	8	4	98	32.5
64 processors	*1-level parallel*	64	1	226	54.2
	2-level parallel	32	2	176	40.1
	2-level parallel	16	4	147	31.3
	2-level parallel	8	8	98	27.4
Fuselage 0.5 million elements with $3.3 \cdot 10^6$ dof					
8 processors	*1-level parallel*	8	1	92	46.2
	2-level parallel	4	2	38	18.6
16 processors	*1-level parallel*	16	1	124	37.2
	2-level parallel	8	2	92	25.9
	2-level parallel	4	4	38	10.1
32 processors	*1-level parallel*	32	1	169	32.3
	2-level parallel	16	2	124	22.1
	2-level parallel	8	4	92	14.3
	2-level parallel	4	8	38	11.8
Rouet 0.33 million elements with $1.3 \cdot 10^6$ dof					
16 processors	*1-level parallel*	16	1	79	60.8
	2-level parallel	8	2	59	34.1
32 processors	*1-level parallel*	32	1	106	44.5
	2-level parallel	16	2	79	38.6
	2-level parallel	8	4	59	21.1
64 processors	*1-level parallel*	64	1	156	42.1
	2-level parallel	32	2	106	22.4
	2-level parallel	16	4	79	26.2
	2-level parallel	8	8	59	25.5

we choose a fixed number of processors and vary the number of subdomains that are allocated to different number of processors.

In this table it can be seen that decreasing the number of subdomains reduces the number of iterations. The parallel implementations of the numerical kernels involved in the iterative loop is efficient enough to speedup the solution time. On the largest Fuselage example, when we have 32 processors, standard (*1-level parallel*) implementation partitions the mesh into 32 subdomains; it requires 176 iterations to convergence and get the solution in 58.1 s. With the *2-level parallel* implementation, either 16 or 8 subdomains can be used. The 16 subdomain partition requires 147 iterations performed in 44.8 s and the 8 subdomain calculation needs 98 iterations performed in 32.5 s. This example illustrates the advantage of the *2-level parallel* implementation from a numerical viewpoint. The main goal of this section was to highlight the numerical and computational assets of the *2-level parallel* implementation to enhance the parallel performance of the iterative solution. The other components of the solvers, setup time of the preconditioner and overall computing time are detailed in the tables depicted in the next section.

5.2. Parallel performance benefits

When running large simulations with a few sub-domains, the memory required by each sub-domain is so large than only one sub-domain can fit into the memory of one SMP node. If standard parallel approach is considered (*1-level parallel*), only one processor of each SMP node is used to perform the simulation, thus leaving the remaining processors idle. In this context, the goal of the *2-level parallel* method, is to exploit the computing facilities of the remaining processors and allow them to contribute to the computation.

In Tables 3–5, for the various test problems, we depict the performance of each step of the solver as well as the overall computing time when the number of domains and number of processors per domain is varied. In these tables, the *"Time in iterative loop"* corresponds to the elapsed time spent in the preconditioned Krylov solver, *"Preconditioner setup time"* is the computing time to factorize the local assembled, possibly sparsified, Schur complement matrices and finally *"Initialization time"* corresponds to the time used by the parallel sparse direct solver to factorize the local interior matrices and compute the local Schur complement.

We report in Table 3, the performance results of the *2-level parallel* method for the Fuselage with 6.5 million degrees of freedom. We use the *2-level parallel* algorithm only for simulations that leave idle processors when the standard (*1-level parallel*) algorithm is run due to memory constraints. In that case, the 8 or 16 subdomain decompositions require respectively 7 GBytes and 5 GBytes of memory; so that the *1-level parallel* implementation can only exploit one of the four SMP processors. That means that the 8 subdomain simulation using the *1-level parallel* approach requires the use of 8 SMP nodes, where only one processor per node is used; which leaves 24 idle processors. Even worse, the 16 subdomain simulation leaves 48 idle processors. In such a context the benefit of the *2-level parallel* approach is clear. The parallel performance of the M_d and M_{sp} are reported in this table and similar results are given for the other test problems (Table 4 for the Fuselage with 3.3 millions degrees of freedom and Table 5 for the Rouet with 1.3 million degrees of freedom)

Table 3
Detailed parallel performance of the *2-level parallel* method for the Fuselage problem with about one million elements and 6.5 Mdof when the number of subdomains is varied, for the various variants of the preconditioner.

# Subdomains or SMP-nodes	**8**			**16**			**32**			**64**
Size of the Schur	40,200			61,251			87,294			122,190
# Processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*
	1	2	4	1	2	4	1	2	4	1
Total solution time										
M_d	525.1	354.1	254.4	217.2	140.8	101.0	124.1	91.2	65.7	83.0
M_{sp}	322.8	234.4	188.3	120.1	88.3	73.0	87.9	73.7	54.1	65.1
Time in the iterative loop										
M_d	94.1	51.5	32.5	77.9	44.8	31.3	58.1	40.8	22.7	54.2
M_{sp}	57.6	34.3	19.9	50.3	29.6	21.6	38.9	31.2	18.2	40.7
# Iterations										
M_d	98			147			176			226
M_{sp}	101			148			177			226
Time per iteration										
M_d	0.96	0.53	0.33	0.53	0.30	0.21	0.33	0.23	0.13	0.24
M_{sp}	0.57	0.34	0.20	0.34	0.20	0.15	0.22	0.18	0.10	0.18
Preconditioner setup time										
M_d	208.0	124.6	70.8	89.0	52.7	30.4	30.0	20.4	13.0	15.0
M_{sp}	42.2	22.1	17.4	19.5	15.4	12.1	13.0	12.6	7.9	11.4
Initialization time										
All precond.	223	178	151	50	43	39	36	30	24	13.1

Table 4
Detailed parallel performance of the *2-level parallel* method for the Fuselage problem with about 0.5 million elements and 3.3 Mdof when the number of subdomains is varied, for the various variants of the preconditioner.

# Subdomains or SMP-nodes	**4**			**8**			**16**			**32**
Size of the Schur	17,568			28,644			43,914			62,928
# Processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*
	1	2	4	1	2	4	1	2	4	1
Total solution time										
M_d	411.1	278.6	203.0	199.4	132.0	91.7	107.7	72.3	48.4	59.3
M_{sp}	311.6	210.8	175.3	125.1	91.5	72.0	72.9	52.3	40.3	45.9
Time in the iterative loop										
M_d	32.5	18.6	10.1	46.2	25.9	14.3	37.2	22.1	13.4	32.2
M_{sp}	22.5	14.0	9.4	30.3	18.1	10.7	25.9	16.2	12.2	23.3
# Iterations										
M_d	38			92			124			169
M_{sp}	40			92			124			169
Time per iteration										
M_d	0.85	0.49	0.27	0.50	0.28	0.15	0.30	0.18	0.11	0.19
M_{sp}	0.56	0.35	0.23	0.33	0.20	0.12	0.21	0.13	0.10	0.14
Preconditioner setup time										
M_d	182.0	100.0	55.8	79.1	46.0	26.4	37.9	24.3	15.0	13.7
M_{sp}	92.5	36.9	28.9	20.8	13.3	10.3	14.4	10.1	8.2	9.3
Initialization time										
All precond.	196.7	160	137	74	60	51	32.7	26	20	13.2

Table 5
Detailed parallel performance of the *2-level parallel* method for the Rouet problem with about 0.33 million elements and 1.3 Mdof when the number of subdomains is varied, for the various variants of the preconditioner.

# Subdomains	**8**			**16**			**32**			**64**
Size of the Schur	31,535			49,572			73,146			112,000
# Processors per subdomain	*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*	*2-levels parallel*		*1-level parallel*
	1	2	4	1	2	4	1	2	4	1
Total solution time										
M_d	453.7	303.7	226.9	264.6	174.7	126.8	110.9	69.4	59.0	70.1
M_{sp}	277.5	219.6	184.8	151.7	124.8	101.6	86.5	70.0	59.8	51.4
Time in the iterative loop										
M_d	57.2	34.3	21.1	60.8	38.7	26.2	44.5	22.4	25.1	42.1
M_{sp}	42.0	26.8	18.8	47.9	28.8	20.3	37.6	27.4	26.1	35.6
# Iterations										
M_d	59			79			106			156
M_{sp}	60			87			114			162
Time per iteration										
M_d	0.97	0.58	0.36	0.77	0.49	0.33	0.42	0.21	0.24	0.27
M_{sp}	0.70	0.45	0.31	0.55	0.33	0.23	0.33	0.24	0.23	0.22
Preconditioner setup time										
M_d	235.0	127.1	75.0	137.0	74.8	43.9	43.5	27.7	17.2	19.0
M_{sp}	74.0	50.5	35.2	37.0	34.8	24.6	26.0	23.3	16.9	6.8
Initialization time										
All precond.	161.5	142.3	130.8	66.8	61.2	56.7	22.8	19.3	16.7	9.0

To study the parallel behaviour of the *2-level parallel* implementation we discuss the efficiency of the three main steps of the algorithm.

- The initialization *Phase 1*: this step mainly consists in calling the parallel version of the sparse direct solver as "black-box" with the best found sets of parameters to get the highest performance.
- Preconditioner setup *Phase 2*: This phase includes two steps, assembling the local Schur complement, and the factorization of either M_d or M_{sp} depending on the selected variant. The factorization of M_d is performed using ScaLAPACK, while M_{sp} is factorize using a parallel instance of MUMPS.The results reported in Tables 3–5 highlight the advantages of the *2-level par-*

allel algorithm. For the dense preconditioner M_d, we can observe that even if we not take advantage of all the working nodes and use only 2 of the 4 available processors to perform the preconditioner setup phase, the benefit is considerable. The computing time is divided by around 1.8 for all test cases. For the sparse preconditioner, the gain is also important, it varies between 1.5 and 2.The objective of the *2-level parallel* method is to take advantage of all the available resources to complete the simulation. If the four processors were used, the performance is even higher. For the dense M_d preconditioner, the calculation is performed around 3 times faster thanks to the efficiency of the parallel dense linear algebra kernels of ScaLapack. For the sparse preconditioner, the speedups vary between 1.5 and 3. This speedup is also significant for the sparse preconditioner M_{sp}.

Finally, it can be noticed that, the best execution times are obtained using the *2-level parallel* method for all test problems. The *2-level parallel* method is needed to attain the best parallel performance.

- The *Phase 3* of the method is the iterative loop, that mainly involves three numerical kernels that are: the matrix–vector product implemented using PBLAS routines; the preconditioner application that relies either on scalapack kernels for M_d or mumps for M_{sp} and a global reduction for the dot-product calculation. The results reported in these tables, show similar speedups as the ones observed for *Phase 2* (preconditioner setup). For the dense preconditioner the execution of the iterative loop is 2–3 times faster than for the *1-level parallel* algorithm. Also, for the sparse preconditioner, the convergence is achieved 1.7–3 times faster.

As a general comment, it can be seen in all the tables that most of the numerical kernels greatly benefit from the two levels of parallelism. In those experiments the initialization is the computing part that does not scale well; on that computational kernel we do hope to benefit from new progresses in parallel sparse direct packages.

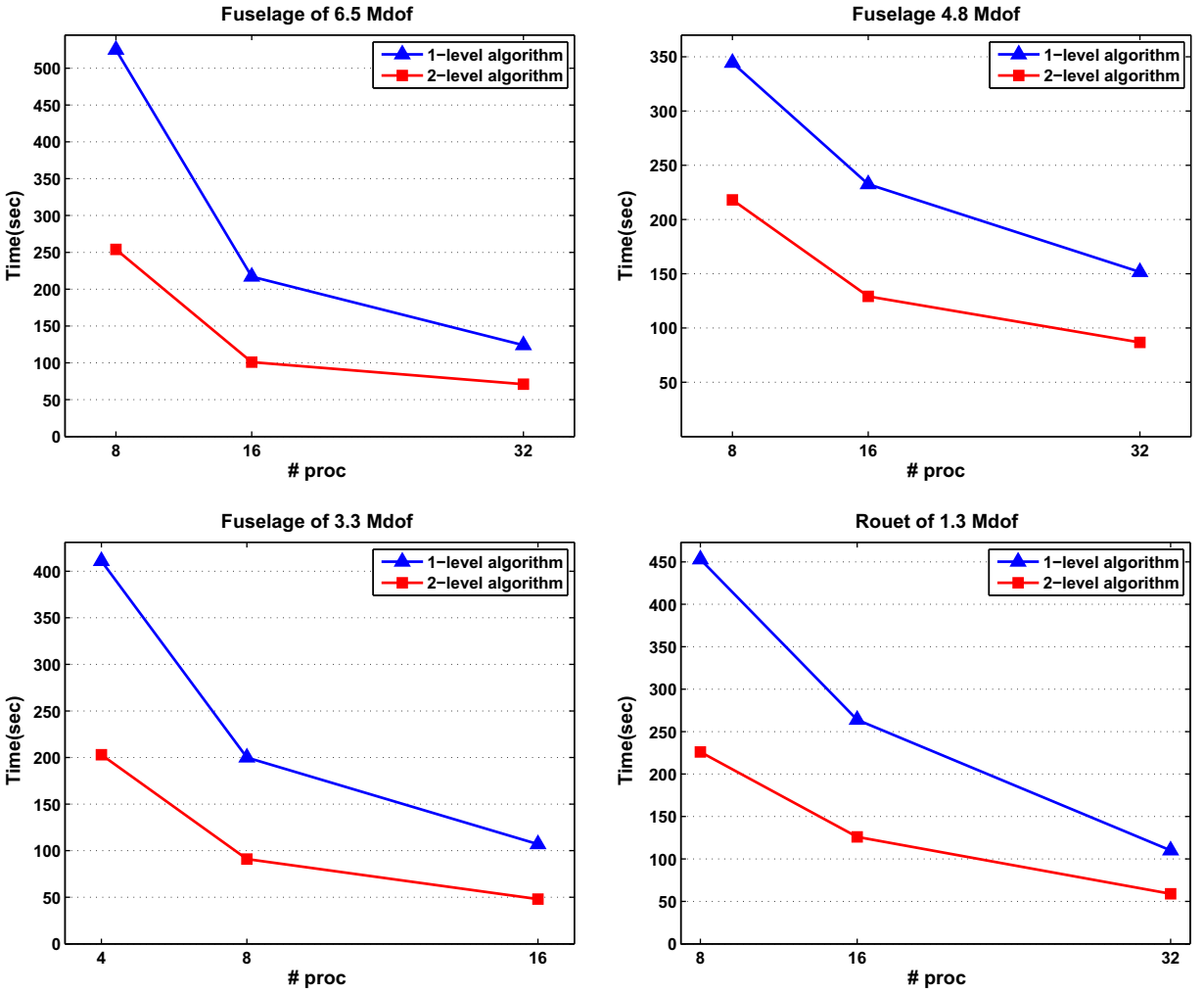


Fig. 3. Parallel performance of the total time of the *2-level parallel* method for the Fuselage and the Rouet test cases, when consuming the same number of SMP nodes as the *1-level parallel* method.

Finally, to summarize the time saving enabled by the *2-level parallel* implementation we display in Fig. 3 the global computing time for the different decompositions. For the sake of readability, we only display the results for the dense preconditioner. Those curves illustrate the benefit of the *2-level parallel* implementation that enables us to get much better computing throughput out of the SMP nodes. In addition, the larger the problem size, the larger the benefit/gain.

6. Concluding remarks

In this paper, we have investigated the numerical behaviour of our preconditioner and the benefit of multiple levels of parallelism for the solution of linear systems arising in three dimensional structural mechanics problems representative of difficulties encountered in this application area. We have demonstrated on 3*D* real life applications the attractive features of our approach.

Such an implementation is attractive in situation where the increase of the number of iterations is significant when the number of domains is increased. In the context of parallel SMP platforms, we have illustrated the benefit of a *2-level parallel* implementation when the memory storage is the main bottleneck. In this case, the *2-level parallel* algorithm can be of great interest. We believe that this implementation could also be very suited for large calculations in many applications.

For clusters of SMP, both message passing and multithreading can be combined. Mixing threads within node and MPI communication between the nodes appears natural. Several researchers have used this mixed model with reasonable success. Future work will address such implementations for hybrid linear solvers.

References

[1] P.R. Amestoy, I.S. Duff, J. Koster, J.-Y. L'Excellent, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications 23 (1) (2001) 15–41.
[2] P.E. Bjørstad, O.B. Widlund, Iterative methods for the solution of elliptic problems on regions partitioned into substructures, SIAM Journal of Numerical Analysis 23 (6) (1986) 1093–1120.
[3] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM Press, 1997.
[4] J.-F. Bourgat, R. Glowinski, P. Le Tallec, M. Vidrascu, Variational formulation and algorithm for trace operator in domain decomposition calculations, in: Tony Chan, Roland Glowinski, Jacques Périaux, Olof Widlund (Eds.), Domain Decomposition Methods, SIAM, Philadelphia, PA, 1989, pp. 3–16.
[5] J.H. Bramble, J.E. Pasciak, A.H. Schatz, The construction of preconditioners for elliptic problems by substructuring I, Mathematical Computation 47 (175) (1986) 103–134.
[6] X.-C. Cai, Y. Saad, Overlapping domain decomposition algorithms for general sparse matrices, Numerical Linear Algebra with Applications 3 (1996) 221–237.
[7] L.M. Carvalho, L. Giraud, G. Meurant, Local preconditioners for two-level non-overlapping domain decomposition methods, Numerical Linear Algebra with Applications 8 (4) (2001) 207–227.
[8] Y.-H. De Roeck, P. Le Tallec, Analysis and test of a local domain decomposition preconditioner. in: Roland Glowinski, Yuri Kuznetsov, Gérard Meurant, Jacques Périaux, Olof Widlund, (Eds.), Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations, SIAM, Philadelphia, PA, 1991, pp. 112–128.
[9] J. Drkošová, M. Rozložník, Z. Strakoš, A. Greenbaum, Numerical stability of the GMRES method, BIT 35 (1995) 309–330.
[10] I.S. Duff, J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear systems, ACM Transactions on Mathematical Software 9 (1983) 302–325.
[11] V. Frayssé, L. Giraud, A set of conjugate gradient routines for real and complex arithmetics. Technical Report TR/PA/00/47, CERFACS, Toulouse, France, 2000. Public domain software availabe on http://www.cerfacs.fr/algor/Softs.
[12] V. Frayssé, L. Giraud, S. Gratton, A set of flexible GMRES routines for real and complex arithmetics on high performance computers. Technical Report TR/PA/06/09, CERFACS, Toulouse, France, 2006. This report supersedes TR/PA/98/07.
[13] V. Frayssé, L. Giraud, S. Gratton, J. Langou, Algorithm 842 a set of GMRES routines for real complex arithmetics on high performance computers, ACM Transactions on Mathematical Software 31 (2) (2005) 228–238. Preliminary version available as CERFACS TR/PA/03/3, Public domain software Availabe on http://www.cerfacs.fr/algor/Softs..
[14] L. Giraud, A. Haidar, L.T. Watson, Parallel scalability study of hybrid preconditioners in three dimensions, Parallel Computing 34 (2008) 363–379.
[15] W.D. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of MPI message passing interface standard, Parallel Computing 22 (6) (1996).
[16] A. Haidar, On the parallel scalability of hybrid solvers for large 3D problems. Ph.D. dissertation, INPT, June 2008, TH/PA/08/57.
[17] C. Paige, M. Rozložník, Z. Strakoš, Modified Gram-Schmidt (MGS), least-squares, and backward stability of MGS-GMRES, SIAM Journal on Matrix Analysis and Applications 28 (1) (2006) 264–284.
[18] C.C. Paige, M.A. Saunders, Solution of sparse indefinite systems of linear equations, SIAM Journal on Numerical Analysis 12 (1975) 617–629.
[19] G. Radicati, Y. Robert, Parallel conjugate gradient-like algorithms for solving nonsymmetric linear systems on a vector multiprocessor, Parallel Computing 11 (1989) 223–239.
[20] B.F. Smith, P. Bjørstad, W. Gropp, Domain Decomposition, first ed., Parallel Multilevel Methods for Elliptic Partial Differential Equations, Cambridge University Press, New York, 1996.
[21] Mathew T. Domain, Decomposition Methods for the Numerical Solution of Partial Differential Equations, Springer Lecture Notes in Computational Science and Engineering, Springer, 2008.